Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 10 (2012) 880 – 887

Procedia Computer Science

The International Symposium on Advances in Transaction Processing

A Family of Test Criteria for Web Services Transactions

Rubén Casado[a]*, Javier Tuya[a], Muhammad Younas[b]

[a]Department of Computing, University of Oviedo, Spain
[b]Department of Computing and Communication Technologies, Oxford Brookes University, United Kingdom

Abstract

Web Services (WS) 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 has developed various models and protocols in order to improve the performance and reliability of WS transactions. However, there is little research on testing WS transaction based applications. This paper presents a family of criteria for testing WS transactions. The proposed criteria are defined by taking into account three testing dimensions: level, feature and depth. Based on such dimensions we develop a generalized transaction model for testing web service transactions.

Keywords: testing, transactions, web services, control flow, data flow, dependency

1. Introduction

The fundamental principle of Web Services (WS) transactions is to ensure reliable execution of web services and consistency of underlying data which is generally accessed concurrently. Though various solutions have been proposed in order to improve the reliability and efficiency of WS transactions [1-3], the testing of WS transactions has been overlooked [4, 5].

In this paper we investigate the testing of WS transactions which is a challenging research issue due to several reasons. Firstly, WS transactions are more complex compared to classical transactions as they involve cooperation between multiple parties, span autonomous and independent partners, define dependencies among its activities, and may have long duration. Thus WS transactions have a more intricate sequence of operations and execution environment. Secondly, various kinds of failures may occur during the processing of WS transactions, including: (i) technical failures such as communication, system and software failures which may 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.

* Corresponding author. Tel.: +34 985 182 277; fax: +34 985 181 986.
E-mail address: rcasado@uniovi.es.

1877-0509 © 2012 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2012.06.116

Our approach is to devise a family of criteria for testing WS transactions. The criteria identify three dimensions: level, feature, and depth. The *Level* defines the granularity level of testing, i.e., testing WS transactions at different levels such as activity, nested transaction or at the whole process level. The *Feature* defines the basis for deriving test conditions, i.e., the flow of execution. The *Depth* defines the different combinations of test coverage items defined by the suitable test criteria.

The rest of the paper is structured as follows. Section 2 describes a generalized transaction model for the web service environment. Section 3 presents the proposed testing criteria. Conclusions are drawn in Section 4.

2. Web services transaction model

This section presents a generalized transaction model for the web service environment. It is defined to capture the behavior of a WT transaction from a testing point of view.

A **Web Service transaction**, *wT*, is a logical unit of work performed by a flow of activities whose goal is to achieve an agreed outcome in a WS based application. It is defined as `wT={A,D}` where `A` is a set of activities and `D` a set of dependencies among them. **Activities** represent points in a *wT* where work is performed. An activity can be atomic (task) or non-atomic (subtransaction). Each activity is executed by an executor. An activity is compensatable if a compensation exists within the *wT* to undo its actions. A **task** is an atomic activity within a *wT*. A task is used when the work in the *wT* is not broken down to a finer level of detail. A task is executed by a specific web service. A **subtransaction** is an activity that is another WS transaction itself. Thus it gives rise to nested transactions. A **compensation** is an activity that undoes from a semantic point of view the actions performed by another activity. An **executor** is a web service responsible for executing a specific activity. The states during the transaction processing are described in Section 2.1 **Dependencies** are constraints on the processing produced by the concurrent execution of activities. A dependency defines a relationship between a set of activities. The dependencies used in this approach are described in Section 2.2

2.1. States of an executor

An executor is a web service in charge of executing an activity. An executor can be in any of the following commonly used states: *Initial*, *Active*, *Completed*, *Compensated*, *Aborted, Cancelled* and *Failed*. The state of an executor is changed by the execution of a primitive action. There are six atomic **primitive actions**: *begin*, *complete*, *compensate*, *A-withdraw*, *A-cancel* and *A-fail*. Note that the primitive action compensate is only applicable if the activity is compensatable. The states of the executor and state transitions are shown in Fig. 1. Solid lines represent external primitive actions while the dashed lines represent internal primitive actions.

