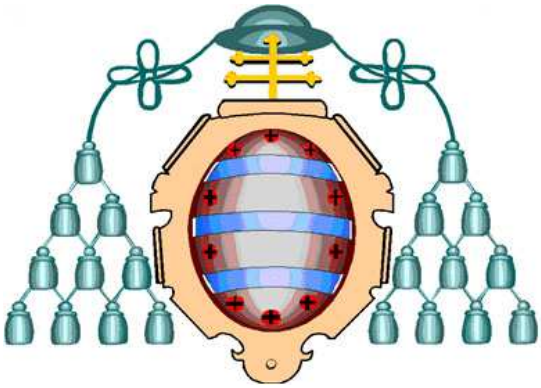


Universidad de Oviedo

Departamento de Informática

Sistemas y Servicios Informáticos para Internet

Testing Advanced Transactions in Service-based Software Systems

Pruebas de Transacciones Avanzadas en Sistemas Software basados en Servicios

Rubén Casado Tejedor

Enero, 2013

UNIVERSIDAD DE OVIEDO
Vicerrectorado de Internacionalización y Postgrado

CENTRO INTERNACIONAL DE POSTGRADO
CAMPUS DE EXCELENCIA INTERNACIONAL

RESUMEN DEL CONTENIDO DE TESIS DOCTORAL

1.- Título de la Tesis	
Español/Otro Idioma: Pruebas de Transacciones Avanzadas en Sistemas Software basados en Servicios	Inglés: Testing Advanced Transactions in Service-based Software Systems

2.- Autor	
Nombre: Rubén Casado Tejedor	
Programa de Doctorado: Sistemas y Servicios para Internet	
Órgano responsable: Departamento de Informática	

RESUMEN (en español)

Las Arquitecturas Orientadas a Servicios (en inglés SOA), y su implementación como Servicios Web (en inglés WS), definen un nuevo paradigma computacional en donde los requisitos funcionales y no funcionales de servicios especializados se publican en Internet. De esta manera pueden ser dinámicamente descubiertos y compuestos con el objetivo de crear composiciones de servicios que proporcionen una funcionalidad mejorada. Las transacciones en Servicios Web (en inglés, WS transactions), son el mecanismo utilizado por los servicios para asegurar una ejecución fiable manteniendo la consistencia de la información. El modelo clásico ACID es inadecuado para el manejo de las WS transactions debido a la naturaleza distribuida y desacoplada de este tipo de procesos. Por ello se han propuesto numerosos modelos y protocolos para afrontar los nuevos retos que arrojan las WS transactions. Esas alternativas tratan de mejorar la calidad de las WS transactions en términos de eficiencia en tiempo de respuesta, recuperación de fallos, flexibilidad y soporte para complejas aplicaciones de negocio de larga duración.

Esta Tesis Doctoral aborda otra dimensión relativa a la calidad: el proceso de pruebas (en inglés testing) de las WS transactions. Nuestro objetivo es detectar posibles fallos en los sistemas software que se basan en WS transactions. Para ello presentamos el Marco de Trabajo para Prueba de Transacciones (el acrónimo inglés F2T). F2T ha sido diseñado y desarrollado especialmente para WS transactions. F2T se ideó para organizar todos los conceptos que participan en el proceso de diseño de casos de prueba. Dada la variedad de protocolos y estándares existentes para manejar WS transactions, esta tesis también presenta el Modelo Abstracto de Transacciones (el acrónimo en inglés AbTM) el cual es capaz de modelar el funcionamiento de las transacciones independientemente del protocolo utilizado. F2T utiliza el AbTM para el diseño de las pruebas. Como parte del desarrollo de F2T, hemos propuesto un conjunto de técnicas y criterios de prueba para probar el comportamiento aislado de los participantes así como las dependencias (relaciones) existentes entre ellos. Estas contribuciones han sido evaluadas utilizando rigurosos experimentos que constatan la eficacia y eficiencia de nuestra propuesta. La evaluación que se ha llevado a cabo incluye un caso de estudio real de una aplicación transaccional bancaria así como un profundo análisis de mutación del recurrente caso de estudio de una agencia de viajes web.

