RADAR

Research Archive and Digital Asset Repository

OXFORD BROOKES UNIVERSITY

Casado, R, Younas, M and Tuya, J (2013) Multi-dimensional criteria for testing web services transactions.

Casado, R, Younas, M and Tuya, J (2013) Multi-dimensional criteria for testing web services transactions. *Journal of Computer and System Sciences*, 79 (7). pp. 1057-1076.

Doi: 10.1016/j.jcss.2013.01.020

This version is available: https://radar.brookes.ac.uk/radar/items/bafa8853-f6bc-59c9-9332-e44489a813e3/1/

Available on RADAR: July 2013

Copyright © and Moral Rights are retained by the author(s) and/ or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This item cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder(s). The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

This document is the postprint version of the journal article. Some differences between the published version and this version may remain and you are advised to consult the published version if you wish to cite from it.

WWW.BROOKES.AC.UK/GO/RADAR

Multi-dimensional Criteria for Testing Web Services Transactions

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

[a] *Departamento de Informática*
University of Oviedo, Spain

[b] *Department of Computing and Communication Technologies*
Oxford Brookes University, Oxford, United Kingdom

Abstract

Web services (WS) transactions are important in order to reliably compose distributed and autonomous services into composite web services and to ensure that their execution is consistent and correct. But such transactions are generally complex and they require longer processing time, and manipulate critical data. Thus various techniques have been developed in order to perform quality assessment of WS transactions in terms of response time efficiency, failure recovery and throughput. This paper focuses on the testing aspect of WS transactions — a key issue that has not been examined in the literature. Accordingly it proposes multi-dimensional criteria for testing the WS transactions. The proposed criteria have the potential to capture the behaviour of WS transactions and to analyze and classify the possible (failure) situations that effect the execution of such transactions. These criteria are used to generate various test cases and to provide (WS transactions) tester with flexibility of adjusting the method in terms of test efforts and effectiveness. The proposed criteria have been designed, implemented and evaluated through a case study and a number of experiments have been performed. The evaluation shows that these criteria have the capability to effectively generate test cases for testing WS transactions as well as enable tester to decide on the trade-off between test efforts and the quality.

Keywords: web services, transactions, software testing, test case generation, mutation, classification-tree

1. Introduction

Web services (WS) are self-described software applications which can be advertised, discovered, and used over the web [1]. WS expose application functionality using high level programmatic interfaces such as WSDL. The composition of web services (i.e., composite web services) enables inter and intra organisational collaboration and realises the true vision of modern Business-to-Business (B2B) practice and processes [2]. Transactions are one of the most fundamental concepts in delivering reliable application processing and ensuring data integrity in web services. However, transactions in web services are different from traditional transactions which support ACID (Atomicity, Consistency, Isolation and Durability) properties. Characteristics such as long-running, business requirements and network latencies however make it difficult for WS transactions to abide by the strict rules of ACID properties.

WS transactions are defined as sequences of activities that are executed under certain constraints in order to maintain application reliability, correctness and data consistency. The management of transactional activities complicates the business logic of web services as their execution requires careful coordination, accounting for fault-tolerance, correct process termination and cancelation, without undesirable consequences at any stage of the execution. Numerous models and protocols have been developed for WS transactions which include Business Transaction Protocol (BTP) [3], Web Services Business Activity (WS-BA) [4] and Web Services Transaction Management (WS-TXM) [5], among others [6].

Similarly, current research proposes various solutions for testing the (non-transactional) web services [7, 8]. But testing of WS transactions has not been researched yet. The process of testing the WS transactions is harder than testing non-transactional web services due to several reasons. First, WS transactions involve hierarchies of interactive activities between distributed systems that should execute in atomic manner. WS transactions are more complex compared to classical database transactions as they involve cooperation among multiple parties, span autonomous and independent partners, and have long duration. Second, WS transactions do not have a homogeneous transaction model such as the ACID model. Instead they are designed and developed using multiple transaction models and protocols and different Web services technologies. Such diversity of models and technologies also complicates the process of testing the 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. Such failures result in loss of

Corresponding author: Rubén Casado, University of Oviedo, Spain
Email: rcasado@uniovi.es

messages, processing of services, and inconsistency of data. (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. Fourth, individual behaviours (e.g., states and transitions) and relationships (e.g., execution order and flow) of services taking part in WS transactions cannot be easily determined. This is due to the fact that such services are heterogeneous and autonomous and thus they have different behaviours and relationships among their activities [6, 9]. In summary, testing of WS transactions involves different challenges due to the complex nature of the environment, distinct behaviours of individual services, the relationships between services, the sharing of data and the consequential risk of data inconsistency.

To address the above challenges, we present multi-dimensional criteria for testing the WS transactions [10]. The criteria define three dimensions; Level, Feature and Depth. The *Level* dimension defines the granularity level of testing, i.e., testing WS transactions at different levels such as activity (web service) level [6], nested subtransaction or overall transaction level. The *Feature* defines the source used to identify the situations to be tested (test conditions). For example, the relationships between activities, the execution flow of activities or the data elements shared by the activities of a WS-transaction. The *Depth* is related to test coverage items, designing of the test cases, and the cost-benefit trade-off of testing the WS transactions. Note that in terms of standard software testing terminology, the proposed criteria belong to the '*integration testing*' category.

Contributions of this paper are as follows:

(i) A method for designing test cases for WS transactions. We present systematic multi-dimensional criteria to identify test conditions and test coverage items by analysing the existing dependencies between the involved activities.

(ii) A family of test criteria. Different criteria have been proposed to combine the test coverage items and to adjust the test efforts.

(iii) We have evaluated the proposed criteria using a case study which has been widely discussed in the literature [11-15]. The test suites generated using the different test criteria were executed in mutated versions of the program. The aim is to measure the quality of test cases and to prove the effectiveness of the criteria.

The rest of the paper is structured as follows. Section 2 reviews related works on WS transactions and verification and validation of transactional processes. Section 3 defines the scope of testing the WS transactions. Section 4 analyses the dependencies between activities from a testing point of view. Section 5 describes how the proposed criteria use such analysis to design the test cases. Section 6 illustrates the evaluation of the criteria and discusses the experimental results. Section 7 concludes the paper.

2. Related Work

Existing literature contains various strategies and solutions to test classical transactions, in the areas of databases [16, 17] or system-on-chips (SoCs) at electronic system level [18, 19]. Similarly, work on performance testing and evaluation of transactions has been received significant attention from existing work [20-22]. In addition, work on verification and validation of classical as well as WS transactions has been carried out in the literature. The authors in [23] developed a model for communicating hierarchical timed automata in order to describe long-running transactions. The aim is to allow the verification of properties using model checking. The work presented in [24] translates programs from compensations to tree automata in order to verify compensating transactions. Channel-based coordination language, Reo, has been used in order to model long-lived transactions and to verify their properties using model checking technology [25]. Further, authors in [26] use event calculus to validate the transactional behaviour of WS compositions. The work in [27] proposes a formal model to verify requirements for relaxed atomicity and another approach in [9] uses Accepted Termination States (ATS) for ensuring the relaxed atomicity of transactions.

The aforementioned literature presents an interesting work on addressing various issues. But it lacks research on testing the WS transactions. Various and models and protocols have been developed for WS transactions including the classical ACID models, advanced or extended transaction models. Two Phase Commit (2PC) protocol and its variants [28] 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 [29], SAGA model [30], open-nested [31], Split-join [32], Contracts [33], Flex [34], and WebTram [35]. 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 previous transaction models several standard specifications have been developed for WS transactions. For instance, BTP [3] adapts 2PC for short lived transactions and nested transaction model for long-lived transactions. WS-CAF [5] 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) [36] and WS-BA [4] are built on top of Web Services Coordination (WS-COOR) [37] and they follow its coordination mechanism. WS-AT follows 2PC protocol while WS-BA uses the SAGA model.

In addition, various research endeavours have been made in testing (non-transactional) web services. For instance, the authors in [7, 8] survey various approaches related to testing web services. Similarly, the work in [38] reviews formal approaches of web services testing. Testing of web services composition has been investigated in [39, 40]. The work presented in [41] uses the Category Partition methodology in order to define test cases for web services. However, this work focuses on non-transactional web services. In our previous works we devised a general framework for testing web services transactions using risk-based testing approach and taking into account individual participants such as executor [6, 12, 42]. In this paper we focus on the overall testing of WS Transaction at different dimensions. We also use dependencies as test basis for WS transactions [43, 44] for defining different test criteria.

3. Testing of WS transactions: Multi-dimensional criteria

Software testing has been used to systematically explore the behaviour of a software system or a component in order to detect unexpected behaviours. Software testing techniques provide guidance to design test cases using some information about the Software Under Test (SUT), for example, the workflow specification or a model of the WS transactions (as in our case). They allow for systematically identifying the most relevant conditions to test the most important values for each condition. Ideally, all the possible situations of the SUT should be tested. But this is not feasible since even if the SUT has a simple logical structure, the number of all possible combinations of situations can be infinite. Furthermore, the test process consumes resources such as time, cost and other resources. For these reasons, testing techniques are used in order to ensure that testing is carried within the constraints of available resources.

Before explaining the proposed multi-dimensional criteria, first we introduce some basic concepts and definitions. These are diagrammatically represented in Figure 1.

Test basis: These represent all sources from which the requirements of a component or a system can be inferred. Examples of test basis are the functional requirements document or the activity diagrams of a system.

Test items: Test basis are broken down into *test items* which represent the minimal functional unit that can be tested in isolation. Test items are, therefore, smaller units of a system such as a specific process or a piece of code.

Test condition: For each test item a set of test conditions is derived. A *test condition* is an item or event of a component or a system that could be verified by one or more test cases; e.g. a function, transaction, feature, quality attribute, or structural element.

Test coverage item: For each test condition several test coverage items can be specified. A *test coverage item* is an entity or a 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.

Test case: The test coverage items must be covered by the test cases. In fact, a test case exercises a combination of test coverage items. A *test case* is defined as set of input values, execution pre-conditions, expected results and execution post-conditions, that cover a set of test coverage items.

Test suite: The set of test cases is called a *test suite*.

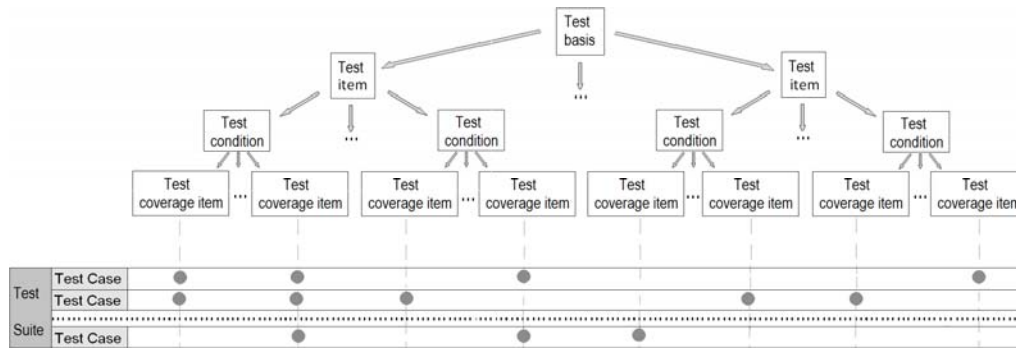


Figure. 1. Test case design concepts

In the following we explain the proposed multi-dimensional criteria for testing the WS transactions.