An executor is in the **Initial** state when it has been enrolled in the *wT* and it is waiting to be executed. An executor is in the **Active** state when it has executed the *begin* primitive action and it has not finished execution. An executor is in the **Completed** state after it has successfully finished its activity. From the completed state, the executor can enter the Compensated state if the activity is compensatable. An executor is in the **Aborted** state after it has executed one of the abort primitive actions. An executor is in the **Cancelled** state after it was cancelled while executing its activity. An executor is in the **Failed** state if it was not able to successfully finish its activity. An executor which has executed a compensatable activity is in the **Compensated** state after it has executed the *compensate* primitive action, that is, its actions have been undone by executing a compensation.

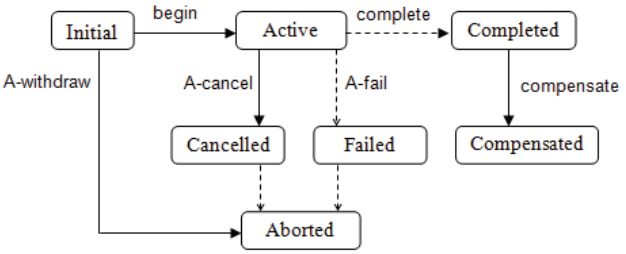

Fig. 1. States and transitions of an executor

2.2. Dependencies

The proposed model defines three kinds of dependencies in WS transactions: **Flow dependencies** define constraints on the workflow in terms of the order of execution of activities. **Data dependencies** define relations between the data used by the activities. These specify relations according to read and write operations on shared data. **Control dependencies** – these are hybrid dependencies (a mix of flow and data dependencies). Different types of dependencies can be combined. Let us assume an example of purchase process; the payment activity must be executed after the items have been selected (flow dependency) but the amount to be charged depends on the calculation process that takes into account the price and quantity of the selected items (data dependency). Finally the payment is carried out if the number of items is at least one (control dependency). Table 1 summarizes the dependencies proposed in our approach.

A **data element** is a piece of information accessed by *wT*. An activity is said to **write** a data element if it generates or changes the value of such data element during its execution. An activity is said to **read** a data element if it reads such data element during its execution. We represent a data dependency as `write(A, d1, d2)` where activity `A` reads the data element d_1 and updates (writes) it to the data element d_2. In other words, `A` requires d_1 to produce d_2.

Table 1. Description of WS transaction dependencies

Name	Description	Example
Serial	One activity can start after another has successfully completed	The item is sent to the customer once the payment has been confirmed
Alternative	Only one activity can begin	In a purchasing process, the customer selects one method (credit card, bank transfer, *Paypal*) to pay for the item.
Fork	All the activities begin	In a journal review process, the editor sends the email to all the reviewers.
Merge	At least one activity must complete before another can begin. Extra conditions can be specified.	At least one means of transport (car, train, plane) has to be available before continuing the package holiday reservation.
Join	All activities must complete before another can begin	All the bookings (flight, hotel, car rental) have to completed before paying for the package holiday
Exclusion	Only one activity can complete	When different hotel providers are consulted, only the cheapest one has to complete
Write	One activity produces a data element and it may require another data element	The tax to be paid depends on the number of items sold in a day

3. Testing WS transactions

This section summarizes the fundamental concepts involved in the test case design process. Fig. 2 depicts the relations between such concepts.

3.1. Background

The aim of testing is to systematically explore the unexpected behaviour of a system or a component. Ideally, all the possible situations of the Software Under Test (SUT) should be tested. But this is not feasible since even if the SUT has an extremely simple logical structure, the number of all possible combinations of situations could be infinite. Furthermore, the test process consumes resources such as time and money. For these reasons, test techniques are used in order to ensure testing is carried out taking into account the effectiveness/cost trade-off. Test techniques provide guidance to design test cases using some information about the SUT, for example, the workflow specification or a model. They allow to systematically identify the most relevant conditions to test and the most important values for each condition. Before explaining the criteria applicable to test the WS transactions, we introduce some definitions. Fig. 2 shows the relation between these concepts.