RESUMEN (en Inglés)

Service Oriented Architectures (SOA), and its implementation as Web services (WS), provide a new computing paradigm in which functional and non-functional requirements of specialised services are published over the Internet such that they can be dynamically discovered and composed in order to create composite services that provide integrated and enhanced functionality. Web services transactions are used to ensure reliable execution of services and to maintain the consistency of data. The classical ACID model has been shown unsuitable for WS environments due to the loosely coupled and distributed nature of the process. Numerous models and protocols have been developed to deal with the new challengers of WS transactions. These aim to improve the quality of WS transactions in terms of response time efficiency, failure recovery, flexibility and support for long running and complex business applications.

This thesis focuses on another quality dimension which is the testing of WS transactions. In it, the focus of testing is to detect possible faults or failures in software systems that rely on WS transactions. To that purpose, we present the Framework for Testing Transactions (F2T) which has been designed and developed for testing WS transactions. F2T has been devised to organize all the concepts involving in the process of test case design. Due to the variety of protocols and standards currently used to manage WS transactions, this thesis also presents the Abstract Transaction Model (AbTM) which is capable of modelling the transaction behaviour independently of the protocol used. F2T uses AbTM for the testing purposes. As part of the F2T development, a set of test techniques and criteria have been defined to test the isolated behaviour of the services and their dependencies (relationships) in WS transactions. These contributions have been evaluated using rigorous experiments that reveal the efficacy and efficiency of the proposed work. The evaluation includes a real industrial case study of a transactional bank application and an in-depth mutation analysis of the well-know web travel agency case study

SR. DIRECTOR DE DEPARTAMENTO DE INFORMÁTICA

Agradecimientos

Este es uno de los momentos más importantes al escribir la tesis. El trabajo de investigación ha sido largo por lo que "mi gente" ha tenido que compartir conmigo buenas e inolvidables situaciones, pero también momentos difíciles y estresantes. Es para mí un placer dedicar unas líneas a esa gente para agradecérselo. Esa gente ha hecho posible esta tesis y les debo mi más profunda gratitud.

Primero y más destacado, me gustaría dar las gracias a mis directores Javier Tuya y Muhammad Younas. Esta tesis no habría sido posible sin sus inestimables consejos y dirección. Gracias Javier por todas las veces que tuviste que explicarme qué es testing, cuál es la diferencia entre defecto y fallo así como las valiosas revisiones que hiciste de mis trabajos. Echaré de menos las veces que me recordabas que tenía que publicar. Gracias Younas por compartir conmigo tu incalculable conocimiento sobre las transacciones así como por tu valiosa ayuda para publicar mis trabajos. Pero gracias también por hacer mis estancias en Oxford tan cómodas. Me he sentido como si viviera y trabajara en mi propia ciudad.

Quiero también dar las gracias a otra gente que me ha ayudado durante mi periodo de investigación, especialmente a mis compañeros del Grupo de Investigación en Ingeniería del Software de la Universidad de Oviedo. Quiero también agradecer a la gente de Oxford Brookes University (Reino Unido) así como a la gente del equipo de investigación del Profesor Claude Godart en INRIA-LORIA (Nancy, Francia).

Finalmente y sin duda más importante, tengo que dar las gracias a mi familia y amigos. Gracias de verdad por vuestro apoyo. Gracias Olai por querer siempre jugar con tu tío viajero. Gracias también a su mamá porque ha criado a mi sobrino maravillosamente. Gracias Violeta por todo este tiempo juntos. Esta tesis está finalizando, pero nuestro tiempo acaba de empezar. Las gracias más importantes son para mi mamá, mi papá y mi hermano. Gracias de verdad por todo, habéis hecho que mi vida sea muy fácil. Gracias por creer en mí incluso cuando no entendíais que es lo que hacía. Espero que algún día podáis estar orgullosos de mi de la misma manera que yo ya lo estoy de vosotros.