3.1. Level dimension: Test Items

The level dimension refers to the granularity level of testing. It defines the test items which include:

Executor or Sub-Transaction Level: This represents testing at the executor or sub-transaction level. In the proposed model, each executor represents a participant which is responsible for executing and terminating a sub-transaction of a WS transaction [45]. In other words, the sub-transactions or activities that compose a WS transaction are carried out by executors. When a web service is enrolled in a WS transaction, it must execute in a way that complies with an agreed outcome of the overall WS transaction. In the proposed criteria, the first level of testing considers each executor as a test item. The test cases for this level have to exercise different situations that an executor has to manage during its life-cycle.

WS Transaction Level: This represents testing at the overall WS transaction level. A WS transaction is represented as a flow of related activities or sub-transactions. Such relations are specified by dependencies between the activities of a transaction. The constraints defined by the dependencies must be tested. Thus at this level, dependencies are considered as test items. Test cases at this level must exercise different possible situations during the execution flow of the activities in order to detect possible faults that may occur in enforcing dependencies between activities.

Recursive or Hybrid Level: A WS transaction is composed of different activities where each activity can be an atomic task or another subtransaction of a WS transaction. Thus the above executor and transaction levels can be applied recursively.

3.2. Feature dimension: Test conditions

Flow: An executor passes through different states during the execution of an activity [6]. The dependencies in a WS transaction define the order of and constraints on the execution of activities. Thus, for executor and transaction levels, a control flow analysis can be derived to identify the test conditions. At the executor level, the flow is defined by the state transition model. Therefore, the test conditions can be defined in terms of the coverage of a particular set of elements in the structure of such model. At the transaction level, the flow is defined by the dependencies involved in the composition of a WS transaction. So the analysis should focus on the type of dependency and the possible behaviours of the involved participants.

Data: An executor generally accesses data elements (stored in data sources) during its execution. Such data elements can be concurrently accessed and updated by more than one activity of WS transactions. Thus data elements must be taken into account during the test process as they are crucial to produce correct outcome of WS transactions. By looking for patterns of data usage, risky situations are identified and more test conditions can be defined.

Control: The decision of an executor to move from one state to another may depend on the value of one or more data elements. This is called a control decision. In addition, there are control decisions during the flow of execution specified by the dependencies. For example in an `exclusion` dependency, the control decision decides which activity is to be selected and started. The goal of testing the control feature is to exercise different values of the data elements that are involved in the control decisions.

3.3. Depth dimension: A combination of test coverage items

Test criteria are used to define the test conditions and identify the test coverage items. The set of test cases must cover all the test coverage items, which can be achieved in different ways, depending on the required test effort. Thus different strategies are used to combine the test coverage items that will be exercised by the test cases. For instance, stronger test criteria should be applied in the areas with greater risk exposure in order to achieve an effective testing. The maximum effort would be to generate all possible combinations between the test coverage items and define a test case to cover each combination. On the other hand, the minimum effort would be to simply cover all the test coverage items using the lowest number of test cases. Thus the test criteria propose different test efforts ranging from minimum to maximum efforts.

4. Classification-Tree analysis of the dependencies

Testing at the overall transaction level is a complex process as it involves various dependencies among different activities (or sub-transactions) such as type of relationship (e.g. merge, union, etc), external conditions derived from the business logic and cardinality of the union. The situation becomes further complicated as for each dependency there exists a large set of possible situations to analyse. It is, therefore important to identify, organize and classify those situations. A well-known method to classify situations for testing purposes is the Classification-tree (CT) approach [46]. CT approach has been successfully used in both academic and industrial sectors [47, 48]. In the proposed approach, the *test basis* is the overall WS transaction while the *test items* are the dependencies. For each dependency we identify the test conditions and test coverage items through generation of a CT. Note that *flow*, *data* and *control* (*feature dimension*) are all taken into account during the tree generation. Section 5 gives further details on the test coverage items (*depth dimension*).

In order to explain the process of CT-based analysis of dependencies, we first present the generalized transaction model [10]. We then describe the process for elaborating the classification tree for each dependency using an example of the `merge` dependency. The CTs for other dependencies are shown in the Annex I.

4.1. Web services transaction model

A **Web Service transaction**, *wT*, is a sequence of logically linked activities (or sub-transactions) that must execute consistently in order to achieve an agreed outcome. 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 (sub-transaction). Each activity is executed by an *executor* [6]. An activity is compensatable if a compensating activity exists within the *wT* to undo its actions. A *wT* can have nested structure which contains different sub-transactions. A **compensation** is an activity that undoes from a semantic point of view the actions performed by another activity.

A **dependency** defines a relationship between a set of activities. In other words, **dependencies** are constraints that define an order of execution between (concurrent) activities. Dependencies are further explained in sub-section 4.1.2 below.

An **executor** is a web service which is responsible for executing a specific activity or a sub-transaction. As explained below, an executor passes through various states during the processing of a *wT*.

4.1.1. States of an executor

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. Figure 2 the state transition diagram of an executer. 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 is waiting to be executed. An executor is in the **Active** state when it has executed the *begin* primitive action but has not f nished 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 is in the **Compensated** state after it has executed the *compensate* primitive action. That is, its actions have been undone by executing a compensating activity.

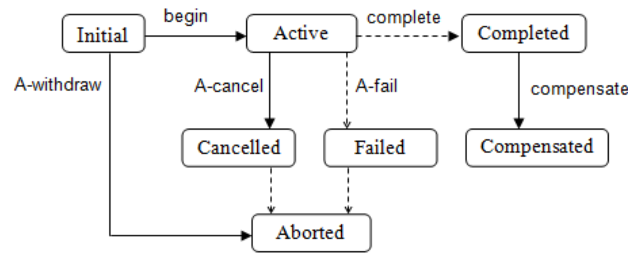


Figure 2. States and transitions of an executor

4.1.2. 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 relationship between the data elements used by the activities. These specify relations according to read and write operations on shared data elements. **Control dependencies** are hybrid dependencies (a mix of flow and data dependencies). These dependencies refer to *feature dimension* described in Section 3.2. Different types of dependencies can be combined. Consider an example of a 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).

A **data element** is a piece of information accessed by *wT*. An activity is said to **write** a data element if it generates or modifies 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 modifies (writes) the data element d_2. In other words, `A` requires d_1 to produce d_2.

A dependency is said to be **final** if it is not the input to any other dependency. A dependency is said to be **composite** if its input includes another dependency. A dependency is said to be **completed** if the necessary activities have been completed. For example in *Exclusive*, the dependency is completed if exactly one activity is completed. In *Join*, the dependency is completed if all activities have been completed.

The dependencies used in the proposed criteria are graphically represented using a BMPN based notation as shown in Table 1.

Name	Notation	Description
Sequence	A_1 → A_2	The activity A_1 must complete before activity A_2 can begin
Alternative		Only one activity can begin.
Fork		All the activities begin.
Merge		At least one activity must complete before another can begin. Extra conditions can be specified.
Join		All activities must complete before another can begin.
Exclusion		Only one activity can complete
Write	d_1 - -> A - -> d_2	One activity produces a data element and it may require another data element

Table 1. Representation of dependencies

4.2. Generation of Classification-Trees for Dependencies

CT relies on the knowledge of the environment in order to provide a step-wise intuitive approach and to define test cases. In the context of WS transactions, this knowledge is included in the transaction model in terms of dependencies and the behaviour of activities. The main features of CT approach are summarized as follows:

(a) To analyze the test basis in order to select the test items (the dependencies in our context). Each test item (a dependency type) is regarded under various aspects assessed as relevant for the test.

(b) For each aspect, disjoint and complete classifications are formed. Classes resulting from these classifications may be further classified (recursively). These identify the test conditions that are relevant for testing purposes. The stepwise partition of the input domain by means of classifications is represented graphically in the form of a tree.

(c) Partition the classes into test coverage items: these represent significant values for each class from the tester's view-point. It is represented graphically as the leaf nodes of the tree.

(d) To determine constraints among choices to prevent the construction of unnecessary combinations of choices.

(e) To design a set of test cases that covers all the test coverage items derived from the test conditions.

In the proposed method, we define a classification tree for each type of dependency. The classification conceives the relevant aspects that can influence the test process. Trees are constructed according to the following steps.

1. The first step is to identify the test items. The dependencies between the activities of a WS transaction are used as test items.
2. We identify the relevant features (classes) for the dependency. These form the test conditions that are used to derive the test coverage items. There are two types of possible classes: *orthogonal* and *exclusive (or non-orthogonal)*.

   A class is *orthogonal* if it can be combined with other classes. By analyzing the logic of the dependencies used in the transactions, we have identified the following orthogonal classes:

   - *Cardinality*: number of activities involved in the situation defined by the dependency.
   - *Behaviour*: sequence of states/transitions of the activities involved in the situation.
   - *Selected*: specific activity involved in the situation
   - *Former*: set of activities that act as input in the dependency
   - *Latter*: set of activities that act as output in the dependency

   A set of sibling classes are *exclusive* (non-orthogonal) if the value of one excludes the rest of them. During the elaboration of the CT, we have identified the following exclusive classes:

   - *Finished:* activity has either completed or compensated
   - *Running:* activity has not finished yet
   - *Aborted:* activity has achieved the aborted state
   - *Valid:* requirements of the dependency are fulfilled
   - *Invalid:* requirements of the dependency are not fulfilled

   The previous classes are hierarchically organized. In some situations these are recursively classified according to their logic so as to achieve elementary classes. A class is elementary if it is not further classified and, thus, specific values for such class can be defined.

3. Each class is divided into further classes and/or values. Each time a class is classified into further classes or values, a new deep level can be defined. These deep levels allow for defining different test effort as discussed in Section 5.1.

4. A set of constraints between the values/classes is specified. For example in the *Sequence* CT (see Annex I), there is a value (belonging to the class *latter behaviour)* that defines the activity must complete and it is not compatible

with the value of the class *former behaviour* that defines the activity to be withdrawn. The reason is due to the logic of the *Sequence* dependency; if the former activity does not complete, the latter cannot begin.

5. The values determined in the CT define the test coverage items. Further details on the criteria and the generation of coverage items are presented in Section 5.

Figure 3 shows the generic structure of a dependency classification-tree.

