This is the submitted version of the article *Evaluating the effectiveness of the abstract transaction model in testing web services transactions, Casado R., Tuya, J., Younas, M., Concurrency and Computation: Practice and Experience,* Copyright © 2015. DOI: 10.1002/cpe.2851. Volume 27, Issue 4, 25 March 2015. Pages 765-781

Evaluating the effectiveness of the abstract transaction model in testing web services transactions

Rubén Casado[1, *, †], Javier Tuya[1] and Muhammad Younas[2]

[1]*Department of Computing, University of Oviedo, Asturias, Spain*
[2]*Department of Computing and Communication Technologies, Oxford Brookes University, Oxford, UK*

* Correspondence to: Rubén Casado, Campus Universitario de Gijón, edificio polivalente, 2.7.12, 33204 Gijón, Spain

† E-mail: rcasado@uniovi.es

SUMMARY

Web services transactions are used to build efficient and reliable web applications which are distributed across the Internet and are accessed by multiple simultaneous users. Current research develops various models and protocols in order to improve the performance and reliability of web services transactions. However, there is little research on testing the different models and protocols of web services transactions. This paper presents an abstract transaction model that patterns different web services transactions standards. This model is capable of deriving concrete models in order to automatically generate test cases for different web services transactions standards. The proposed model is implemented as a prototype system and is evaluated using a case of the Jboss Transaction. The evaluation shows that the proposed system has the capability to automatically generate test cases and detect possible failures of transactions running under different web services transactions standards.

KEY WORDS: transactions; web services; testing, failure detection, reliability;

1. INTRODUCTION

Web services (WS) are software applications which provide uniform interfaces for interaction and communication with other web applications in a dynamic manner. They also provide compositional facilities such that different web services can be composed to enact an integrated service that provides enhanced functionalities. Web services transactions (or WS Transactions) are defined as sequences of operations that are executed under certain constraints in order to maintain application correctness and data consistency. The fundamental principle of WS Transactions is to provide web services applications with reliability and efficiency. In order to ensure reliable execution of web services it is crucial that their activities are modelled as transactions such that they achieve an mutually agreed outcome [1].

Numerous models and protocols have been developed for WS Transactions. These include standard models and protocols such as Business Transaction Protocol (BTP) [2], Web Services Business Activity (WS-BA) [3] and Web Services Transaction

Management (WS-TXM) [4]. However they fall short of testing WS Transactions, for instance, in terms of reliability and failures [5]. The process of testing WS Transactions is not trivial due to several reasons. First, WS Transactions are more complex compared to classical transactions as they involve cooperation among multiple parties, span autonomous and independent partners, and may have long duration. Thus WS Transactions have more intricate sequence of operations and execution environment. Second, WS Transactions do not have a homogeneous transaction model such as the ACID (Atomicity, Consistency, Isolation and Durability) model. Instead they are characterized by a diversity of transaction models such as BTP, WS-BA and WS-TXM. Such diversity of models also complicates the process of testing WS Transactions. Third, various kinds of failures may happen during the processing of WS Transactions, including: (i) technical failures such as communication, system and software failures can occur. Such failures result in loss of messages, processing of services, etc (ii) service level failures such as service acquisition failures wherein services cannot be acquired due to unavailability of the desired services, payment problems, or service cancellation.

In [6, 7] we proposed the Abstract Transaction Model (AbTM) for testing the WS Transactions. AbTM serves as a template for modelling different WS Transactions standards. AbTM identifies the different roles involved in a WS Transaction, the relationship between them and also models the behaviour of each one during the transaction life cycle. In this paper we extend the AbTM in order to automatically generate test cases for WS Transactions. We also evaluate the AbTM through the development of a prototype system using a case study of the Jboss Transaction [8]. The novel features and contributions of the work presented in this paper are as follows:

(i) To automatically generate the abstract test cases and map them to different WS Transactions standard (such as BTP, WS-BA, etc).

(ii) To automatically compare the expected and actual outcomes in order to identify failures in WS Transactions running under different WS Transactions standards.

(iii) To perform testing and evaluation using the standard case study of *Night Out*, which is provided by Jboss [8] in their implementation of the WS-BA standard.

The rest of the paper is organized as follows. Section 2 gives an analysis of WS Transaction models and standards. Section 3 presents the Abstract Transaction Model. Section 4 illustrates the process of modelling WS Transaction standards. Section 5 presents the AbTM-based approach for testing WS Transactions. The evaluation of our approach is discussed in Section 6. Section 7 reviews existing work on WS Transactions. Conclusions are presented in Section 8.

2. WS TRANSACTION STANDARDS

WS Transactions are based on various models ranging from classical ACID models to advanced or extended transaction models. Two Phase Commit (2PC) protocol and its

variants [10] have commonly been used for maintaining ACID properties. ACID properties are vital for WS Transactions that need strict data consistency. However, they are not suitable for long running applications due to resource locking/blocking problems. Advanced transaction models have been developed to address 2PC and ACID related issues. These includes, nested transaction model [11], SAGA model [12], open-nested [13], Split-join [14], Contracts [15], Flex [16], and WebTram [17]. The underlying strategy of these models is to allow compensation of partially completed transactions in order to maintain data consistency and reliability.

Based on the above transaction models several standard specifications have been developed for WS Transactions. For instance, BTP [2] adapts 2PC for short lived transactions and nested transaction model for long-lived transactions. WS-CAF [4] is a set of WS specifications for applications composed of multiple WS used in combination. WS-CAF uses WT-TXM to manage the transactions. WT-TXM defines three models, ACID Transaction (TXACID), Long Running Transaction (TXLRA) and Business Transaction Process (TXBP) that address different scenarios. Web Services Atomic Transactions (WS-AT) [18] and WS-BA [3] are built on top of Web Services Coordination (WS-COOR) [19] and they follow its coordination mechanism. WS-AT follows 2PC protocol while WS-BA uses the SAGA model.

The above standards are summarized and analysed in Table I. 'Coordination' represents whether a particular standard provides coordination facilities. 'Short' and 'Long' represent that the underlying model is respectively based on ACID properties and advanced transaction models. 'Related' represents the remaining standards which belong to a same family.

Table I. WS Transaction standards

Standards	Coordination	Short	Long	Related
BTP	✓	2PC	Nested	×
WS-CAF	×	×	×	WS-TXM
WS-TXM	✓	×	×	TXACD, TXLRA, TXBP
TXACID	×	2PC	×	WS-TXM
TXLRA	×	×	SAGA	WS-TXM
TXBP	×	×	Open	WS-TXM
WS-COOR	✓	×	×	WS-AT, WS-BA
WS-AT	×	2PC	×	WS-COOR
WS-BA	×	×	SAGA	WS-COOR

It is observed that all standards separate the coordination and the management of the subtransactions and also distinguish short-lived transactions from long-lived transactions. It is also observed that these standards have proprietary definitions of their underlying transaction models despite the fact they are based on similar concepts. This makes it difficult to use them in a uniform way. Our analysis shows that WS Transactions standards are not homogenous and have different processing and testing requirements. Thus it is not practical (nor easier) to test just a single WS Transaction model and evaluate its reliability. In the next section we present the proposed model that automatically represents and tests different WS transactions standards.