Abstract

Service Oriented Architectures (SOA), and its implementation as Web services (WS), provide a new computing paradigm in which functional and non-functional requirements of specialised services are published over the Internet such that they can be dynamically discovered and composed in order to create composite services that provide integrated and enhanced functionality. Web services transactions are used to ensure reliable execution of services and to maintain the consistency of data. The classical ACID model has been shown unsuitable for WS environments due to the loosely coupled and distributed nature of the process. Numerous models and protocols have been developed to deal with the new challengers of WS transactions. These aim to improve the quality of WS transactions in terms of response time efficiency, failure recovery, flexibility and support for long running and complex business applications.

This thesis focuses on another quality dimension which is the testing of WS transactions. In it, the focus of testing is to detect possible faults or failures in software systems that rely on WS transactions. To that purpose, we present the Framework for Testing Transactions (F2T) which has been designed and developed for testing WS transactions. F2T has been devised to organize all the concepts involving in the process of test case design. Due to the variety of protocols and standards currently used to manage WS transactions, this thesis also presents the Abstract Transaction Model (AbTM) which is capable of modelling the transaction behaviour independently of the protocol used. F2T uses AbTM for the testing purposes. As part of the F2T development, a set of test techniques and criteria have been defined to test the isolated behaviour of the services and their dependencies (relationships) in WS transactions. These contributions have been evaluated using rigorous experiments that reveal the efficacy and efficiency of the proposed work. The evaluation includes a real industrial case study of a transactional bank application and an in-depth mutation analysis of the well-know web travel agency case study.

Resumen

Las Arquitecturas Orientadas a Servicios (en inglés SOA), y su implementación como Servicios Web (en inglés WS), definen un nuevo paradigma computacional en donde los requisitos funcionales y no funcionales de servicios especializados se publican en Internet. De esta manera pueden ser dinámicamente descubiertos y compuestos con el objetivo de crear composiciones de servicios que proporcionen una funcionalidad mejorada. Las transacciones en Servicios Web (en inglés, WS transactions), son el mecanismo utilizado por los servicios para asegurar una ejecución fiable manteniendo la consistencia de la información. El modelo clásico ACID es inadecuado para el manejo de las WS transactions debido a la naturaleza distribuida y desacoplada de este tipo de procesos. Por ello se han propuesto numerosos modelos y protocolos para afrontar los nuevos retos que arrojan las WS transactions. Esas alternativas tratan de mejorar la calidad de las WS transactions en términos de eficiencia en tiempo de respuesta, recuperación de fallos, flexibilidad y soporte para complejas aplicaciones de negocio de larga duración.

Esta Tesis Doctoral aborda otra dimensión relativa a la calidad: el proceso de pruebas (en inglés testing) de las WS transactions. Nuestro objetivo es detectar posibles fallos en los sistemas software que se basan en WS transactions. Para ello presentamos el Marco de Trabajo para Prueba de Transacciones (el acrónimo inglés F2T). F2T ha sido diseñado y desarrollado especialmente para WS transactions. F2T se ideó para organizar todos los conceptos que participan en el proceso de diseño de casos de prueba. Dada la variedad de protocolos y estándares existentes para manejar WS transactions, esta tesis también presenta el Modelo Abstracto de Transacciones (el acrónimo en inglés AbTM) el cual es capaz de modelar el funcionamiento de las transacciones independientemente del protocolo utilizado. F2T utiliza el AbTM para el diseño de las pruebas. Como parte del desarrollo de F2T, hemos propuesto un conjunto de técnicas y criterios de prueba para probar el comportamiento aislado de los participantes así como las dependencias (relaciones) existentes entre ellos. Estas contribuciones han sido evaluadas utilizando rigurosos experimentos que constatan la eficacia y eficiencia de nuestra propuesta. La evaluación que se ha llevado a cabo incluye un caso de estudio real de una aplicación transaccional bancaria así como un profundo análisis de mutación del recurrente caso de estudio de una agencia de viajes web.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

No tale is so good that it can´t be spoiled in the telling.