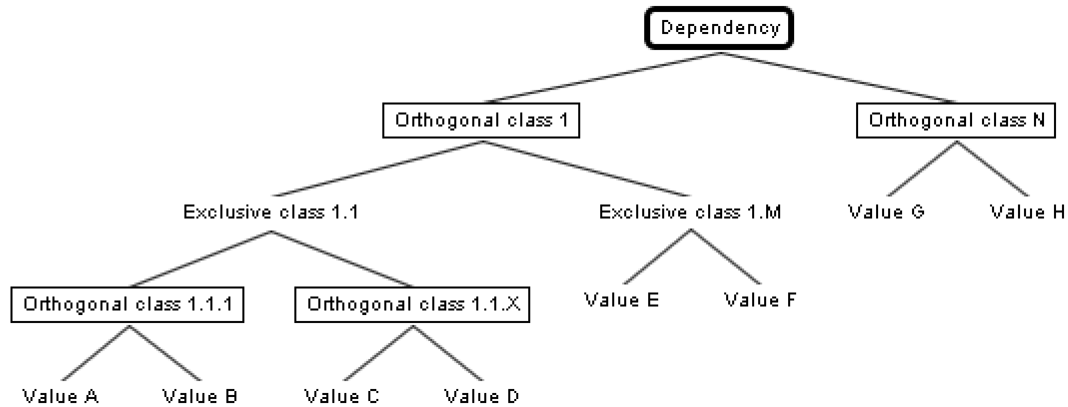


Figure 3. Concepts in a dependency classification-tree

4.3. Classification tree for Merge dependency

This section illustrates the process of designing classification tree for the *Merge* dependency according to the aforementioned steps (section 4.2). Figure 4 depicts the classification tree for the Merge dependencies. Classification trees for other dependencies (Sequence, Join, Fork. Alternative, Exclusion, and Write) are shown in Annex I. Orthogonal classes are shown as rectangles. Exclusive classes and concrete values are shown without rectangles. Leaf nodes in the tree represent values. The dotted lines represent the separation between the deepest level and level above. The criteria for defining the test coverage items depend on this separation as is shown in the next section.

The Merge dependency defines relationships between various activities of a WS transaction. It defines that at least one of the previous activities must complete before the latter can begin. The main feature we address in the tree is to show that the desired behaviour and constraints of the whole dependency are fulfilled or not. Thus we identified two exclusive classes at the top level. *Valid* class classifies the situations where at least one activity has completed, whereas *Invalid* class defines the situations where no single activity has completed successfully.

In the case of *Valid* class, we identify two orthogonal classes in order to represent the *Cardinality* of the completed activities: *Only one* and *More than one*. We define such cardinality in order to specify the condition that at least one activity must be completed.

Selected class represents the activity which in the Completed state. i.e., an activity is completed. This class maintains a list of values for each activity involved in the dependency. The rest of activities involved are represented through the *Running* class. This class shows whether activities are still running or not. If they are still running it means all of them are in the Active state. If they are not running and are not completed, it means that they are aborted (*Aborted* class) due to some unexpected situation. We identify four types of the *Aborted* class: withdrawn, cancelled, failed, combination (aborted in different ways). These values represent the possible types of abort of an activity and an extra value; which is the combination of different aborts types that is needed from a testing point of view.

If more than one activity has completed, we classified the behaviour of the rest of activities (*Rest of activities* class) in the same way as in the *Aborted* class (explained above).

The other branch of the tree (*Invalid* class) means that the dependency was not fulfilled. In other words, no activity has completed. So the relevant feature to test is the way in which the activities have finished. Since there are different ways to abort, it is necessary to test all those possibilities. We use the same four values to classify the behaviour of the non-completed activities: all were withdrawn, all were cancelled, all failed or all of them have aborted but in different ways.

In this dependency, there are no extra constraints about combining leaf nodes apart from the specification by the orthogonality of some classes as described above. The way the leaf nodes are used to define the test coverage items are explained in Section 5.

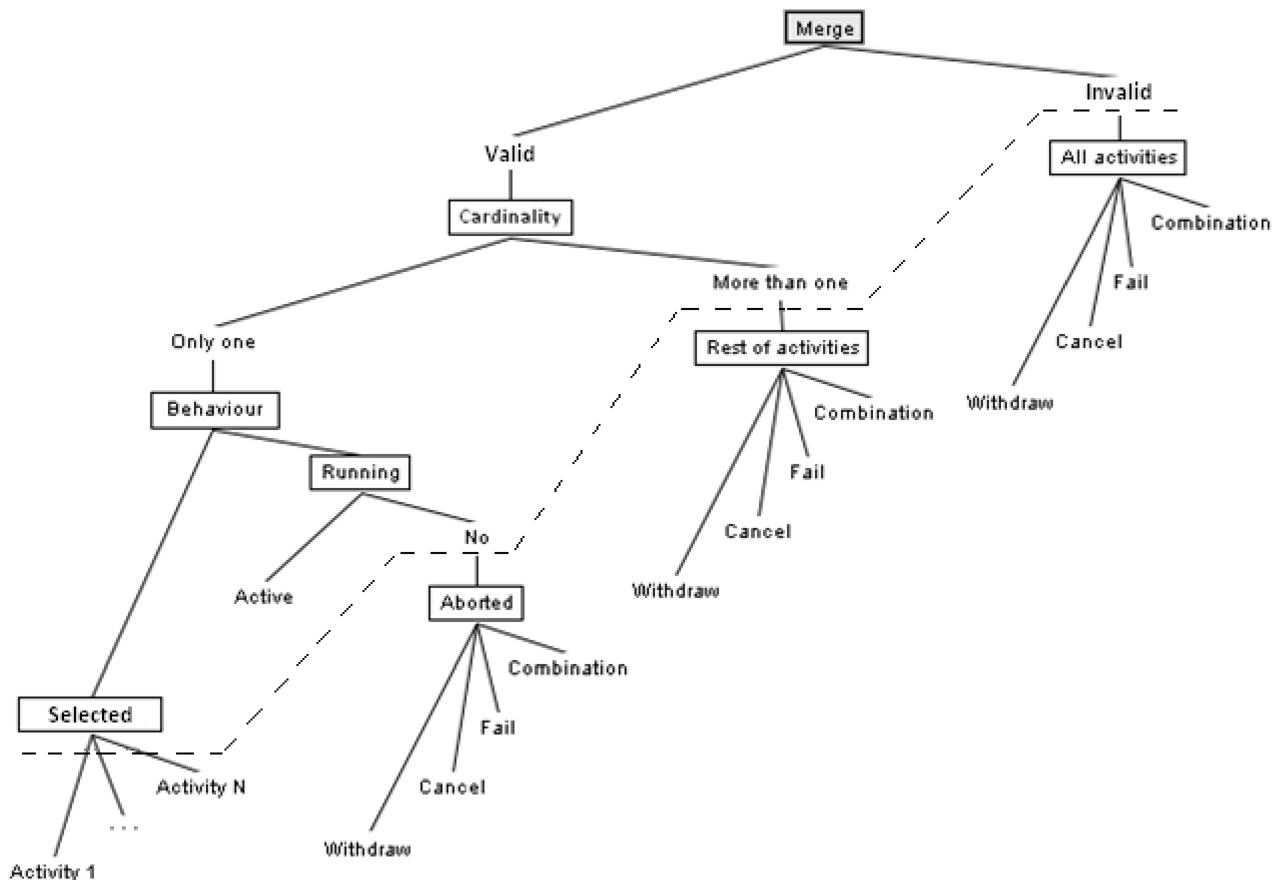


Figure 4. Merge dependency classification-tree

5. Test case design

The goal of the test case design process is to achieve a suitable test suite. A test suite is a set of test cases that meet a test criterion. Each test case is designed to exercise a combination of the test coverage items. All the test cases (test suite) must cover (exercise) all the identified test coverage items. In this work, the test technique is the CT method and the test criteria used to combine the test coverage items are presented below.

According to the CT method (see Section 4), the leaf nodes define the test coverage items that are combined to generate the test cases. In the context of testing WS Transactions at the *transaction level*, a test case defines a specific scenario that goes through different dependencies. Therefore, a test case cannot be defined only in terms of one CT (i.e. the test coverage items derived from a dependency), but it should be defined from a set of CTs. That is why, it is necessary to refine the concept of test coverage item in the CTs in order to represent combinations of leaf nodes (Combined Test Coverage items). In this way, these Combined Test Coverage items will be combined again leading to the scenarios that constitute the test cases. Two kinds of coverage items defined are:

- *Primitive Test Coverage Item (Primitive TCI)*: It shows each value of a class in the analysis. It is shown as a leaf node in the CT.

- *Combined Test Coverage Item (Combined TCI)*: The specific behavior that all involved activities in the dependency must follow. It is generated through the combination of its Primitive TCIs using different combination criteria (of the *depth dimension*).

Figure 5 illustrates an example of Combined TCI derived from combining two Primitive Test Coverage Items. Note that the states are specified in brackets and transitions between dashes.

The criterion to generate the Combined TCIs is related to the test effort required. Therefore, it is included in the *depth dimension* as described in Section 3. The following sub-section (5.1) presents different criteria for the use and combination of the Primitive TCIs. The strategy used to organize and combine the Combined TCIs in scenarios that define a test case is presented in Section 5.2.

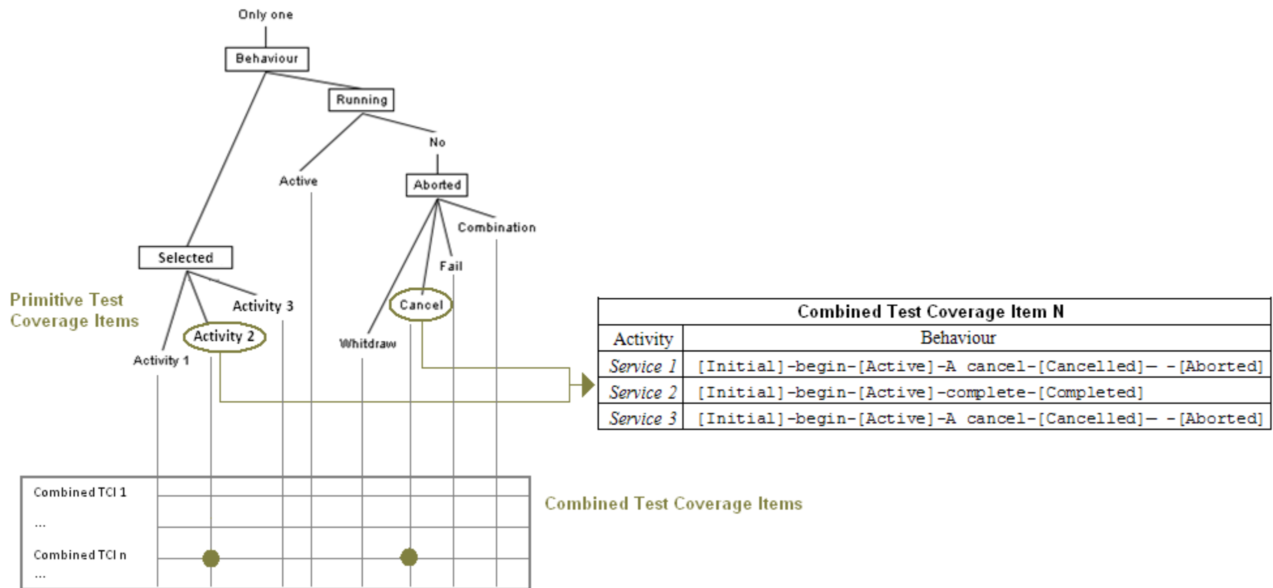


Figure 5. Combination of Primitive TCIs

5.1. Depth Dimension: Generation of the Combined Test Coverage Items

In order to combine the Primitive TCIs, we propose a family of coverage strategies by taking into account two orthogonal aspects:

- *Primitive TCI selection*: This is to select the leaf nodes which are to be used in the combination
- *Primitive TCI combination*: This is to combine the selected Primitive TCIs

The above two aspects allow for adjusting the number of Combined TCIs that will be generated. Note that such combinations must fulfil the constraints defined in the CT. The more the Combined TCIs defined the more the requirements for the test suite. Thus more test cases are generated which result in the increase in test effort. But more test cases potentially increase the effectiveness of testing. Considering these two aspects we achieve four different criteria that allow for adjusting the testing thoroughness.