3. THE ABSTRACT TRANSACTION MODEL

A WS Transaction is a service composition where the participants collaborate to create a new and more functional unit of work. Some activities are always presented in the transaction management, such as creation or termination. For example, in an application that allow booking different services for a night out (e.g., theatre tickets, restaurant and taxi), the client side of the application starts the transaction since it has the customer information. Also it finishes the transaction due to it knows the customer´s requirements about the whole reservation process. A WS Transaction has a hierarchical structure, i.e., is composed by partially independent activities (e.g. theatre tickets and restaurant reservation) where each activity may be a compound activity (e.g. the payment activity where the money transfer is a classic ACID transaction). Different standards can be used to manage the WS Transactions.

The abstract model aims to model (or pattern) different WS Transaction standards discussed above. It is designed using the well-known Unified Modeling Language (UML) statecharts notations which reflect the event-driven (message communication) nature of WS Transactions.

Definition of WS Transaction

A *WS Transaction*, *wT*, is defined as a set $S=\{s_1,\ldots,s_n\}$ of activities (or subtransactions) which are executed in order to achieve an agreed outcome in a WS application. Each *wT* is associated with one *Coordinator*, *k*, while each subtransaction, s_i, is executed by an *Executor*, e_i, as defined below. *Transaction context* is defined as the set of functional information and transaction configuration shared by the activities. Each s_i could be a single level subtransaction or it may have nested subtransactions, denoted wT_c. In the proposed model, nested transactions are related in a *parent:child* relationship. Figure 1 shows such relationship wherein wT_p, is a parent of s_1, s_2 and s_3. s_1 (or wT_c) is in turn a parent of s_{c1} and s_{c2}.

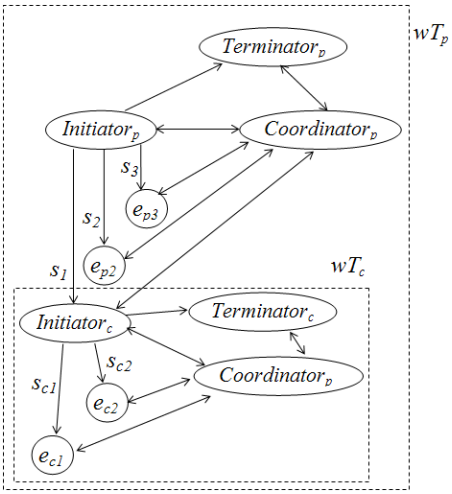

Figure 1. Abstract model relationships

The outcome of *wT* is called *atomic* if all its subtransactions are either successfully completed or compensated. Alternatively, if subtransactions can differ (some completed and some not), then the outcome is called *mixed*.

Subtransactions

Subtransactions can have different types. A subtransaction, s_i, is *lockable* if the resources (or data) that it uses can be locked until the completion of the parent transaction (wT_p *or* wT_c). A subtransaction is *compensatable* if its effect can be semantically undone through a compensating transaction. If a subtransaction is neither *lockable* nor *compensatable* then it is said to be *pivot*. Any *compensatable* subtransaction s_i has a compensation denoted by c_i that undoes, from a semantic point of view, the actions performed by s_i. A subtransaction is *retriable* if it can be re-executed without causing any data inconsistency. A subtransaction is *replaceable* if there is an alternative that can perform the same task.

Roles

The execution of a *wT* involves different participants, each of which plays a certain role. As shown in Figure 2, we identify four different roles of the participants involved in processing *wT*:

- *Executor:* It represents a participant which is responsible for executing and terminating a subtransaction.

- *Coordinator:* It coordinates *wT* and manages failures and compensations. It also collects the results from the participants in order to maintain consistency of data after the execution of *wT*.

- *Initiator:* It represents a participant which starts *wT*. First it requests the coordinator for a transaction context. Then it asks others participants to participate in *wT*.

- *Terminator:* It 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 purpose of defining the above roles is to automatically and uniformly model the roles of participants in different WS Transactions standards. Figure 2 represents the roles and relationships of the proposed model. Figures 4 and 5 further expand the roles of Executor and Coordinator and show their state transitions and message communication.

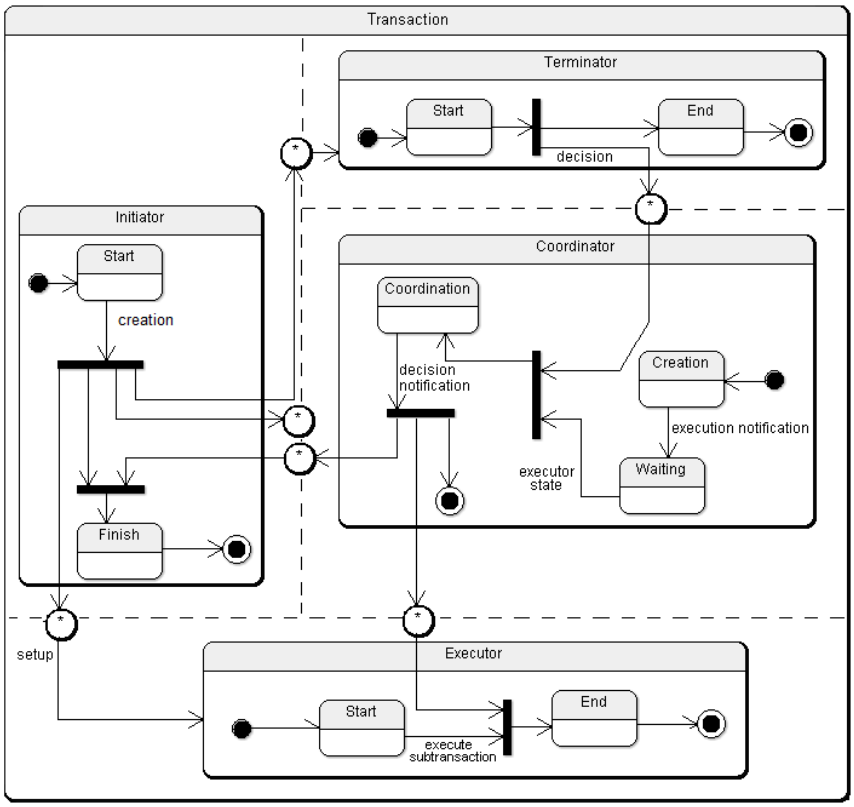

Figure 2. Roles and Relationships in the Abstract Transaction Model

4. MODELING WS TRANSACTIONS STANDARDS

This section shows how different WS Transaction standards can be modelled using the proposed model. As a proof of concept, we present the processes of modelling the BTP and WS-BA standards as these are the most widely accepted standards in WS Transactions. The modelling process is composed by the following activities:

- *Role identification and modelling*: it identifies the roles of participants in a target WS Transaction standard and models it using the roles defined in the abstract transaction model.
- *State transitioning*: it captures the important states of a target WS Transaction standard and maps them to the state transitions of the abstract transaction model.
- *Messages syntax*: it maps the messages between the abstract transaction model and a specific WS Transaction standard.