Proverb

This chapter introduces the context of the research work and presents the proposed research hypothesis and goals. The main contributions and research outcomes are also summarized. The chapter outlines the fulfilment of the requirements for obtaining the qualification of International Doctor. Finally, the chapter describes the structure of the thesis.

1.1. Context

Transaction processing constitutes a key component of most modern software systems which are required to fulfil Quality of Service (QoS) requirements such as reliability, integrity, consistency and efficiency. Transactions are based on the principle of (semantic) atomicity which ensures "all or nothing" operation of software applications. Thus transactions enable software systems to remain in consistent state despite failures of communication systems and/or computer systems. Further, transactions are executed concurrently thus ensuring the efficiency of software applications.

The creation of complex software systems as a composition of simpler, heterogeneous, and possibly distributed parts have been always present in the different areas of computer science. The need of integrate dynamic, independent and distributed components of software, has been addressed through the concept of Service Oriented Architecture (SOA). Web Services (WS) is the most widely accepted and used implementation of SOA [1].

Rapidly evolving WS technologies facilitate the development of applications that enable collaboration and integration between different businesses. Such applications rely on complex interactions that involve many parties, span many different organisations, and can have long duration. Transactions used in such scenarios differ significantly from those used in classical systems which were closed and homogenous systems. In WS environment, transactions involve loosely-coupled parties, often from different administrative domains. Thus, they require commitments to be negotiated at runtime and isolation levels to be relaxed [2]. The theme of this dissertation is to test such transactions in WS environment proposing suitable software testing techniques.

Software testing is the process of finding unexpected behaviours in software systems. Software testing plays a key role in the development of software systems in order to evaluate whether the application meets its functional as well as non-functional (QoS related) requirements. The process

of testing transactions is a key issue in order to ensure the reliability of the service oriented software systems.

1.2. Research hypothesis

This thesis sets the following research hypothesis:

A Web Services transaction represents a composition of services but with special properties and requirements such as preservation of atomicity and maintenance of consistency. By analyzing the transactional requirements, we claim that a set of good test cases can be systematically achieved. These generated test cases cannot be obtained using the current testing techniques developed for web services compositions.

1.3. Research aims and objectives

The main goal of this thesis is to devise a framework to organize all the concepts involved in the process of test case design for WS transactions. The framework shall include the knowledge about the existing transaction models and standards, a hierarchical organization of the aspects to test, as well as the definition of new testing techniques to address the identified test requirements. The application of the proposed testing techniques shall provide the tester a systematic method to define different test suites according to the features tester wants to focus on about the WS transaction. The research objectives are detailed as follow:

- To identify the different roles that the services play during the transaction processing.
- To model the behaviour of the identified roles from a testing point of view.
- To define the possible dependencies between services involved in a transaction.

- By using the previously identified roles models and dependencies, to define an abstract transaction model capable of modelling the execution of a service based transaction independently of the underlying protocol.
- To identify the different properties that shall be taking into account in the transaction testing process.
- To analyze those properties in order to identify the test requirements.
- To organize the testing concepts involved in the addressing of such test requirements.
- To propose new test techniques to address the identified test requirements.
- To define systematic test methods for transaction-based applications using the abstract transaction model and the proposed test techniques.
- To evaluate the test cases generated for the proposed methods in real software applications.

1.4. Research outcomes

This section outlines the research outcomes and the potential contributions of the proposed work in this thesis.

1.4.1. Contributions

The potential contributions of this thesis are summarized as follow:

- Definition of the Abstract Transaction Model (AbTM) which is capable of representing the existing transaction standards and protocols. This generic model allows focusing the test methods on a single model rather than defining specific technique for each protocol. A test suite generated for the AbTM can be automatically translated to any of the existing transaction standards. Furthermore, as the test model is different from the implementation one, the AbTM allows applying a black-box testing approach.