5.1.1. Primitive TCI selection

We use the depth level information in order to select the Primitive TCI (leaf nodes). The selection is done using the following two levels:

- *Strong level*: It uses the deepest level (called *N level*) in order to cover all Primitive TCIs. Assume that there are three activities involved in the Merge CT (Figure 4), then this criterion requires using the 16 Primitive TCIs which are found in the tree.
- *Weak level*: This level, called *N-1 level,* requires covering all non-elementary classes but with one value for each elementary class. In this criterion, a random leaf node (from child nodes) is selected. In the Merge CT, this criterion requires using 5 Primitive TCIs: one value for each class *Selected*, *No*, *More than One* and *Valid*, plus the *Active* value in the class *Running*. The number of Primitive TCI is widely reduced. But from a testing point of view, the randomly selected leaf node can be less significant, than the other candidate node. It also brings a nondeterministic factor in the design.

Note that the *strong level* criterion subsumes the *weak level* criterion. The dotted line in the Figure 4 separates the N level to N-1 level in the *Merge* dependency.

5.1.2. Primitive TCI combination

Combined TCIs are generated from the Primitive TCIs. There is a range of criteria that can be used. The simplest coverage criterion, *each-used* coverage, does not enforce any requirement on how to combine the Primitive TCIs. The more complex coverage criteria, such as *pair-wise* or *N-wise* coverage, is concerned with (sub-) combinations of interesting values of different parameters [49]. In this work we use the combination of the *simplest* and the strongest *criteria*.

- *Each-used*: it requires that every selected Primitive TCI to be included in at least one Combined TCI — which is derived from this dependency using the lowest number of Combined TCIs possible. It is the weakest coverage criterion.
- *N-wise*: it requires all possible combinations of all selected Primitive TCIs to be included in the set of Combined TCIs which is derived from this dependency. It is the strongest coverage criterion and thus subsumes the *each-used* criterion.

The Primitive TCIs specify requirements in terms of behaviours of their activities. But in a composite dependency, such requirements may refer to another dependency. In the latter, the dependency involved as argument (argument dependency) is considered as an activity in terms of behaviour. In other words, if the Primitive TCI requires that such activity (argument dependency) has to complete, a random scenario is selected where the activity (argument dependency) is completed. If the Primitive TCI requires the activity (argument dependency) to cancel/abort/fail, all the activities involved in the argument dependency will take such behaviour.

Figure 6 depicts the four set of Combined TCIs for the Merge CT. These are generated by combining the orthogonal criteria of level and combination. In the *strong level,* each leaf node is a Primitive TCI while in the *weak level*, a leaf node of each elementary class was randomly selected. Each row defines a Combined TCI which is achieved by applying the specified combination criterion.

At the *weak level* the same Combined TCIs are generated for both *N-wise* and *each-used* criterion. This is because *weak level* prunes the leaf nodes in the elementary classes *Selected*, *Aborted*, *Rest of Activities* and *All Activities.* This pruning eliminates the possibility of combination. But this does not occur in other classification-trees. The *strong level* generates more Combined TCIs than *weak level* since the former always selects more Primitive TCIs. This allows for analysing the dependency more rigorously. However, the test effort (i.e., the number of Combined TCI generated) can considerably grow if strong combination criteria are used.

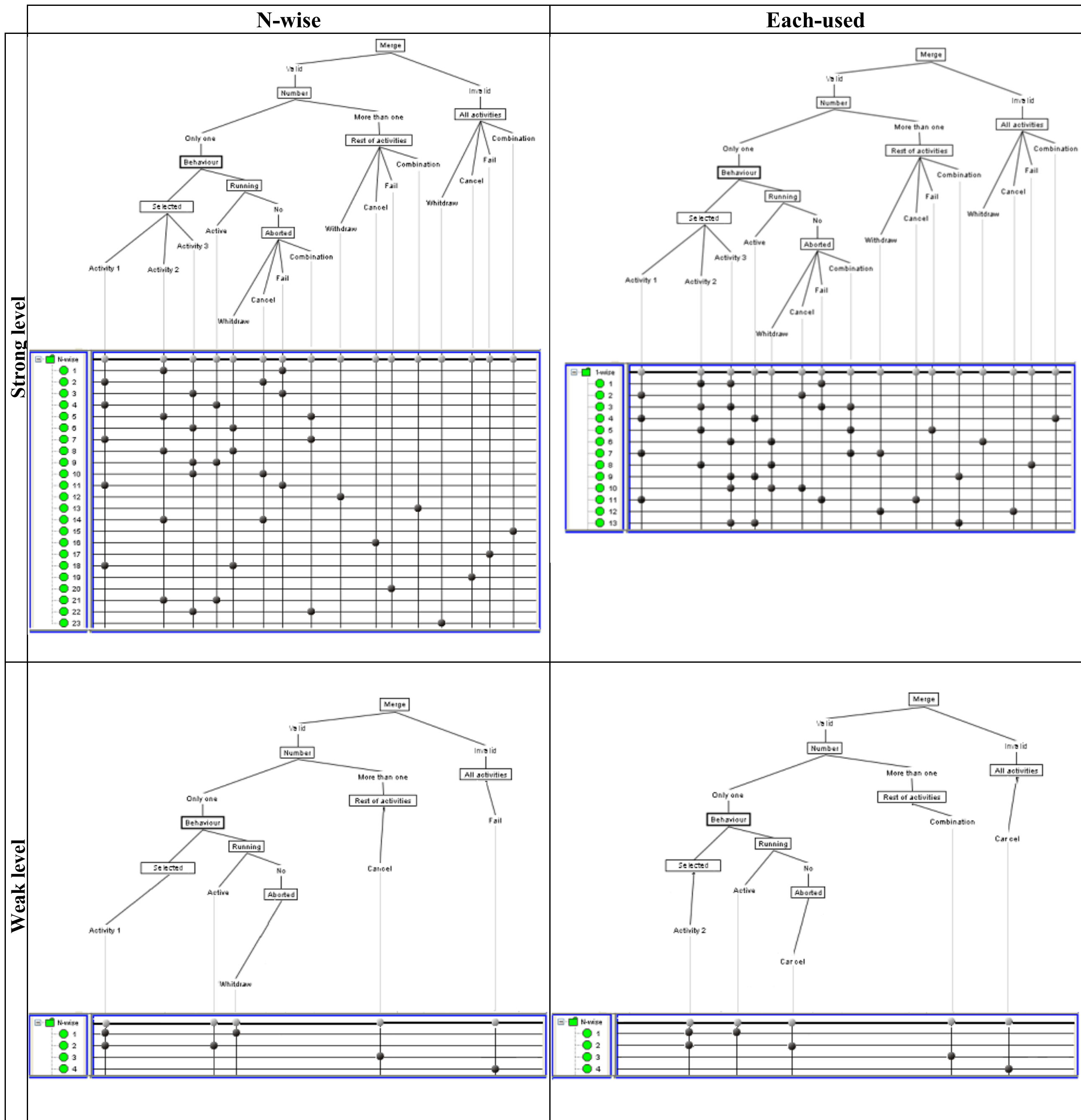


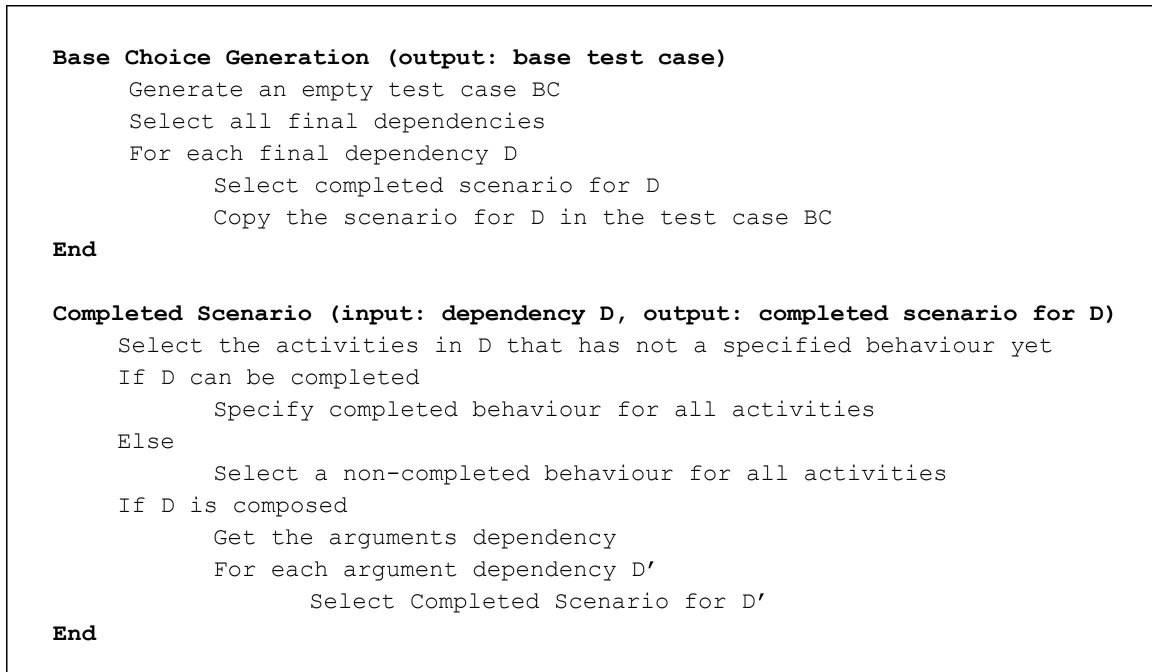
Figure 6. Generation of Combined TCIs

5.2. Generating the test cases

Once all the Combined TCIs are defined, the next step is to generate a set of test cases that can cover them. Note that one test case can cover more than one Combined TCI. To generate the test suite we use the *base choice* (BC) strategy [49]. BC is a determinist iterative strategy; which means that given a base test case the same test suite is produced every time. The f rst step of BC is to identify a base test case. The base test case combines the most ‘important’ value for each parameter. Importance may be based on any pre-defined criterion such as most common, simplest, or smallest. From the base test case, new test cases are created by varying the minimum number of values at a time while keeping the values of the other parameters f xed.