4.1 Modelling of Business Transaction Protocol

BTP allows coordinating multiple autonomous, cooperating services to ensure that the overall application achieves a consistent result. This consistency can be defined a priori: all the work is confirmed or none; or it can be determined by user's application intervention in the selection of the work to be confirmed. The protocol coordinates the state changes caused by the exchange of messages.

4.1.1 Roles identification and modelling

This activity models the roles of the BTP participants involved in executing *wT* and its subtransactions (as defined in section 3). BTP implements nested transaction model [11], wherein a parent transaction, *wT*, is composed of subtransactions, s_i. BTP defines *Superior:Inferior* relationship between the parent and subtransactions. Fig. 3 depicts the modelling of BTP using the abstract transaction model. Fig. 3 (a) represents the BTP coordination of *wT* and its subtransactions using the *Superior:Inferior* relationship, and (b) represents the coordination of the same *wT* using the abstract transaction model. In BTP the *superior* makes the decision and the *inferior* abides such decision in order to complete the transaction. In BTP the *Superior:Inferior* relationship can be recursively extended to define a transaction tree having intermediates nodes as superior and inferior. The superior (of BTP) is modelled as *Initiator* (in the abstract model). Also the superior can be modelled as *Coordinator* and *Terminator* as it decides on the outcome of the subtransactions. *Inferior* (in BTP) executes a subtransaction and is therefore modelled as *Executor* (in the abstract model).

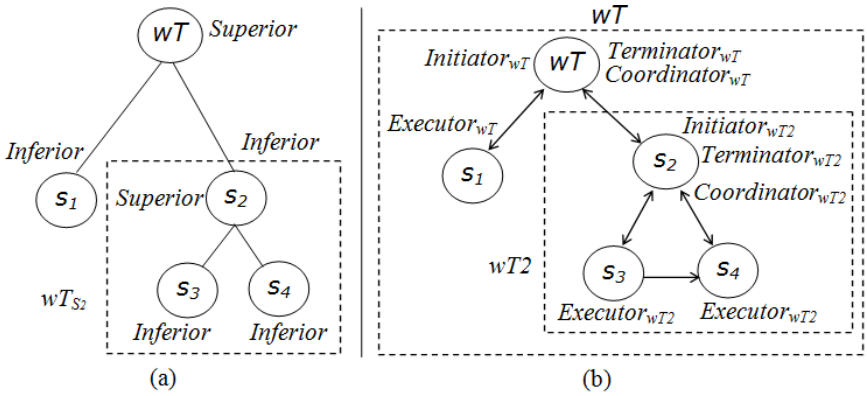

Figure 3. BTP relationships modeling

4.1.2 State transitioning

Figures 4 and 5 show the states and transitions during the processing of *wT*. The abstract model uses these to model the BTP (as well as WS-BA) states and transitions. When a *wT* is started at the initiative of an initiator it causes the creation of a context for a new transaction. It moves from START state to FINISH state through *creation*. The coordinator replies the context and moves from INITIAL state to ACTIVE state. Executor receives a context, enrols with the Coordinator and moves from READY to ACTIVE state. The Executor moves to COMPLETED state after processing its

subtransaction. Coordinator moves to PREPARE state awaiting decisions from Executors. The Executor sends its outcome to Coordinator and moves to DECISION state. The Coordinator collects the outcomes from all Executors and takes the final decision. It moves from PREPARE state to DECISION state. The final decision is sent to each Executor and the Coordinator then moves to CONFIRM state. Executor sends acknowledgement and changes its state to END state through the transition (either completed rollback or completed successfully) according to the decision. Once the Coordinator has received all confirmation, it moves to END state. Note that an Executor can leave the *wT* before confirming the subtransaction. So it can move from ACTIVE state to CANCEL state.

Although BTP uses a 2PC protocol, Executors are not required to lock data on becoming prepared (i.e., in prepared state). This can produce a contradicted decision since the Coordinator could take a decision for all the Executors but some Executors may take their own decisions. When the Coordinator detects a 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 subtransaction, it moves to FAILED state. A new Executor is selected and the previous one moves to END state.