Test basis are all sources from which the requirements of a component or system can be inferred. They are broken down into **test items** that are the minimal functional unit that can be tested in isolation. For each test item a set of test conditions is derived. A **Test condition** is an item or event of a component or system that could be verified by one or more test cases, e.g. a function, transaction, feature, quality attribute, or structural element. For each test condition several test coverage items can be specified. A **Test coverage item** is an entity or property with a concrete value derived from a test condition; e.g. a logical value in a decision or a concrete state of a statechart. The test coverage items must be covered by the test cases. A **Test case** is a set of input values, execution preconditions, expected results and execution postconditions, developed to cover a set of test coverage items. The set of test cases is called a **Test suite**.

3.2. Level dimension: Test Items

Executor. The activities that compose a WS transaction are carried out by executors. In fact an executor is a role entrusted to a web service. When a web service is enrolled in a WS transaction, it must follow the protocol specified for such process in order to be able to achieve an agreed outcome of the whole transaction. So a first level of testing should specify the executor as the test item. We assume that the web service has already been tested, so we focus on the transaction-related behaviour. The test cases for this level have to exercise the different situations that an executor has to manage during its life-cycle. In [6] we presented an approach to test the transaction executor level. That work proposes an abstract model for modeling distinct web service transaction standards and testing their reliability in terms of failures. The proposed approach exploits model-based testing techniques in order to identify the test conditions and derive the test coverage items.

Transaction. A WS transaction may be seen as a flow of related activities and such relations are specified by dependencies between them. The constraints defined by the dependencies must be tested so they should be the test item at this *transaction* level. Test cases at this level must exercise different possible situations during the flow of execution in order to detect faults in the compliance of the specified dependencies. In [7] we proposed a first approach to deal with this issue. A set of possible dependencies is defined using logical expressions and using such expressions as test conditions, a family of test criteria based on control-flow testing is proposed.

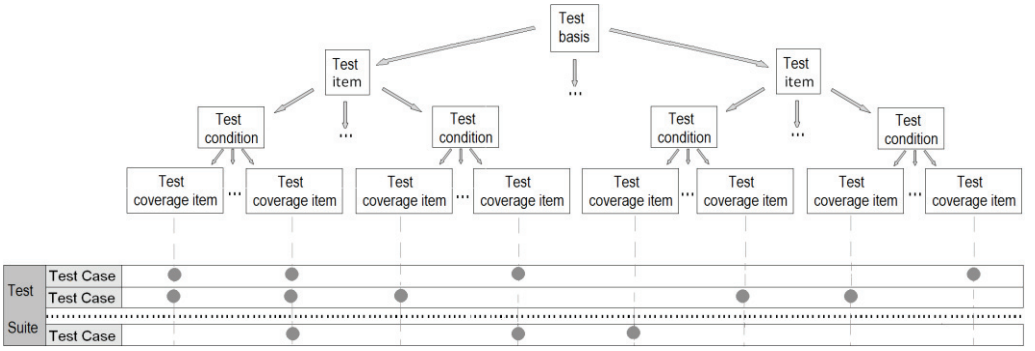

Fig. 2. Test case design concepts

Recursive levels. As was defined in Section 2, a WS transaction is composed by activities where each activity can be an atomic task or another WS transaction itself (subtransaction). Even a business process can include several different WS transactions. So the executor and transaction levels can be applied recursively. In order to depict the recursive relations, Fig. 3 shows a business process *P* composed by two WS transactions *wT1* and *wT2*. *wT1* is composed by the tasks *A* and *B* while *wT2* is composed by task *C* and subtransaction *DwT*, also composed by the tasks *E* and *F*. As an example, in *wT2* the executor level can be applied over C and also over *DwT* if we assume it to be a logical unit of work. The transaction level in *wT2* will take into account the relationships between *C* and *DwT*. Since *DwT* is a transaction itself, recursively we can use the executor level to test *E* and *F* and the transaction level to check their relationships.

3.3. Feature dimension: Test conditions

Flow

An executor crosses through different states during the execution of an activity. The dependencies in a WS transaction define the order of execution of the activities. So, for both *executor* and *transaction* levels, a control flow graph can be derived. In the *executor* level the graph is the states/transitions model depicted in Fig. 1. In the *transaction* level, the graph is composed by the different paths in the execution of the transaction according to the dependencies.