In the context of testing WS Transactions, the parameters are the behaviour of each activity (i.e. sequence of states/transitions of its executor, see Figure 2). To generate the base test case we adopt the criterion of “maximum of dependencies completed”. So the base test will define specific behaviour for all activities forming a scenario where the number of dependencies, that are completed, is maximum.

The algorithm to generate the base test case is shown in Figure 7. The strategy selects, for each final dependency, a scenario where the dependency is completed. Then the strategy is recursively applied to the dependencies included in the input of final dependencies in order to specify the behaviour for the activities whose behaviour was not previously defined. If it is impossible to select a scenario to complete a dependency without modifying the behaviour of a previously specified activity, then the dependency takes a non-complete scenario while keeping the behaviour of such activity fixed.

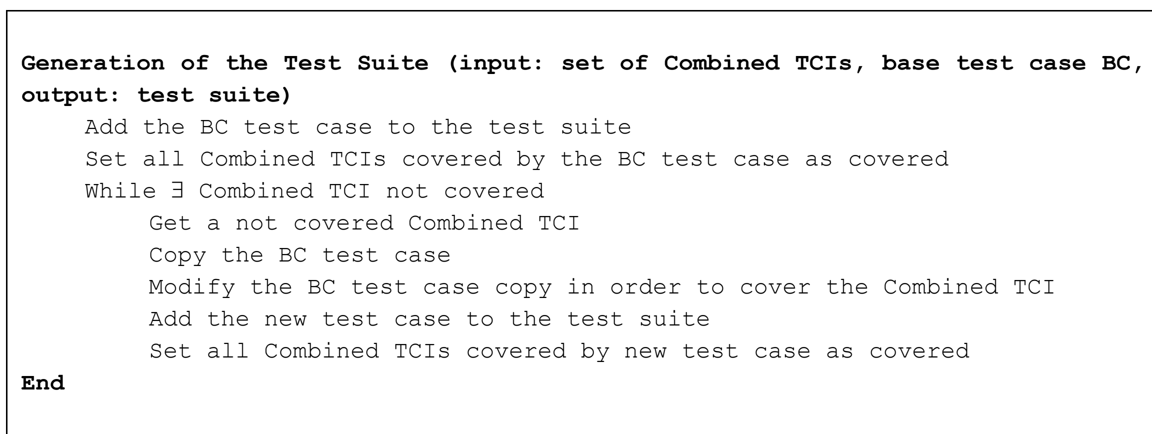


```
Base Choice Generation (output: base test case)
     Generate an empty test case BC
     Select all final dependencies
     For each final dependency D
           Select completed scenario for D
           Copy the scenario for D in the test case BC
End

Completed Scenario (input: dependency D, output: completed scenario for D)
    Select the activities in D that has not a specified behaviour yet
    If D can be completed
           Specify completed behaviour for all activities
    Else
           Select a non-completed behaviour for all activities
    If D is composed
           Get the arguments dependency
           For each argument dependency D’
                  Select Completed Scenario for D’
End
```

Figure 7. Base Case generation algorithm

Once the base test case is defined, the strategy creates the rest of necessary test cases. The algorithm to generate the test suite is shown in Figure 8. The strategy gets the test coverage items which are not covered by the base test case. It creates new test cases by varying the behaviour of the base test case´s activities in order to cover all the test coverage items. This gives a set of test cases that covers all Combined TCIs which are generated previously.



```
Generation of the Test Suite (input: set of Combined TCIs, base test case BC,
output: test suite)
     Add the BC test case to the test suite
     Set all Combined TCIs covered by the BC test case as covered
     While ∃ Combined TCI not covered
          Get a not covered Combined TCI
          Copy the BC test case
          Modify the BC test case copy in order to cover the Combined TCI
          Add the new test case to the test suite
          Set all Combined TCIs covered by new test case as covered
End
```

Figure 8. Test Suite generation algorithm

Test suite is obtained by following the steps of the above algorithm. Each test case defines a concrete scenario for the overall WS transaction by specifying the behaviour of all its activities. A test case can cover one Combined TCI for each dependency of the transaction (as shown in Figure 9). This provides a set of generated test cases that cover all the Combined TCI.

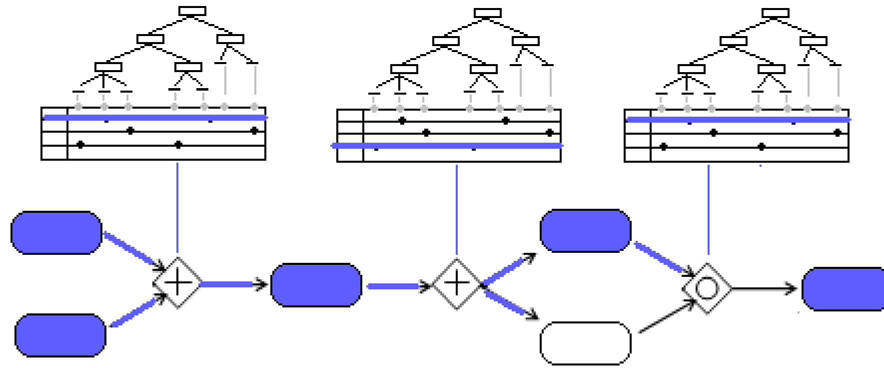


Figure 9. WS Transaction test case

6. Evaluation

In this section we evaluate the proposed criteria by taking into account the following research questions:

RQ1: Effectiveness of the proposed criteria in detecting failures in WS Transactions.
RQ2: Usefulness of the proposed criteria useful in adjusting the test efforts and providing a trade-off in terms of cost-benefit.
RQ3: Resiliency of the proposed criteria to different types of defects or failures

6.1. Case study

In order to evaluate the proposed criteria we have developed a prototype application which implements the widely used *Travel Agency* case study [11-15].

Travel Agency is an application in which customers are offered with the facilities for making travel arrangements as follows. The Agency service receives an itinerary from a customer. After checking the itinerary for errors, it determines the types of reservations to make and sends simultaneous requests to the transport, hotel and car rental providers. There are three alternative airline companies, two hotel agencies, one train company and one vehicle reservation service.

If any of the reservation tasks fails, the itinerary is cancelled by performing the compensatory action and the customer is notified of the problem. Agency service waits for confirmation of the reservation requests. Upon receipt of all confirmations, the Agency service informs the customer about the reservation confirmation and final itinerary details. The Agency then contacts the payment services charge the customer´s credit card of the total amount. The payment services also charges an extra 1% for using their service.

Travel Agency is a distributed software application written in Java 1.5. The application includes 23 Java classes and 2,540 Java lines of code (LoC). Each service is composed of two classes: service*Logic* and service*WS*. The services that make reservations (flights, hotels, train and car) also have a service*Reservation* class. The service*Logic* classes implement the business logic of the activity, for example, checking the availability and booking of a room in a hotel. The service*WS* wraps the logic class and other classes required by the service. Auxiliary classes, *Customer* and *Itinerary,* are used by all the services. The class, *TAcontext,* represents the data elements shared by the activities (itinerary, amount and customer data) while *TAflow* manages the execution of the all services.

6.2. Transactional modeling of the Case study

We model the travel agency case study according to the transaction model presented in Section 4.1. In other words, we model the request for travel arrangement as a WS transaction. Figure 10 depicts the modelling of the travel agency and shows some of the important activities, data elements and dependencies

The services (and their activities) involved in the travel agency WS transaction are defined as follow.

- *Agency*: Checks if the departure and arrival cities are under the coverage of the agency. It also coordinates the flow execution of the activities.
- *Gold Air*: An airline with high availability but is the most expensive

- *Cheap Air*: An airline with cheaper prices but less availability
- *Train*: Train tickets service
- *Five Star Hotel*: A luxury hotel chain. High availability and high cost.
- *Two Star Hotel*: A low cost hotel chain.
- *Car*: Rental cars service
- *Payment*: Credit card services for online payments

The following data elements are used by the above activities:

- *Itinerary (I):* Departure and arrival cities and dates
- *Hotel reservation (H):* Hotel address, date of arrival and number of nights booked at the hotel
- *Flight reservation (F):* City, date and time of departure and city, date and time of return
- *Train reservation (T):* City, date and time of departure and city, date and time of return
- *Car reservation (R):* City, date and number of days booked
- *Amount (A):* Amount to be charged to the client
- *Credit balance (B):* The customer credit balance

The dependencies among the above activities are defined as follows:

- *D1: Fork (Gold Air, Cheap Air, Train, 5*Hotel, 2*Hotel, Car)*
- *D2 :Exclusion (5*Hotel, 2*Hotel)*
- *D3 :Merge (Gold Air, Cheap Air, Train)*
- *D4: Join (D2, D3, Car)*
- *D5: Sequence (Agency, D1)*
- *D6: Sequence (D4, Payment)*
- *D7: Write({Agency, Gold Air, Cheap Air, Train Air, 5*Hotel, 2*Hotel, Car },{Payment})*

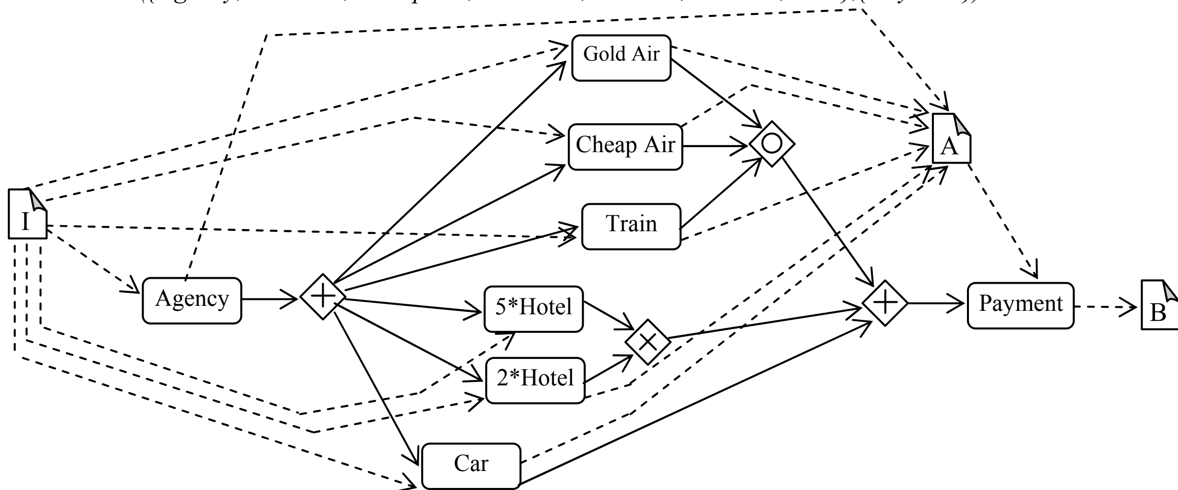


Figure 10. Modelling of the Travel Agency case study

6.3. Experimental parameters

Existing works on testing (non-transactional) web services focus on the unit testing of flow management such as WS-BPEL process [50-54]. But they do not address the evaluation of fault detection in a particular implementation of services. We measure the effectiveness of the proposed criteria as the degree on which the generated test cases are able to detect defects injected in a concrete implementation (described above) of the overall WS transaction.