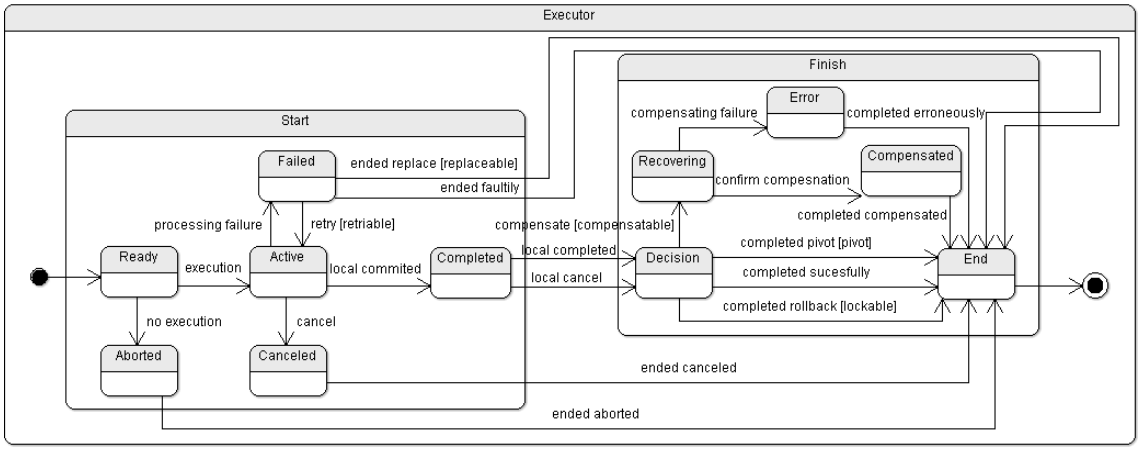

Figure 4. Abstract executor

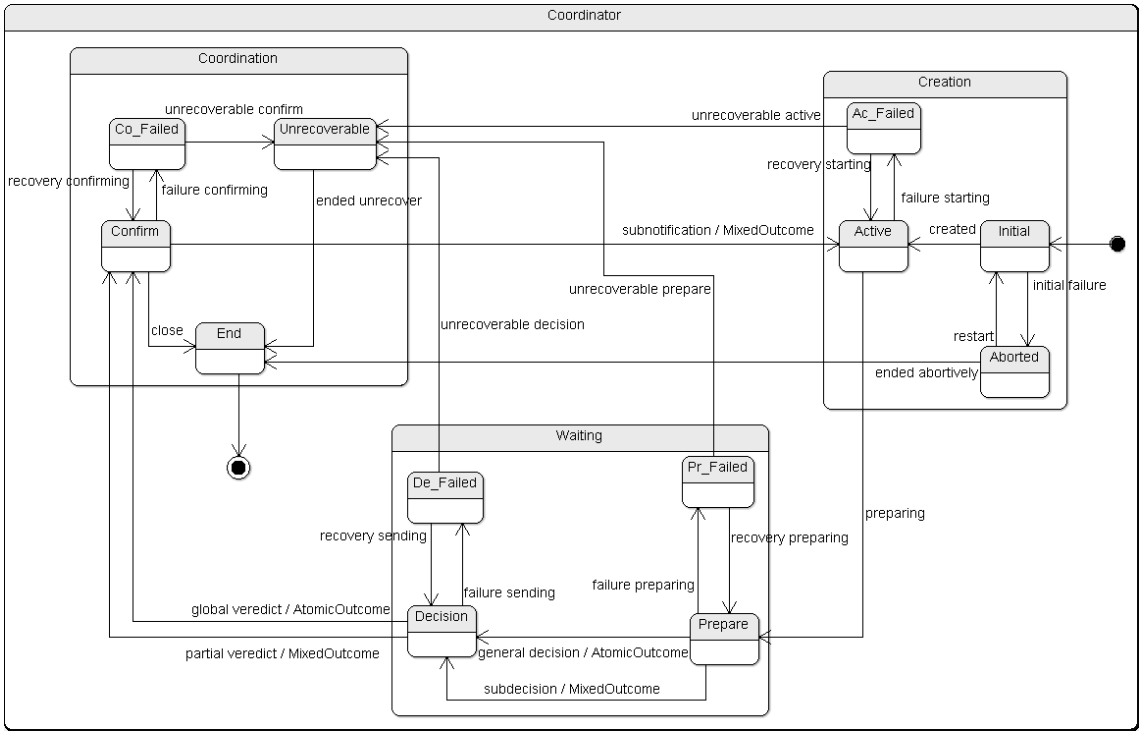

Figure 5. Abstract coordinator

4.1.3 Messages syntax

Table II presents the transformation of messages from the abstract transitions to BTP specific message syntax. The table shows that the abstract model captures all the messages required to complete a transaction using BTP.

Table II. BTP message mapping

Abstract model	**BTP**
Creation	Initiator sends BEGIN to coordinator.
Created	Coordinator sends BEGUN to initiator.
Setup	Initiator sends the context to the executors
Execution	Execuntor sends ENROL to coordinator. It responses with ENROLLED. If the exeuctor is a superior of a new *wT*, it response with CONTEXT_REPLY.
Local commited	Coordinator sends PREPARE to executor. Due a protocol optimization, this transiction could be omitted.
Local completed	Executor sends PREPARED to coordinator
Local cancel	Executur sends CANCEL to coordinator
Completed successfully	Coordinator sends CONFIRM to executor and it responses with CONFIRMED.
Completed rollback	Coordinator sends CANCEL to executor and it responses with CANCELLED.
Preparing	It receives CONFIRM_TRANSACTION from the terminator and sends PREPARE to all executors.
General_decision	Coordinator receives all executor messages

Global veredict	Coordinator sends the suitable message for each executor.
Close	The coordinator receives all the responses from executors and send TRANSACTION_CONFIRMED/ TRANSACTION_CANCELLED to initiator.
Cancel	Executor sends RESIGN to coordinator.
Ended cancelled	Coordinator sends RESIGNED to executor.
Completed rollback	Coordinator wants confirm but there is a contradiction. Coordinator sends CONTRADICTION to executor, and/or executor sends HAZARD to coordinator.
Completed pivot	Coordinator cancels but there is a contradiction. Coordinator sends CONTRADICTION to executor, and/or executor sends HAZARD to coordinator.
Processing failure	The executor is not working. Coordinator knows it receiving a FAIL message or throw a non response message.
Ended replaceablity	Coordinator sends REDIRECT with the address of the new executor.

4.2 Modelling of Web Services Business Activity (WS-BA)

WS-BA manages activities (transactions) that apply compensations to handle exceptions which occur during the execution of activities. WS-BA works with WS-COOR coordination protocol. WS-BA supports two coordination types, *MixedOutcome*, and *AtomicOutcome*, and two protocol types. *MixedOutcome* allows each activity to achieve a specific outcome while *AtomicOutcome* requires all the activities to finish in the same way. The protocols types differ according to the participant's roles in processing subtransactions; Executor (*BusinessAgreementWithParticipantCompletion*, *BAWPC*) or Coordinator (*BusinessAgreementWithCoordinatorCompletion*, *BAWCC*).

4.2.1 Roles identification

Figure 6 depicts the modelling of WS-BA using the abstract transaction model. Figure 6 (a) shows the *AtomicOutcome* protocol, whilst (b) shows *MixedOutcome* protocol. In both protocols 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 is the participant that knows all Executors's output and, therefore, it knows the final outcome: close or terminate if all executors have successfully executed their activities, or compensate otherwise. In *MixedOutcome*, the Initiator is the Terminator since each Executor may have its specific decision so the outcome depends on the business logic.