- Design, development and implementation of the Framework for Testing Transactions (F2T). This novel approach identifies the relevant properties, attributes and dimensions of a WS transaction from a testing point of view. F2T allows the identification of different set of test requirements and helps in the definition of new test criteria for WS transactions.

- Design, implementation and validation of a testing method for checking the isolated behaviour of the services involved in a WS transaction. The method uses the structure elements of the AbTM to achieve the set of abstract test cases. Each abstract test case is automatically translated to a concrete test case composed by the test scenario and the expected system outcome according to the WS transaction standard used in the software under test. Thus, the proposed method allows the automation of the test design and test outcome evaluation phases.

- Design, implementation and validation of testing methods for checking the dependencies (relationships) between the services of a WS transaction. Two different approaches have been proposed. The *control-flow based* approach defines the dependencies in terms of task relationships. Using the logical expression derived of those relationships, we propose a novel family of test criteria to exercise the implementation of the flow dependencies. The *Classification-Tree based* approach identifies, analyzes, and classifies the possible dependencies between services. We use the Classification-Tree technique to derive the test requirements. A family of novel test criteria are proposed. Such criteria allow the tester to adjust the proposed test method in terms of effectiveness, test effort and cost-benefit analysis.

1.4.2. Publications

This section presents a complete list of publications which are the outcomes of this research work. An acronym represents each publication in the Figure 1.1. It classifies our contributions according to the year of publication (vertically) and the topic they address (horizontally). We have used three topics: *Framework* refers to the organization of the concepts

involved in testing WS transaction. *Unit testing* refers to the techniques proposed to test the behavior of the services. *Integration testing* refers to the techniques proposed to test the dependencies between the services. Note that we use *unit* and *integration* topics just as *conspiratorial wink* to the classic software testing levels. For each publication a geometric shape is presented together with the acronym. Circle represents a publication in a journal indexed in the Journal Citation Reports® (JCR) [3]. Square represent a publication in an international conference ranked in the ERA Conference Ranking Exercise (CORE) [4] and Microsoft Academic Research ranking [5] or a published book chapter. Finally, triangle represents a publication in a workshop, national conference or other journals. As illustrated, the graph shows a clear definition of what has been our research path until now.