We use a mutation approach in order to inject faults in the WS transaction of the travel agency Mutation testing has been widely accepted as the test adequacy criteria. The idea is to make many small changes called mutants in a given

program (or WS transaction in our case). Small changes of the original program are expected to produce observable different outputs. A mutant is said to be killed if it gives different outputs from the original with some test case. The Mutation Score (MC) is the relation between the number of mutants killed (KM) by the test suite and the number of total generated mutants (TM). Formally, it is calculated as: `MC=KM/TM * 100`. Mutation score is, therefore, an objective measure to evaluate the effectiveness of a test suite.

To apply mutation to the WS transaction of *Travel Agency*, we have used the MuJava tool [55] which generates two types of mutants. Traditional mutations are generated by applying syntactic actions such as an arithmetic or logic operator replacement in the code. Class mutations are generated by applying semantic actions such as changing the access modifier of a variable. We have applied both types of mutants in all classes obtaining a total of 2.507 faulty versions of the *Travel Agency* WS transaction. The number of mutants generated by each Java class is shown in Table 2.

Type of class	Class	Total Mutants	Traditional Mutants	Class mutants
service	`AgencyWS`	104	97	7
service	`CarWS`	173	156	17
service	`CheapAirWS`	125	116	9
service	`FiveHotelWS`	173	156	17
service	`GoldAirWS`	133	124	9
service	`PaymentWS`	112	105	7
service	`TrainWS`	133	124	9
service	`TwoHotelWS`	173	156	17
logic	`AgencyLogic`	0	0	0
logic	`CarLogic`	238	233	5
logic	`FlightLogic`	387	366	21
logic	`HotelLogic`	231	226	5
logic	`PaymentLogic`	8	6	2
logic	`TrainLogic`	207	195	12
reservation	`FlightReservation`	28	10	18
reservation	`HotelReservation`	28	20	8
reservation	`TrainReservation`	28	10	18
reservation	`CarReservation`	28	20	8
auxiliar	`Customer`	78	59	19
auxiliar	`Itinerary`	7	0	7
auxiliar	`PaymentTransfer`	10	8	2
transaction	`TAcontext`	59	14	45
transaction	`TAflow`	217	211	6

Table 2. Combined TCIs generated by the criteria

To evaluate the requirement RQ2, we assess the cost-benefit relation of using the different criteria for generating the Combined TCIs, and the test cases. The cost is estimated according to the number of test cases generated. The test benefit is highly related to the number of defects that the test suite can reveal. It is therefore approximated by the number of mutants killed. We define the cost-benefit relation (CB) as the relationship between the number of test cases generated by the criterion (TC) and the mutants killed for that set of test cases (KM), CB=TC/MT. In order to compare different CB values, we normalize the values (CBN) ranging from 0 – 1 and using the highest CB with test suite of CBN=1. Thus, the metric used to address RQ2 is the CBN.

6.4. Experimental Results

We obtained the results using the following three steps:

The first step is to generate the Combined TCIs for each dependency. We developed a script that requests as input the type of dependency, the service classes that implement the involved activities, the level and combination criteria and generation of the set of Combined TCIs. The number of Combined TCIs generated for each dependency and each combination of criteria are shown in Table 3. As was expected, the *strong level* significantly increases the number of Combined TCIs generated independently of the combination criterion selected. Regarding the combination strategy,

we realize that *N-wise* criterion increases the Combined TCIs but the increase is only noticeable when this strategy is combined with the *strong level*. Also we see that the number of Combined TCI is very similar to the *weak level* criterion irrespective of the combination criterion used.

ID	Dependency	Strong level N-wise	Weak level N-wise	Strong level each-use	Weak level each use
D1	Fork	10	4	5	2
D2	Exclusion	18	9	4	3
D3	Merge	22	5	13	5
D4	Join	15	3	5	2
D5	Sequence	13	3	8	3
D6	Sequence	13	3	8	3
D7	Write	35	3	7	3
	Total	**126**	**30**	**50**	**21**

Table 3. Combined TCIs generated by the proposed criteria

The second step is to generate the test cases. We developed a script that receives the set of Combined TCIs of all dependencies involved in the WS transaction and then execute the algorithms (Section 5.2) to generate the test cases. Each test case specifies the behaviour that the activities have to follow during the execution of a WS transaction.

The third step is to automatically execute the test cases in the 2.507 mutant versions. Using this version the code was instrumented and the services were configured in a way that follows a specific behaviour. Thus, when the *Travel Agency* WS transaction starts, it reads a test case file and configures all the services.

The different test suites generated by combining the level and combination criteria were automatically executed using MuJava and the mutation score. The mutation score and cost-benefit results are summarized in Table 4. Mutation scores grouped by type of class are shown in Table 5. 'S' and 'W' mean *strong level* and *weak level* respectively, while 'N' and 'E' respectively mean *N-wise* and *each-used*. Information about the number of the mutants killed, mutants alived, traditional mutants score and class mutation score for each class is shown in Annex II.

Test suite level	Strong level N-wise	Weak level N-wise	Strong level each-used	Weak level each-used
Mutation Score	99,85	81,19	92,5	65,45
Number of Test Cases (TC)	71	27	37	17
Number of mutants killed (KM)	2676	2176	2479	1754
Cost-benefit relation (CB)	0,02653	0,01241	0,01493	0,00969
CB normalized (CBN)	1	0,47	0,56	0,37

Table 4. Test suites results

Type of class	Number of classes	Mutation score				Traditional mutation score				Class mutation score			
		S/N	W/N	S/E	W/E	S/N	W/N	S/E	W/E	S/N	W/N	S/E	W/E
Service	8	99,91	77,98	89,47	65,79	99,88	77,38	89,13	66,13	100,00	78,00	87,63	57,38
Logic	6	99,85	81,07	92,62	60,07	99,80	82,40	95,60	60,60	98,20	55,20	74,00	36,80
Reservation	4	99,11	61,61	83,04	40,18	97,50	73,75	95,00	57,00	100,00	47,75	74,25	34,50
Auxiliar	3	100,00	49,73	98,72	30,07	100,00	45,50	97,00	21,00	100,00	48,67	100,00	19,00
Transaction	2	100,00	91,45	100,00	98,00	100,00	99,50	100,00	100,00	100,00	56,50	100,00	65,00
Average		99,77	72,37	92,77	58,82	99,44	75,71	95,35	60,95	99,64	57,22	87,18	42,54

Table 5. Results by type of class

6.5. Discussion

The results presented in Table 4 show the effectiveness of the proposed criteria which is measured in terms of the mutation score. The results show that the proposed criteria are indeed effective thus meeting the RQ1. All test suites achieve a mutation score greater than 65% and some score greater than 80%. We see that the *strong level* generates test suites that reach very high effectiveness with mutation scores greater than 90%. According to the type of class, Table 5 shows that the *weak level* criterion achieves notably inferior mutation scores, especially in reservation and auxiliary classes. These two types of classes are less complex than the other. The results suggest that the *strong level* is more suitable for simple classes.

With regard to the RQ2, we see that different level and combination criteria lead to different test efforts measured in terms of the number of test cases generated (TC). The benefit (killed mutants, KM) also differs in different test suites as shown above. There are significant differences in the test efforts (TC). The lowest value of CBN is achieved by the criteria combination *weak level* / *each used*. On the other hand, a similar CBN value is achieved by *weak level* / *N-wise* (0,47) and *strong level* / *each-used* (0,56). Although the best CBN relation is reached by the *weak level / each-use* criteria, this combination achieves the lowest mutation score. Further, *strong level / N-wise* have the worst CBN value but it achieves the best mutation score. The experiments also revealed that the *weak level* / *N-wise* and *weak level* / *each- used* combinations achieve a high mutation score. Thus these results provide a metric to the tester in order to decide the test effort in terms of cost or benefit.

In relation to the RQ3, we consider two types of injected faults: traditional mutants and class mutants. According to the results summarized in Table 5, we see that the mutation score achieved in the traditional mutants is greater than the one achieved in the class mutation. The average mutation scores of all classes (see Annex II) show the same tendency. Thus the results show that the type of fault influences the effectiveness of the testing criteria.

In some cases, the above evaluation may have limitations. The first issue is that the mutation technique simulates the faults that could appear during the development of WS Transactions. But due to the lack of information about actual faults, we are not certain how representative these injected faults are. However, empirical studies, comparing the fault detection ability of test suites on hand-seeded, automatically-generated (mutation) and real-world faults, suggest that the generated mutants provide a good indication of the fault detection ability of a test suite [56]. This proves the validity of our evaluation. The second issue, which is common in software engineering, is that how representative the case study is? We have used the case study which is widely adapted in the literature. This ensures that the case study is highly representative and is not limited to a particular situation or scenario.

7. Conclusions

In this paper we have proposed novel multi-dimensional criteria for testing the WS Transactions. Our approach generates test cases according to the dependencies between the activities involved in a WS transaction. The proposed criteria elaborate a classification-tree analysis for each kind of dependency in order to identify the relevant test condition and test coverage items. The aim was to derive the test cases that thoroughly exercise all dependencies. Two orthogonal families of test criteria are used for the test coverage item selection (*strong level*, *weak level*) and the test coverage item combination (*N-wise*, *each-used*). To evaluate the proposed method we have used a well-known case study of *Travel Agency*. Evaluation results showed that the proposed criteria have the potential to design effective test cases for WS transactions and to allow the tester to adjust the method in terms of its effectiveness, test effort and cost-benefit analysis. It also provides the advantages of performing the testing process in a resource-scarce environment. Further the design of the test cases is automatically generated in order to meet the requirements of the distinguishing characteristics of WS transactions. It reduces the cost of the test design and also improves its effectiveness. It allows for adjusting the intensity of the test process by taking into account the time and effort. It also allows the tester to prioritize the tests by firstly using low-effort criteria and then subsequently complement them whenever additional test efforts are required.

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] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web Services Platform Architecture: Soap, Wsdl, Ws-Policy, Ws-Addressing, Ws-Bpel, Ws-Reliable Messaging and More*: Prentice Hall PTR, 2005.

[2] Z. Liangzhao, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, “QoS-aware middleware for Web services composition,” *Software Engineering, IEEE Transactions on,* vol. 30, no. 5, pp. 311-327, 2004.

[3] OASIS, "Business Transaction Protocol," http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=business-transaction, [29 Nov 2011, 2004].

[4] OASIS. "Web Services Business Activity," http://docs.oasis-open.org/ws-tx/wsba/2006/06.

[5] OASIS. "Web Services Composite Application Framework," https://www.oasis-open.org/committees/ws-caf.