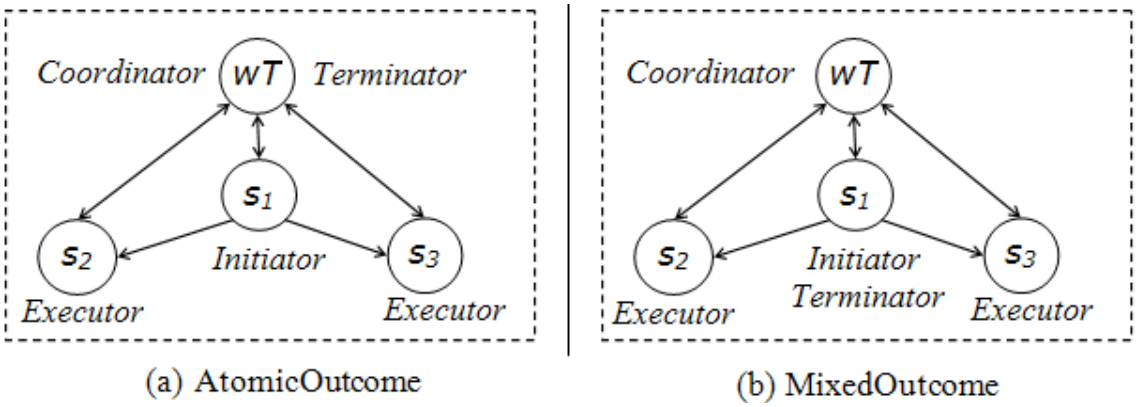

Figure 6. WS-BA relationships modeling

4.2.2 State transitioning

Similar to BTP the abstract transaction model uses the state transitions of Figures 4 and 5 to model the WS-BA. The Initiator requests a context and moves from START to FINISH. The Coordinator responds with a context (from INITIAL to ACTIVE state). That context is sent to the Executors by the Initiator. The Executor joins the current *wT* and moves from READY to ACTIVE state. After making a decision an Executor moves from ACTIVE to COMPLETED state and the Coordinator moves from ACTIVE to PREPARE state. When the transaction is *MixedOutcome*, the decision for each subtransaction is taken alone. The Coordinator moves from PREPARE to DECISION state when it receives an Executor's notification. The Coordinator decides about its outcome and moves from DECISION to CONFIRM. The Coordinator receives the confirmation and goes back to wait for the rest of Executor's notifications (from CONFIRM to ACTIVE state). In the A*tomicOutcome* type, the Coordinator moves from PREPARE to DECISION state when it has a global outcome about the transaction. The Coordinator then sends the global decision and moves from DECISION to CONFRIM state. Finally it waits for the confirmations and moves to END state. When an Executor is not able to start executing its subtransaction it moves from READY to ABORTED state. If the subtransaction was cancelled while it was under execution, the Executor moves from ACTIVE to CANCELLED state. In case of failure it moves from ACTIVE to FAILED state.

4.2.3 Messages syntax

Table III presents the transformation from the abstract transitions to WS-BA specific message syntax.

Table III. WS-BA message mapping

Abstract model	WS-BA
Creation	Initiator sends CREATECOORDINATIONCONTEXT to coordinator.
Created	Coordinator sends CREATECOORDINATIONCONTEXTRESPONSE to initiator. The initiator sends the context for the executors.

Setup	Initator sends the context to the participants
Execution	Each executor, after reciving the context, sends a REGISTER message to its chosen coordinator. The coordinator responses with a REGISTERRESPONSE message.
Local commited	If the coordination type is BAWCC, coordinator sends COMPLETE to executor. In the other coordination type this transition is omitted.
Local completed	Executor sends COMPLETED to the coordinator.
Local cancel	Exectur sends CANNOTCOMPLETE to the coordinator
Completed sucesffully	Coordinator sends CLOSE to executor and it responses with CLOSED.
Compensatable	Coordinator sends COMPENSATE to executor.
Confirm compensation	Executor executes the compensation. There is no WS-BA message for this transition.
Completed compensated	Participant sends COMPENSATED to coordinator.
Preparing	If the coordination type is BAWCC, coordinator sends COMPLETE to executor. In the other coordination type this transition is omitted.
General decision	It is an AtomicOutcome transaction and the coordinator has received either a FAIL message or all Completed messages..
Global veredict	It is an AtomicOutcome and the coordinator sends CLOSE / COMPENSATE message for all completed executors.
Subdecision	The coordinator receives a COMPLETED message.
Partial veredict	The coordinator sends CLOSE / COMPENSATE message to a specific executor.
Subnotification	The coordinator receives the confirmation of a subtransaction.
Close	The coordinator receives all the confirmation messages (CLOSED / COMPENSATED) from the executors.
No execution	Executor sends EXIT to coordinator.
Ended abortively	Coordinator sends EXITED to executor.
Cancel	Participant sends CANCEL to coordinator.
Ended cancelled	Coordinator sends CANCELLED to participant.
Processing failure	Participant sends FAIL to coordinator.
Ended faultily	Coordinator sends FAILED to executor.
Compensating failure	Executor sends FAIL to coordinator.
Completed erroneously	Coordinator sends FAILED message to executor.

5. TEST DESIGN AND EXECUTION PROCESSES

In general, testing aims at showing that the intended and actual behaviours of a system differ, or at gaining confidence that they do not. The main goal of testing is failure detection, i.e., the observable differences between the behaviours of implementation and what is expected on the basis of the specifications of WS Transaction standards. We exploit the 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 [20]. In order to evaluate our approach we have designed a test process which comprises test design, test implementation, test execution and outcome evaluation. These phases are described in the following subsections. A prototype system has been developed in order to automate these steps.

5.1 Definitions

The abstract model can be used to generate test cases for different WS Transactions. The first step is to define a test criterion. Since the model is based on states and transitions, we use the well known criterion of transition coverage [21]. By applying a test criterion over the proposed model, AbTM, we obtain a set of abstract test cases. Each abstract test case is mapped to a concrete test case which is composed by the test scenario and the expected system outcome. The basic concepts used in the test process are defined as follows.

- *Test criterion*: A rule or a collection 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.
- *Abstract test case*: A sequence of states and transitions of a participant using the abstract transaction model. The notation $S_i \xrightarrow{t} S'_i$ 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 $S^a{}_i \xrightarrow{t^x} S^b{}_i - \ldots - S^c{}_i \xrightarrow{t^y} S^d{}_i$ to denote a sequence of states/transitions.
- *Test scenario*: 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 $i\ [m_1]j - l[m_2]o - \ldots - v\ [m_n]z$ to denote a sequence of messages.

5.2 Test design

This phase defines the test requirements for an item and, then, derives the logical (abstract) test cases. At this level 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 reached using this criterion are a set of paths that cover all states and transitions of a model.

5.3 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 as is shown in Section 4. As discussed above the proposed AbTM has the ability to capture the behaviour of a WS Transaction standard as well as mapping the abstract cases to a specific WS Transaction standard. These features provide the capability of automatically obtaining the test scenario and the expected system output.

5.4 Test execution and outcome evaluation

Once the test cases are implemented, they are executed over the system under test (i.e. an application that uses a specific WS Transaction standard) 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) the *user outcome* refers to what the user perceives, for instance, to reserve theatre tickets whether the number of booked tickets is correct. (ii) the *system outcome* refers to the non-visible process that the system has carried out to achieve the requirements - in this case, the correct exchange of messages between the services according to the transaction standard.

Both outcomes are necessary to detect differences from the correct behaviour of the web services application. Let us consider that the application for booking theatre tickets has a fault in creating messages and has an incorrect format of confirmation messages. In a test scenario where the user confirms a reservation, the system outcome would inform the user that a booking was successfully completed because the application sent the confirmation message to the service. Since the message was incorrectly created, the theatre service would reject the reservation and, as a result, the tickets would not 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 transaction management of their activities. Thus we only need to evaluate the system outcome

5.5 Prototype system

We have developed a prototype system that implements the proposed AbTM and the different steps of the test processes (Figure 7). These steps are as follows:

- *Modelling*: the tester models the transaction according to the roles specified by the AbTM (see Section 3).
- *Abstract test case generation*: the abstract test cases for all the participants are automatically generated from the model.
- *Test case mapping*: the specific standard is selected and the tool asks for the necessary information (e.g. the coordinator URL). The tool automatically generates

the concrete test cases composed by the test scenario and the expected system outcome.

- *Test execution*: the tester executes those test cases in the application producing the actual system outcome.
- *Outcomes comparison*: the prototype system compares the actual system outcome to the expected system outcome in order to detect failures.