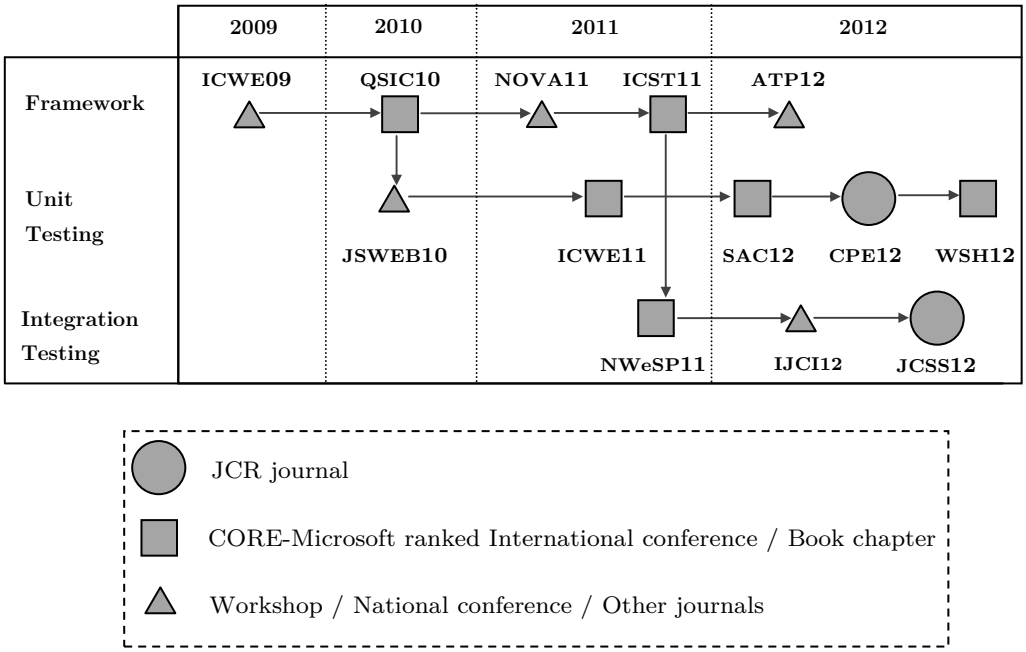


Figure 1.1. Summary of publications

- [ICWE09] Rubén Casado, Javier Tuya. **Testing Transactions in Service Oriented Architectures**. *International Conference on Web Engineering, (ICWE) Doctoral Consortium*. San Sebastian, June 2009

- [QSIC10] Rubén Casado, Javier Tuya, Muhammad Younas. **Testing Long-lived Web Services Transactions Using a Risk-based**

Approach. Proceedings of the *10th International Conference on Quality Software (QSIC)*. Zhangjiajie, China, July 2010.

- [JSWEB10] Rubén Casado, Javier Tuya, Muhammad Younas. **Specifying and testing recoverability requirements in WS-BusinessActivity transactions**. *Actas de las VI Jornadas Científico-Técnicas en Servicios Web y SO*A, 2010

- [ICST11] Rubén Casado, Javier Tuya, Muhammad Younas. **A Framework to Test Advanced Web Services Transactions**. Proceedings of *IEEE 4th International Conference on Software Testing, Verification and Validation (ICST).* Berlin, March 2011.

- [ICWE11] Rubén Casado, Javier Tuya, Muhammad Younas. **An Abstract Transaction Model for Testing the Web Services Transactions**. Proceedings of the *9th IEEE International Conference on Web Services (ICWS).* Washington DC, July 2011

- [NOVA11] Rubén Casado, Javier Tuya, Muhammad Younas. **Especificación y prueba de requisitos de recuperabilidad en transacciones WS-BusinessActivity**. *Revista de la Asociación de Técnicos en Informática (NOVATICA)*, 37 (211), pp 61-65, May 2011

- [NWeSP11] Rubén Casado, Javier Tuya, Claude Godart. **Dependency-based Criteria for Testing Web Services Transactional Workflows**. Proceedings of the *7th IEEE International Conference on Next Generation Web Services Practices (NWeSP).* Salamanca, October 2011.

- [SAC12] Rubén Casado, Javier Tuya, Muhammad Younas. **Testing the Reliability of Web Services Transactions in Cooperative Applications**. Proceedings of the *27th Symposium of Applied Computing (SAC). Trento, Italy,* March 2012.

- [CPE12] Rubén Casado, Javier Tuya, Muhammad Younas. **Evaluating the effectiveness of the abstract transaction model in testing**

web services transactions. *Concurrency and Computation - Practice and Experience,* (in press) May 2012.

- [IJCI12] Rubén Casado, Javier Tuya, Claude Godart , Muhammad Younas. **Test case design for transactional flows using a dependency-based approach**. *International Journal of Computer Information Systems and Industrial Management Applications,* 5 20-40, 2012.

- [ATP12] Rubén Casado, Javier Tuya, Muhammad Younas. **A Family of Test Criteria for Web Services Transactions**. Proceedings of the *International Symposium on Advances in Transaction Processing*. Niagara Falls, Ontario, August 2012.

- [WSH12] Rubén Casado, Muhammad Younas, Javier Tuya. **A Generic Framework for Testing the Web Services Transactions**. Chapter in *Web Services Handbook*, to be published, 2012.