Using the flow as basis for the test case design is widely accepted and is included in the so-called transition-based testing [8]. Its goal is to specify the test conditions in terms of the coverage of a particular set of elements in the structure of the element under test. The test conditions are, therefore, the elements of the control flow graph. In testing WS transaction, it would be applied for example as shown in Table 2.

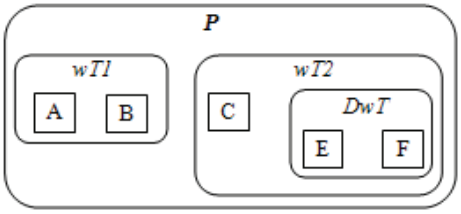

Fig. 3. Recursive test levels

Table 2. Testing flow dimension examples

Level	Test criterion	Test coverage item

Executor	All transitions in the executor model must be visited	`Begin – Complete – Compensated`
		`Begin – A-cancel`
		`Begin – A-fail`
		`A-withdraw`

Transaction	All different paths must be visited	Activities `A`, `B` and `C` complete their actions, `D` fails Activities `A`, `B` and complete their actions, Activity `A` is compensated, `B` is aborted

Data

An executor may use some data elements during its execution. Depending on the executor´s behaviour, such data can be modified by one way (e.g. after it has completed) or another (e.g. after it is compensated). Also different activities from a WS transaction can use the same data elements. So the data elements are a key issue regarding the transaction outcome and should be taken into account during the test process.

The test techniques that use the data elements as a basis for designing the test cases are classified in the approach known as data flow testing [8]. These approaches look at the life-cycle of a particular data element in the element under test. By looking for patterns of data usage, risky situations are identified and more test conditions can be defined. For example an activity `A` requires the value of a particular data element d_1, produced by the activity `B`, to produce another data element d_2. A risky situation could be that d_1 is modified because `B` is compensated while `A` is still under execution. Consequently the test conditions for this feature are the data usage during the process, i.e. creation, read and write actions over the data elements.

Control

The decision of an executor moving from one state to another may depend on the value of a data element. This is called a control decision. In the same way, there are control decisions during the flow of execution specified by the dependencies. For example in an `exclusion` dependency, the control decision decides which is the selected activity to start.

The goal of testing the control feature is to exercise different values of the data elements that are involved in the control decisions. The test techniques that use the control decision to define the test conditions are called control-flow testing [8]. They can be complemented with other techniques such as boundary analysis [8]. An example of the use of control as a feature in testing WS transaction is shown in Table 3.

Table 3. Testing control dimension examples

Level	Test conditions	Test coverage item
Executor	If time > 5 move to *Compensated* state	`time >5` `time < =5`

Transaction	The cheapest alternative will begin	`A_price= 10 ; B_price= 20`
		`A_price= 40 ; B_price= 30`
		`A_price= 5 ; B_price= 5`

3.4. Depth dimension: A combination of test coverage items

Test criteria are used to define the test conditions and identify the test coverage items. Then the set of test cases must cover all the test coverage items, but this can be achieved in different ways, depending on the required test effort. So different adequacy criteria are applicable to exercise the test coverage items. In this way, stronger test criteria should be applied in the areas with greater risk exposure in order to achieve an effective testing. The minimum effort would be simply to cover all the test coverage items. On the other hand, the maximum effort would be to generate all possible combinations between the test coverage items and define a test case to cover each combination. Thus the test criteria propose different test efforts from the minimum to the maximum effort. Applicable test criteria for WS transactions is presented below [8].

Moreover, test conditions derived from different features can be combined. Therefore the test effort is suitable in order to achieve effective testing and avoid an unmanageable number of test coverage item combinations to cover.

Test criteria

In the *flow* feature, the test basis is typically a flow graph. The test coverage items are defined in terms of transitions between the nodes of such graph. Two test criteria are commonly used to specify the effort to cover the transitions.**Transition coverage** (TrC) requires that each transition in the flow graph is taken at least once while **Transition-pair coverage** (TPC), for each state S, form test conditions such that for each incoming transition and each outgoing transition, both transitions must be taken sequentially. So TPC is stronger than TrC since TPC subsumes TrC.