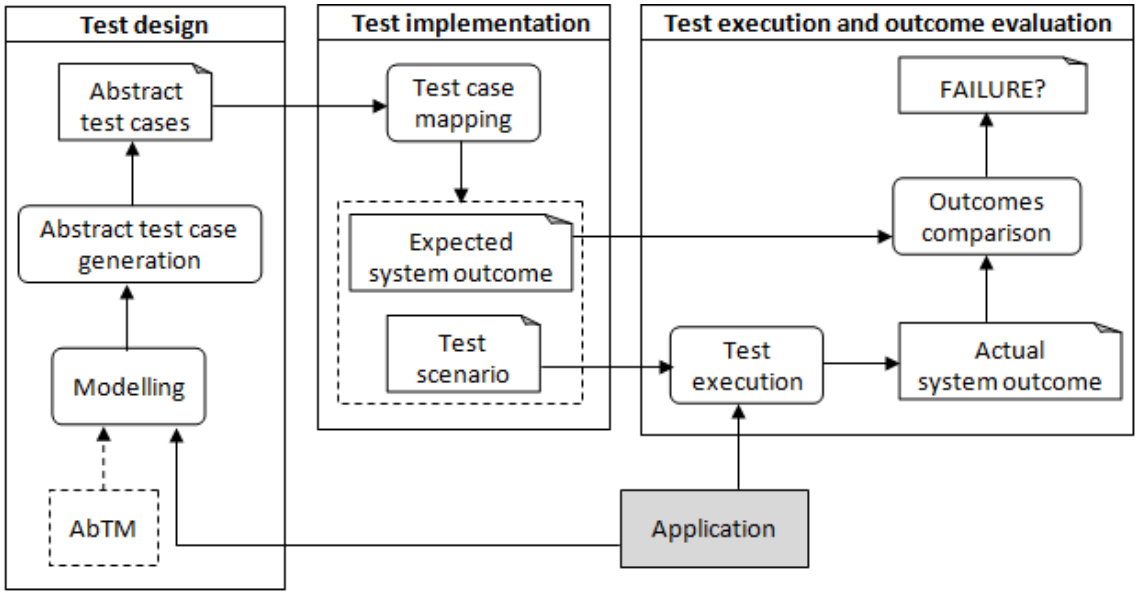

Figure 7. Test process using the AbTM

6. EVALUATION OF THE PROPOSED APPROACH

In order to evaluate the proposed AbTM-based testing approach, we utilise the *Night Out* case study of the Jboss WS-BA standard [8]. The specification of the application is described in subsection 6.1. The test design process for such application is described in subsection 6.2. Subsection 6.3 briefly describes the test implementation process and the result of their execution is discussed in subsection 6.4. Finally, subsection 6.5 explains in detail a test case as an example of test case and detected failure.

6.1 *Night Out* specification

The *Night Out* is an application based around booking independent services for night time leisure. It is composed of three services. *Restaurant* service allows customers to reserve a table for a specified number of dinner guests. *Theatre* service provides automatic reservation of seats in a theatre. There are three kinds of seats (circle, stalls, and balcony) and the service allows customer to book a specified number of tickets for each kind of seat. *Taxi* service provides the facility to reserve a taxi.

Night Out is implemented in a client/server architecture. The client provides an interface to select the nature and quantity of the services reservations. The server components consist of three services (*Restaurant*, *Theatre*, *Taxi*) which are

implemented as transactional web services. The client side of the application is implemented as a servlet which allows users to select the 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 the restaurant or the theatre, the taxi will not be necessary. Each service, exposed as Java API for XML Web Services (JAX-WS) [9] endpoint, has a GUI with state information and an event trace log. The application provides logs for step of its activity. As the transaction proceeds, each of the WS pops up a window of its own in which its state and activity log can be seen. Some events in the service code are also logged.

The client obtains service endpoint proxies from JAX-WS and uses them to invoke the remote service methods. The client begins a transaction that may involve three services: reserve theatre tickets, a restaurant table and a taxi according to the selected parameters. *Night Out* notifies the final outcome of the transactional process, i.e., whether the reservations were confirmed or not.

6.2 Test design

The transactional process included in the *Night Out* application has been modelled according to the roles identified in the abstract transaction model as is shown in Figure 8. *Night Out* (client side) takes the role of Initiator since it starts the transaction and asks the other web services to participate. *Restaurant*, *Theatre* and *Taxi* services are modelled as Executors since they execute a specific activity. The role of Terminator is taken by the *Night Out* application since some activities (e.g. *Theatre*) are independent of others services (e.g. *Restaurant*). Thus even if one service can not complete its action the others are allowed to commit. The *Taxi* activity is dependent. 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, *WSCoor11*, provided by the server. It follows the WS-COO [19] and WS-BA[3] standards to exchange suitable messages.

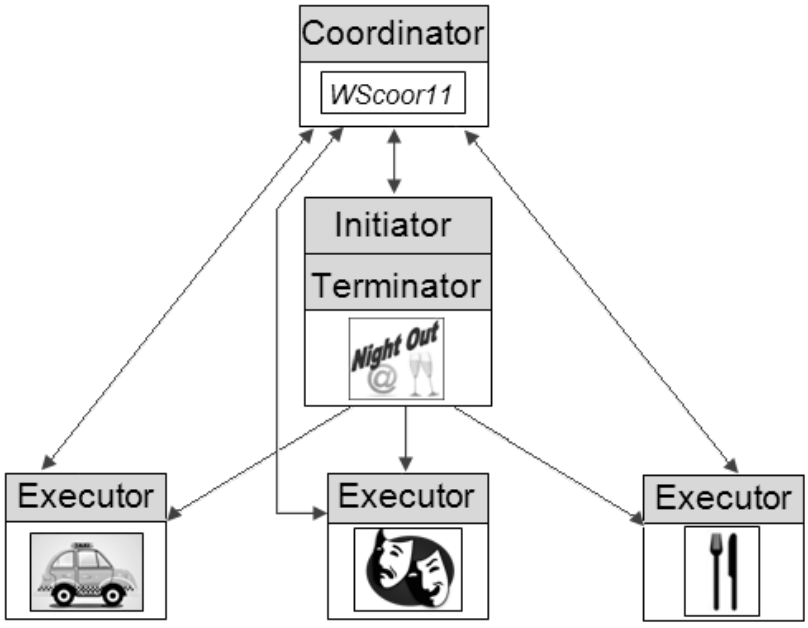

Figure 8. *Night Out* case study modeling

6.3 Test implementation

In this paper we focus on testing the role of Executor: *Restaurant*, *Theatre* and *Taxi*. According to testing approach explained in Section 5, eight abstract test cases are generated for each Executor. Those abstract test cases were automatically mapped to generate test cases, i.e., the test scenario and the expected system outcome for *Restaurant*, *Theatre* and *Taxi* services. As an example we summarize the eight test cases for *Theatre* service in Table IV. The_1 means test case 1 for a *Theatre* service. The_2 means test case 2 and so on.