[6] R. Casado, J. Tuya, and M. Younas, “Evaluating the effectiveness of the abstract transaction model in testing Web services transactions,” *Concurrency and Computation: Practice and Experience*, pp. in press, 2012.

[7] G. Canfora, and M. Penta, "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.

[8] M. Bozkurt, M. Harman, and Y. Hassoun, *Testing Web Services: A survey*, Department of Computer Science, King's College London, Technical Report TR-10-01, 2010.

[9] S. Bhiri, W. Gaaloul, C. Godart, O. Perrin, M. Zaremba, and W. Derguech, “Ensuring customised transactional reliability of composite services,” *Journal of Database Management,* vol. 22, no. 2, pp. 29, 2011.

[10] R. Casado, J. Tuya, and M. Younas, “A Family of Test Criteria for Web Services Transactions,” *Procedia Computer Science,* vol. 10, no. 0, pp. 880-887, 2012.

[11] L. Bocchi, C. Laneve, and G. Zavattaro, “A Calculus for Long-Running Transactions,” *Formal Methods for Open Object-Based Distributed Systems,* vol. 2884, pp. 124-138, 2003.

[12] R. Casado, J. Tuya, and M. Younas: 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

[13] J. Jiang, G. Yang, Y. Wu, and M. Shi, “CovaTM: a transaction model for cooperative applications,” in Proceedings of the 2002 ACM symposium on Applied computing, Madrid, Spain, 2002, pp. 329-335.

[14] A.-B. Arntsen, M. Mortensen, R. Karlsen, A. Andersen, and A. Munch-Ellingsen, “Flexible transaction processing in the Argos middleware,” in Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management, Nantes, France, 2008, pp. 12-17.

[15] R. T. Khachana, A. James, and R. Iqbal, “Relaxation of ACID properties in AuTrA, The adaptive user-defined transaction relaxing approach,” *Future Gener. Comput. Syst.,* vol. 27, no. 1, pp. 58-66, 2011.

[16] D. Yuetang, P. Frankl, and C. Zhongqiang: Testing database transaction concurrency. *Prooceedings of the Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 2003. 6-10 Oct. 2003;184-193

[17] Y. Deng, P. Frankl, and D. Chays, “Testing database transactions with AGENDA,” in Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, 2005, pp. 78-87.

[18] V. Guarnieri, N. Bombieri, G. Pravadelli, F. Fummi, H. Hantson, J. Raik, M. Jenihhin, and R. Ubar: Mutation analysis for SystemC designs at TLM. *Prooceedings of the Test Workshop (LATW), 2011 12th Latin American*, 2011. 27-30 March 2011;1-6

[19] C. Chin-Yao, H. Chih-Yuan, L. Kuen-Jong, and A. P. Su: Transaction Level Modeling and Design Space Exploration for SOC Test Architectures. *Prooceedings of the Asian Test Symposium, 2009. ATS '09.*, 2009. 23-26 Nov. 2009;200-205

[20] X.-D. Wu, Z.-W. Sun, and Z.-J. Xing: A data-centered transaction scheduling strategy of realtime database in micro-satellite ground test system. *Prooceedings of the Mechatronics and Automation, 2009. ICMA 2009. International Conference on*, 2009. 9-12 Aug. 2009;2952-2956

[21] M. Younas, and K.-M. Chao, “A tentative commit protocol for composite web services,” *Journal of computer and system sciences,* vol. 72, no. 7, pp. 1226-1237, 2006.

[22] R. M. Czekster, P. Fernandes, A. Sales, T. Webber, and A. F. Zorzo, “Stochastic Model for QoS Assessment in Multi-tier Web Services,” *Electron. Notes Theor. Comput. Sci.,* vol. 275, pp. 53-72, 2011.

[23] R. Lanotte, A. Maggiolo-Schettini, P. Milazzo, and A. Troina, “Design and verification of long-running transactions in a timed framework,” *Science of Computer Programming,* vol. 73, no. 2-3, pp. 76-94, 2008.

[24] M. Emmi, and R. Majumdar: Verifying Compensating Transactions. *Prooceedings of the International Conference Verification, Model Checking, and Abstract Interpretation*, 2007. 29-43

[25] N. Kokash, and F. Arbab, “Formal Design and Verification of Long-Running Transactions with Eclipse Coordination Tools,” *Services Computing, IEEE Transactions on,* vol. PP, no. 99, pp. 1-1, 2011.

[26] W. Gaaloul, M. Rouached, C. Godart, and M. Hauswirth, “Verifying composite service transactional behavior using event calculus,” in OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I, Vilamoura, Portugal, 2007, pp. 353-370.

[27] J. Li, H. Zhu, and J. He: Specifying and Verifying Web Transactions. *Prooceedings of the International Conference on Formal Techniques for Networked and Distributed Systems*, 2008. 149-168

[28] “Database transaction models for advanced applications,” Morgan Kaufmann Publishers, 1992 Pages.

[29] N. Ben Lakhal, T. Kobayashi, and H. Yokota, “FENECIA: failure endurable nested-transaction based execution of composite Web services with incorporated state analysis,” *The VLDB Journal,* vol. 18, no. 1, pp. 1-56, 2008.

[30] H. Garcia-Molina, and K. Salem, “Sagas,” in SIGMOD 87, 1987, pp. 249-259.

[31] G. Weikum, and H.-J. Schek, "Concepts and applications of multilevel transactions and open nested transactions," *Database transaction models for advanced applications*, pp. 515-553: Morgan Kaufmann Publishers Inc., 1992.

[32] C. Pu, G. E. Kaiser, and N. C. Hutchinson, “Split-Transactions for Open-Ended Activities,” in 14th International Conference on Very Large Data Bases, 1988, pp. 26-37.

[33] Reuter, “ConTracts: A Means for Extending Control Beyond Transaction Boundaries,” *Proceedings of the 3rd International Workshop on High Performance Transaction Systems*, 1989.

[34] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres, “Ensuring relaxed atomicity for flexible transactions in multidatabase systems,” *ACM SIGMOD Record*, 1994.

[35] M. Younas, B. Eaglestone, and R. Holton, “A formal treatment of a SACRED Protocol for Multidatabase Web Transactions,” *Database and Expert Systems Applications,* vol. 1873, pp. 899-908, 2000.

[36] OASIS, "Web Services Atomic Transaction," http://docs.oasis-open.org/ws-tx/wsat/2006/06, [29 Nov 2011, 2009].

[37] OASIS, “Web Services Coordination,,” *http://docs.oasis-open.org/ws-tx/wscoor/2006/06*, 2007.

[38] A. T. Endo, and A. d. S. Simão: A Systematic Review on Formal Testing Approaches for Web Services. *Prooceedings of the IV Brazilian Workshop on Systemati and Automated Software Testing*, 2010. Natal, Brasil, 89-98

[39] B. Antonio, M. Hernán, and S. Francesco, "Testing Service Composition," 2008.

[40] M. Palacios, J. Garcia-Fanjul, and J. Tuya, “Testing in Service Oriented Architectures with dynamic binding: A mapping study,” *Inf. Softw. Technol.,* vol. 53, no. 3, pp. 171-189, 2011.

[41] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, “Bringing white-box testing to Service Oriented Architectures through a Service Oriented Approach,” *J. Syst. Softw.,* vol. 84, no. 4, pp. 655-668, 2011.

[42] R. Casado, J. Tuya, and M. Younas: A Framework to Test Advanced Web Services Transactions. *Prooceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011. IEEE Computer Society: Berlin, 443-446

[43] R. Casado, J. Tuya, and C. Godart: Dependency-based criteria for testing web services transactional workflows. *Prooceedings of the Next Generation on Web Services Practices*, 2011. IEEE: Salamanca, Spain, 74-79

[44] R. Casado, J. Tuya, C. Godart, and M. Younas, “Test case design for transactional flows using a dependency-based approach,” *International Journal of Computer Information Systems and Industrial Management Applications,* vol. 5, pp. 20-40, 2012, 2012.

[45] R. Casado, J. Tuya, and M. Younas, “Testing the reliability of web services transactions in cooperative applications,” in Proceedings of the 27th Annual ACM Symposium on Applied Computing, Trento, Italy, 2012, pp. 743-748.

[46] M. Grochtmann, and K. Grimm, “Classification trees for partition testing,” *Software Testing, Verification and Reliability,* vol. 3, no. 2, pp. 63-82, 1993.

[47] T. Y. Chen, and P. L. Poon, “On the effectiveness of classification trees for test case construction,” *Information and Software Technology,* vol. 40, no. 13, pp. 765-775, 1998.

[48] H. Singh, M. Conrad, and S. Sadeghipour, “Test Case Design Based on Z and the Classification-Tree Method,” in Proceedings of the 1st International Conference on Formal Engineering Methods, 1997, pp. 81.

[49] M. Grindal, J. Offutt, and S. F. Andler, “Combination testing strategies: a survey,” *Software Testing, Verification and Reliability,* vol. 15, no. 3, pp. 167-199, 2005.

[50] C.-H. Liu, S.-L. Chen, and X.-Y. Li, “A WS-BPEL Based Structural Testing Approach for Web Service Compositions,” in Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering, 2008, pp. 135-141.

[51] J. Yan, L. Zhongjie, Y. Yuan, S. Wei, and Z. Jian: BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. *Prooceedings of the Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, 2006. 7-10 Nov. 2006;75-84

[52] D. Manova, S. Ilieva, F. Lonetti, A. Bertolino, and C. Bartolini, "Towards automated robustness testing of BPEL orchestrators," in Proceedings of the 12th International Conference on Computer Systems and Technologies, Vienna, Austria, 2011, pp. 659-664.

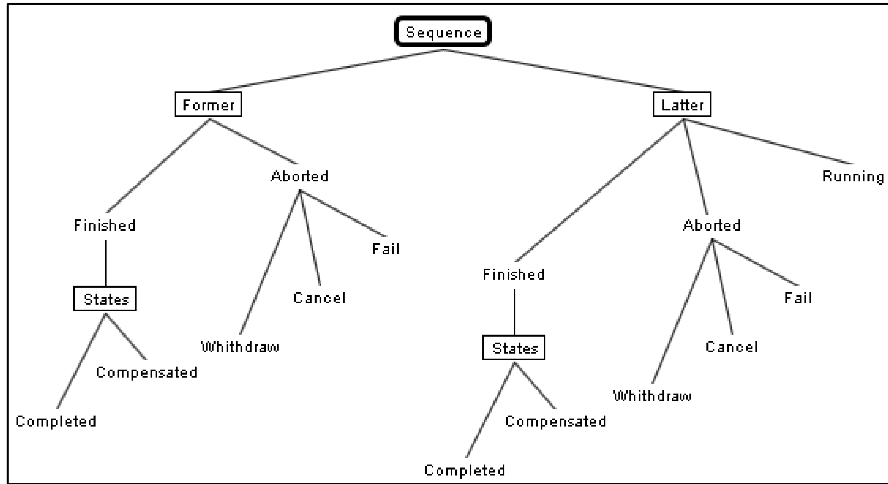
[53] T. Lertphumpanya, and T. Senivongse, "A basis path testing framework for WS-BPEL composite services," in Proceedings of the 7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, Cambridge, UK, 2008, pp. 107-112.