In the *data* feature, the test conditions are derived from the definition and use (write and read) of the data elements. Some applicable test criteria are the following. **All definitions paths** (ADP) requires that all nodes where a data element is written shall be taken at least once while **All uses paths** (AUP) requires the nodes where a data element is written. **All definitions – uses paths** (ADUP) requires that all different flows where a data element is written and then read shall be taken at least once. ADUP is stronger than ADP and AUP since it subsumes them.

In the *control* feature, the test conditions are derived from the decisions that manage the flows. The decisions are composed by logical conditions where the value of the data elements influences the flow of execution. Some applicable test criteria are the following. **Decision criterion** (DC) requires that each decision shall take true and false outcome at least once while **Decision/Condition criterion** (DCC) also requires that all conditions in each decision shall take true and false outcome at least once. The strongest criterion is **Multiple Combination** (MC) that requires covering all possible combinations of the truth in every condition. **Modified condition/decision coverage** (MCDC) requires that each decision shall take true and false outcome at least once, all conditions in each decision shall take true and false outcome at least once and each condition shall be shown to independently affect the decision´s outcome.

4. Conclusions

Although transactions are a key issue in web service compositions, there are few works about testing them. This work aims to approach the test case design concepts to the WS transaction field. In this paper we have defined three dimensions of testing WS transactions (*level*, *feature*, *depth*) according to some basic test concepts (*test unit*, *test conditions*, *test coverage items*). Furthermore, we have presented some ideas of how a family of well-known test criteria such TPC, ADUP or MC, could be used to create a specific framework to test WS transactions. More research is needed to evaluate such proposals.

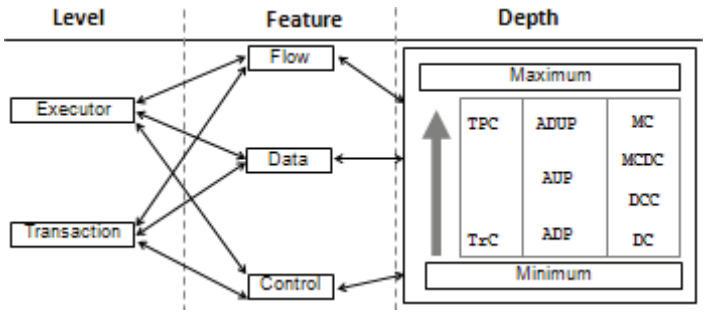

Fig. 4. WS transaction testing dimensions

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 also has been funded by the research grant BES-2008-004355. It has also been carried out in collaboration with the Oxford Brookes University, UK.

References

[1] Bhiri S, Gaaloul W, Godart C, Perrin O, Zaremba M, and Derguech W. Ensuring customised transactional reliability of composite services. *Journal of Database Management* 2011; 22 (2): 29.

[2] Ben Lakhal N, Kobayashi T, and Yokota H. FENECIA: failure endurable nested-transaction based execution of composite Web services with incorporated state analysis. *The VLDB Journal* 2008; 18 (1): 1-56.

[3] Choi S, Kim H, Jang H, Kim J, Kim SM, Song J, and Lee Y-J. A framework for ensuring consistency of Web Services Transactions. *Information and Software Technology* 2008; 50 (7-8): 684-696.

[4] Canfora G, and Penta M, "Service-Oriented Architectures Testing: A Survey", *Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, pp. 78-105: Springer-Verlag, 2009.

[5] Bozkurt M, Harman M, and Hassoun Y. Testing & Verification In Service-Oriented Architecture: A Survey. *Software Testing, Verification and Reliability*. To appear.

[6] Casado R, Tuya J, and Younas M. Testing the Reliability of Web Services Transactions in Cooperative Applications. *Proc. of the 27th ACM Symposium on Applied Computing (SAC)*, 2012. ACM: Riva del Garda, Trento, Italy,

[7] Casado R, Tuya J, and Godart C. Dependency-based criteria for testing web services transactional workflows. *Proc. of the 7th International Conference on Next Generation Web Services Practices (NWeSP)*, 2011. Salamanca, Spain, 74-79.

[8] Zhu H, Hall PAV, and May JHR. Software unit test coverage and adequacy. *ACM Comput. Surv.* 1997; 29 (4): 366-427.