Table IV. Test cases for the *Theatre* service

ID	Description
The_1	To cancel the theatre booking once it has started but before it has confirmed the reservation
The_2	To force the theatre to be not able to book because there is no available seats
The_3	To undone a tickets booking by executing the compensation
The_4	To confirm successfully a tickets booking when the transaction has to commit
The_5	To confirm successfully a tickets booking when the transaction has to be compensated
The_6	To abort a tickets booking before it has started
The_7	To force a failure during the theatre compensating booking process
The_8	To force a failure during the theatre booking process. To retry the request.

6.4 Test execution and outcome evaluation

The generated test cases have been executed over the case study and Table V summarizes the results. 'Pass' means that a test case did not detect any failure. 'Fails' means that the actual outcome differs from the expected outcome (i.e. a failure has been detected). 'Blocked' means that a test case cannot be executed. In this section we use a number to identify each test case according to the Table IV. For each number there are actually three test cases, one for each executor (e.g. Rest_3, The_3, Tax_3).

Two of the designed test cases were blocked due to the following reasons; test case 1 requires cancelling the activity (`Cancel` message) once the Executor has started and has not finished yet, but the application does not allow cancelling a booking. Test case 8 requires the Executor to retry its activity once it has notified that it was not able to complete the activity before (`CanNotComplete` message), and the application neither

allows resending the data nor registering again the Executor without starting a new transaction.

Table V. Test execution results

Executor	Test cases generated	Pass	Fails	Blocked
Restaurant	8	3	3	2
Theatre	8	3	3	2
Taxi	8	3	3	2

The test case 5 detected an important transaction-related failure in the compensation process. This test case and the detected failure are further explained in subsection 6.5. During the execution of test cases 3 and 4 interface-related failures were detected: the application, which shall allow changing manually the capacity of each 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.

6.5 A test case in detail

As an example of test case and detected failure we consider the test case generated using the following abstract test which was obtained applying the transition coverage criterion over the executor abstract model:

$$READY \xrightarrow{Execution} ACTIVE \xrightarrow{Local\_commited} COMPLETED \xrightarrow{Local\_completed} DECISION \xrightarrow{Completed\_sucesfully} END.$$

The abstract test case was mapped (see section 4) to a specific sequence of WS-BA message as depicted in Figure 9. From this sequence of messages, our prototype system automatically generates the test scenario shown in Figure 10. Note that the transaction creation and participant register processes are defined by the Initiator as was shows in Figure 2 (*creation* and *setup* transitions).

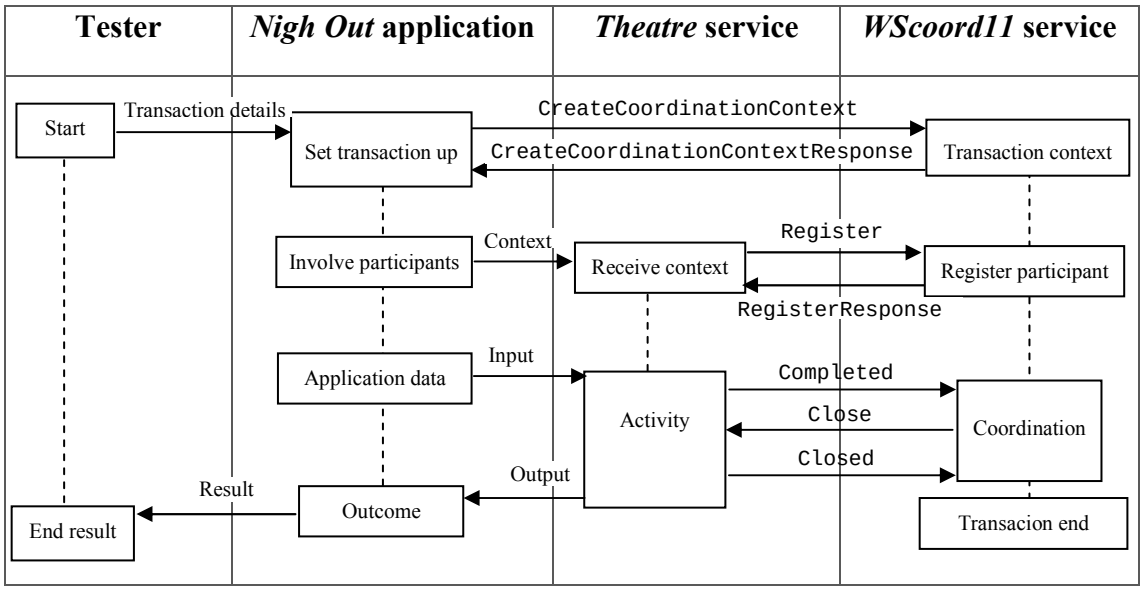

Figure 9. Sequence diagraam of a test scenario for *Theatre* service

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

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

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

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: *NighOut* sends the application data to *Theatre*

STEP 7: *Theatre* completes successfully its activity. *Theatre* sends `Completed` message to notifies 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 notifies to the coordinator *WScorr11* and it responses with a `Closed` message

Figure 10. Test scenario for test case The_5

As described in Table IV, the goal of the test case The_5 is to successfully confirm the theatre tickets booking when the other services reservations have been undone through compensating transactions.

After the execution of the test case, we obtain the expected system outcome. By comparing the expected system outcome and the actual system outcome, a failure is detected by the prototype system (Figure 11). The expected system outcome requires receiving a CLOSE message once the *Theatre* service has successfully completed its activity. 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 also found by the prototype system since there is a difference in the register message: the *Nigh Out* application registers the *Theatre* service as an atomic outcome when a mixed outcome was expected. 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 tickets.

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.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> (a) Expected outcome	<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"> <soap:Header> <Action xmlns="http://www.w3.org/2005/08/addressing"> http://docs.oasis-open.org/ws-tx/wsba/2006/06/**Compensate** </Action>> (b) Actual outcome

Figure 11. Outcomes comparation

7. RELATED WORKS

Current work on WS transactions mainly deals with transaction modelling from a design perspective. A theoretical approach is proposed in [22] 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 [23]. In [24] the authors present a high level UML-based language to design transaction process with diverse transactional semantics. An XML representation is proposed in [25]. Unlike these approaches, we propose a generic model to model the existing approaches to manage WS Transactions.

Though there exists literature on verification and validation of web services, testing of WS transactions has not been addressed so far. In our previous work [26], a risk-based approach is used to define general test scenarios for compensatable transactions. Further in [27] we present test criteria for transactional web services composition. The approach is based on the dependencies which are defined between participants of a transaction. Other work focused on verifying long-lived transactions from a theoretical point of view. In [28], authors have developed a model of communicating hierarchical timed automata in order to describe long-running transactions. This approach allows the verification of properties by model checking. The work presented in [29] translates programs with compensations to tree automata in order to verify compensating transactions. In addition [30] proposes a formal model to verify the requirement of relaxed atomicity with temporal constraints whilst [31] uses event calculus to validate the transactional behaviour of WS compositions. In summary, current research work does not address the issue of testing the reliability and failures of different WS Transactions.