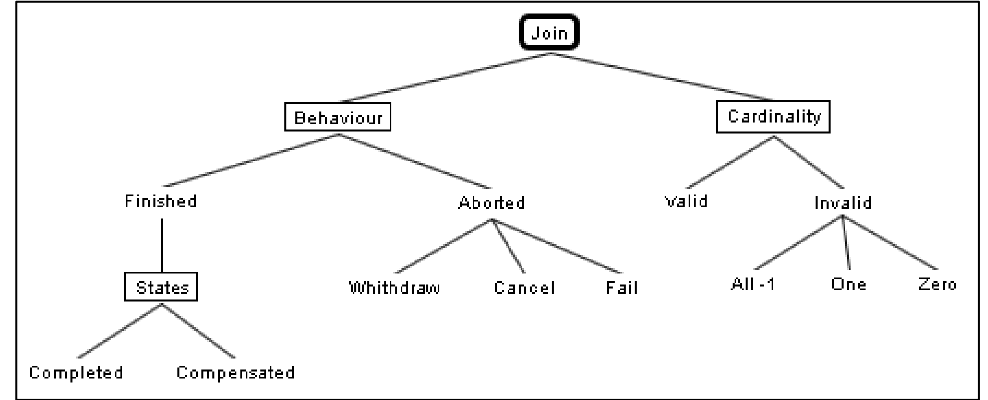
[54] J. Garcia-Fanjul, C. de la Riva, and J. Tuya: Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking. *Prooceedings of the Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, 2006. 29-31 Aug. 2006;127-130

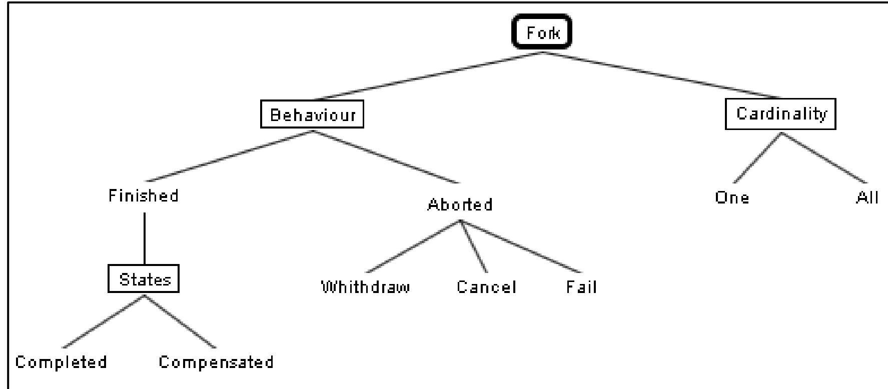
[55] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system: Research Articles," *Softw. Test. Verif. Reliab.,* vol. 15, no. 2, pp. 97-133, 2005.

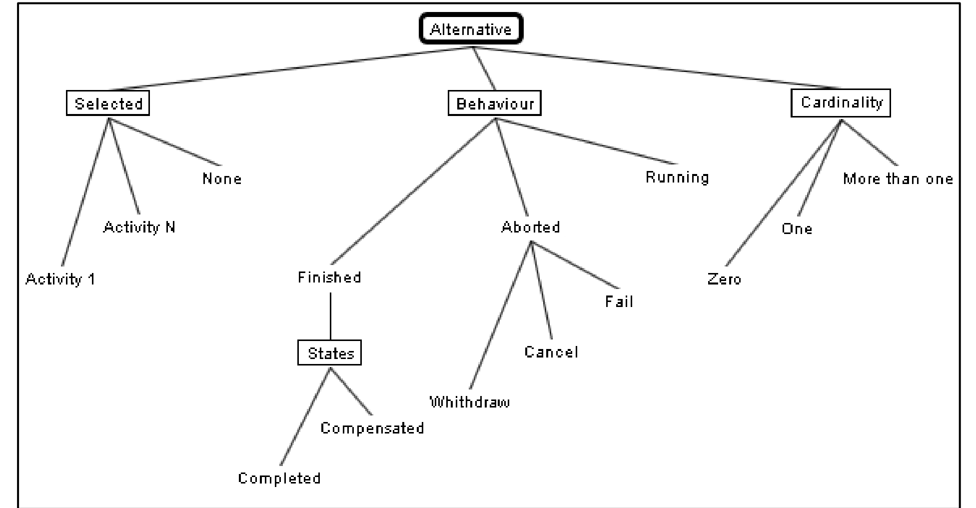
[56] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, 2005, pp. 402-411.

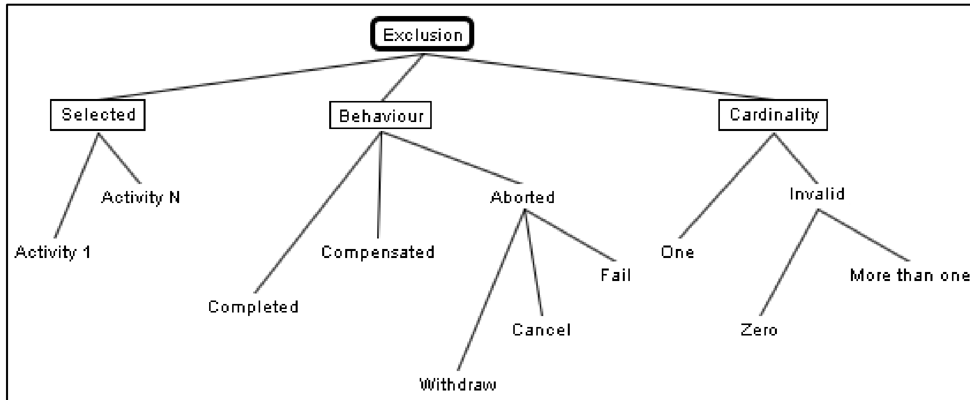
ANNEX I. Dependencies Classification-Trees

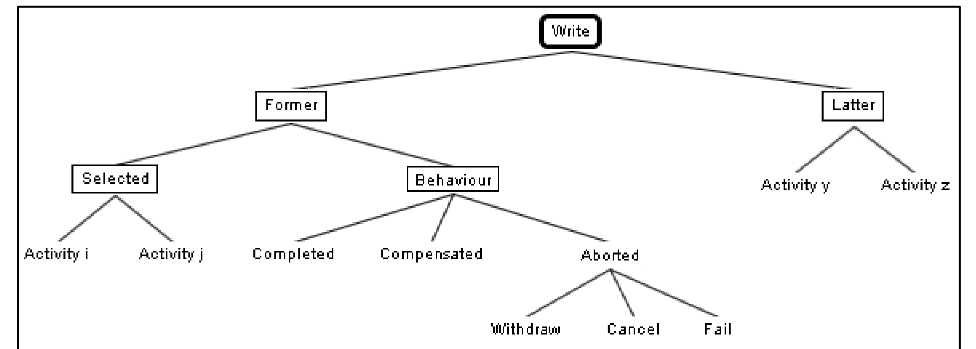












ANNEX II. Detailed results

	Total mutants	**Mutation score**				**Mutants killed**				**Mutants alive**				**Number of mutants**		**Traditional mutation score**				**Class mutation score**			
Class	Mutants	S/N	W/N	S/E	W/E	S/N	W/N	S/E	W/E	S/N	W/N	S/E	W/E	Traditional	Class	S/N	W/N	S/E	W/E	S/N	W/N	S/E	W/E
AgencyLogic	0	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-	-	-	-	-	-	-	-
AgencyWS	104	100,00	84,62	97,12	84,62	104	88	101	88	0	16	3	16	97	7	100	84	96	85	100	85	100	71
CarLogic	238	100,00	91,60	99,58	78,57	238	218	237	187	0	20	1	51	233	5	100	91	99	79	100	100	100	20
CarReservation	28	100,00	96,43	96,43	89,29	28	27	27	25	0	1	1	3	20	8	100	95	100	85	100	100	87	100
CarWS	173	100,00	97,11	98,84	91,91	173	168	171	159	0	5	2	14	156	17	100	99	99	92	100	76	94	88
CheapAirWS	125	100,00	92,00	96,80	81,60	125	115	121	102	0	10	4	23	116	9	100	92	96	82	100	88	100	66
Customer	78	100,00	82,05	96,15	23,08	78	64	75	18	0	14	3	60	59	19	100	79	94	30	100	89	100	0
FiveHotelWS	173	100,00	79,19	97,69	56,65	173	137	169	98	0	36	4	75	156	17	100	76	97	57	100	100	100	52
FlightLogic	387	99,74	73,64	99,22	61,24	386	285	384	237	1	102	3	150	366	21	99	74	99	63	100	61	95	38
FlightReservation	28	96,43	42,86	96,43	42,86	27	12	27	12	1	16	1	150	10	18	90	60	100	63	100	33	94	38
GoldAirWS	133	99,25	69,17	97,74	48,87	132	92	130	65	1	41	3	68	124	9	99	66	97	49	100	100	100	44
HotelLogic	231	100,00	88,74	100,00	80,09	231	205	231	185	0	26	0	46	226	5	100	89	100	80	100	40	100	60
HotelReservation	28	100,00	64,29	100,00	14,29	28	18	28	4	0	10	0	24	20	8	100	80	100	20	100	25	100	0
Itinerary	7	100,00	57,14	100,00	57,14	7	4	7	4	0	3	0	3	0	7	-	-	-	-	100	57	100	57
PaymentLogic	8	100,00	62,50	87,50	50,00	8	5	7	4	0	3	1	4	6	2	100	66	100	50	100	50	50	50
PaymentTransfer	10	100,00	10,00	100,00	10,00	10	1	10	1	0	9	0	9	8	2	100	12	100	12	100	0	100	0
PaymentWS	112	100,00	69,64	99,11	58,04	112	78	111	65	0	34	1	47	105	7	100	70	99	59	100	57	100	42
TAcontext	59	100,00	84,75	100,00	98,31	59	50	59	58	0	9	0	1	14	45	100	100	100	100	100	80	100	97
TAflow	217	100,00	98,16	100,00	97,70	217	213	217	212	0	4	0	5	211	6	100	99	100	100	100	33	100	33
TrainLogic	207	99,52	88,89	76,81	30,43	206	184	159	63	1	23	48	144	195	12	100	92	80	31	91	25	25	16
TrainReservation	28	100,00	42,86	39,29	14,29	28	12	11	4	0	16	17	24	10	18	100	60	80	60	100	33	16	0
TrainWS	133	100,00	71,43	63,16	45,11	133	95	84	60	0	38	49	73	124	9	100	71	63	45	100	66	55	44
TwoHotelWS	173	100,00	60,69	65,32	59,54	173	105	113	103	0	68	60	70	156	17	100	61	66	60	100	52	52	52
Total	2680	99,85	81,19	92,50	65,45	2676	2176	2479	1754	4	504	201	1060	2412	268	99,76	82,28	92,99	67,08	99,60	65,72	84,51	49,91