8. CONCLUSIONS

This paper investigated into the issue of testing the WS Transactions. In it we developed and evaluated the Abstract Transaction Model which is capable of dynamically modelling different WS Transaction standards such as BTP and WA-BA. The model exploits model-based testing technique in order to automatically generate test cases for

testing the failures and reliability of WS Transaction standards. The proposed model is implemented as a prototype system with which various test cases were automatically generated and mapped to WS Transaction standards. The evaluation was performed using the case study of *Nigh Out*, which is an open source WS-BA-based application provided by Jboss. The experiments show that our approach can be used to define different test cases as well as test the reliability and failures of different WS Transactions Standards.

ACKNOWLEDGEMENTS

This work has been performed under the research project TIN2010-20057-C03-01, funded by the Spanish Ministry of Science and Technology. This work has also been funded by the research grant BES-2008-004355.

REFERENCES

1. Younas M, Chao K-M, Wang P, and Huang C-L. QoS-aware mobile service transactions in a wireless environment. *Concurrency and Computation: Practice and Experience* 2007; 19 (8): 1219-1236. DOI:10.1002/cpe.1157
2. OASIS. *Business Transaction Protocol*, 2004. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=business-transaction [29 Nov 2011].
3. OASIS. *Web Services Business Activity*, 2009. http://docs.oasis-open.org/ws-tx/wsba/2006/06 [29 Nov 2011].
4. OASIS. *Web Services Composite Application Framework*, 2005. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf [29 Nov 2011].
5. Canfora G, and Penta M. Service-Oriented Architectures Testing: A Survey. *Prooceedings of the Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, 2009. Springer-Verlag: 78-105. DOI:10.1007/978-3-540-95888-8_4
6. Casado R, Tuya J, and Younas M. An Abstract Transaction Model for Testing the Web Services Transactions. *Prooceedings of the IEEE International Conference on Web Services*, 2011. Washington, USA, DOI:10.1109/ICWS.2011.98
7. Casado R, Tuya J, and Younas M. Testing the Reliability of Web Services Transactions in Cooperative Applications. *Prooceedings of the Symposium On Applied Computing (SAC)*, 2012. ACM: Riva del Garda, Trento, Italy,
8. Jboss. Jboss Transactions. 2006. http://www.jboss.org/jbosstm [29 Nov 2011]
9. GlassFish. JAX-WS. 2005. http://jax-ws.java.net/ [29 Nov 2011]
10. Elmagarmid AK.*Database transaction models for advanced applications.* Morgan Kaufmann Publishers: 1992
11. Moss EB. Nested Transactions: An Approach to Reliable Distributed Computing. *Massachusetts Institute of Technology* 1981.
12. Garcia-Molina H, and Salem K. Sagas. *Prooceedings of the SIGMOD 87*, 1987. 249-259. DOI:10.1145/38713.38742
13. Weikum G, and Schek H-J. *Concepts and applications of multilevel transactions and open nested transactions.* Morgan Kaufmann Publishers Inc.: 1992
14. Pu C, Kaiser GE, and Hutchinson NC. Split-Transactions for Open-Ended Activities. *Prooceedings of the 14th International Conference on Very Large Data Bases*, 1988. Morgan Kaufmann Publishers Inc.: 26-37.
15. Reuter. ConTracts: A Means for Extending Control Beyond Transaction Boundaries. *Prooceedings of the 3rd International Workshop on High Performance Transaction Systems*, 1989.

16. Zhang A, Nodine M, Bhargava B, and Bukhres O. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. *ACM SIGMOD Record* 1994. DOI:10.1145/191839.191850
17. Younas M, Eaglestone B, and Holton R. A formal treatment of a SACRED Protocol for Multidatabase Web Transactions. *Database and Expert Systems Applications* 2000; 1873 899-908.
18. OASIS. *Web Services Atomic Transaction*, 2009. http://docs.oasis-open.org/ws-tx/wsat/2006/06 [29 Nov 2011].
19. OASIS. *Web Services Coordination*, 2009. http://docs.oasis-open.org/ws-tx/wscoor/2006/06 [29 Nov 2011].
20. Cavalli A, Cao T-D, Mallouli W, Martins E, Sadovykh A, Salva S, and Zaïdi F. WebMov: A Dedicated Framework for the Modelling and Testing of Web Services Composition. *Prooceedings of the IEEE International Conference on Web Services*, 2010. Florida, USA, DOI:10.1109/ICWS.2010.24
21. Offutt J, Liu S, Abdurazik A, and Ammann P. Generating Test Data From State-based Specifications. *Journal of Software Testing, Verification and Reliability* 2003; 13 (13): 25-53. DOI:10.1002/stvr.264
22. Chrysanthis PK, and Ramamritham K. Synthesis of extended transaction models using ACTA. *ACM Trans. Database Syst.* 1994; 19 (3): 450-491. DOI:10.1145/185827.185843
23. Bhiri S, Godart C, and Perrin O. Transactional patterns for reliable web services compositions. *Prooceedings of the 6th International Conference on Web Engineering*, 2006. ACM: Palo Alto, California, USA, 137-144. DOI:10.1145/1145581.1145613
24. Gioldasis N, and Christodoulakis S. UTML: Unified Transaction Modeling Language. *Prooceedings of the The Third International Conference on Web Information Systems Engineering* 2002. DOI:10.1109/WISE.2002.1181649
25. Hrastnik P, and Winiwarter W. Using advanced transaction meta-models for creating transaction-aware web service environments. *International Journal of Web Information Systems* 2005. DOI:10.1108/17440080580000086
26. Casado R, Tuya J, and Younas M. Testing Long-Lived Web Services Transactions Using a Risk-Based Approach. *Prooceedings of the 10th International Conference on Quality Software*, 2010. IEEE Computer Society: 337-340. DOI:10.1109/QSIC.2010.46
27. Casado R, Tuya J, and Godart C. Dependency-based criteria for testing web services transactional workflows. *Prooceedings of the Next Generation on Web Services Practices*, 2011. IEEE: Salamanca, Spain,
28. Lanotte R, Maggiolo-Schettini A, Milazzo P, and Troina A. Design and verification of long-running transactions in a timed framework. *Science of Computer Programming* 2008; 76-94.
29. Emmi M, and Majumdar R. Verifying Compensating Transactions. *Prooceedings of the International Conference Verification, Model Checking, and Abstract Interpretation*, 2007.
30. Li J, Zhu H, and He J. Specifying and Verifying Web Transactions. *Prooceedings of the International Conference on Formal Techniques for Networked and Distributed Systems*, 2008. DOI:10.1007/978-3-540-68855-6_10
31. Gaaloul W, Rouached M, Godart C, and Hauswirth M. Verifying composite service transactional behavior using event calculus. *Prooceedings of the OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, 2007. Springer-Verlag: Vilamoura, Portugal, 353-370. DOI:10.1007/978-3-540-76848-7_23