- [JCSS12] Rubén Casado, Muhammad Younas, Javier Tuya. **Multi-dimensional Criteria for Testing Web Services Transactions**. *Journal of Computer and Systems Sciences*, (in press) 2012

1.4.3. Visits

During the preparation of this thesis, I have visited the Department of Computing and Communication Technologies of Oxford Brookes University (Oxford, UK) and the Services and Cooperation Research team (SCORE), a mixed research group from *Laboratorie lorrain de Researche en Informatique et ses Applications* (LORIA) and *Institut National de Recherche en Informatique et en Automatique* (INRIA) at Nancy, France. In Oxford, I worked closely with Dr. Muhammad Younas, one of the world lead researchers in the web services transactions area. In Nancy, I collaborated with Professor Claude Godart who has a strong background of publications about web services transaction validation and verification. Table 1.1 summarizes the visits information.

Year	Duration	Place	Goal
2010	6 months	Oxford, UK	Definition of the Abstract Transaction Model
2011	3 months	Nancy, France	Definition of the dependencies between services
2011	2 months	Oxford, UK	Development of the Abstract Transaction Model
2012	4 months	Oxford, UK	Validation of the proposed test techniques

Table 1.1. Summary of research visits

1.5. International thesis

With this thesis, we aim to obtain the qualification of International Doctor. The requirements, defined in the Spanish Royal Decree 99/2011 [6], are the following:

1. During the research period, the student has completed a minimum stay of three months outside Spain in a higher education institution in another country studying or doing research. This stay must be acknowledged by the thesis director and must be certified by the manager of the research group of the institution where the student completed this stay.

2. At least part of the thesis, as well as the abstract and conclusions must be written and submitted in one of the languages commonly used for scientific communication in the area of knowledge in question. In the Computer Science area, the most widely accepted language is English.

3. At least the abstract and conclusions must be written in one of the official language of Spain.

4. The thesis has been informed by a minimum of two experts from a higher education institution or research institution from another country than Spain.

5. The Board of Examiners is comprised by at least one PhD holder who is an expert on the field and belongs to a foreign higher education institution or research institute different from the ones mentioned in requirements 1 and 4.

We have met the above requirements during the completion of this thesis. The requirement 1 is widely met with the certified research visits (see Section 1.4.3) at Oxford Brookes University and INRIA/LORIA. This document is written in English (meeting requirement 2) but also the Abstract and Conclusions sections are written in Spanish too (meeting requirement 3). According to the requirement 4, the document will be sent to Professor Makoto Takizawa (Seikei University, Japan) and Dr. David Taniar (Monash University, Australia), to obtain their evaluation reports of the thesis. To meet the requirement 5, Professor Irfan Awan (University of Bradford, UK) will be invited to the Board of Examiners.

1.6. Thesis structure

This remainder of the document is structured as follows:

Chapter 2 surveys the current work on transaction processing including classical models as well as the current WS transaction standards. The second part of the chapter reviews the works on testing web services as well as the existing approaches about transaction verification. After the literature review, a set of open issues are identified.

Chapter 3 presents the core of this thesis. The first part presents the Abstract Transaction Model which is capable of modelling different WS transaction protocols from a testing point of view. The second part of the chapter presents the Framework for Testing Transaction which has been devised to organize all the concepts involving in the process of test case design.

Chapter 4 addresses in depth the issue of testing transaction participants as defined in the framework (developed in Chapter 3). This chapter focuses on the isolated behavior of the executors and presents an

automatic method to generate test case for different WS transaction standards.

Chapter 5 addresses the issue of dependencies between participants of a WS transaction. This chapter proposes a control-flow based technique to test such relationships. The chapter also presents the validation of the proposed approach through an industrial case study.

Chapter 6 addresses the same issue as that of Chapter 5 but proposing a different testing technique. In this chapter, the approach is based on the systematic analysis of the possible situations. A set of test criteria are presented and evaluated with a case study.

Chapter 7 presents the main conclusions of our research works and reports our plans for future research.

Chapter 8 (*Capítulo 8*) presents the conclusions and future research work in Spanish.

1.7. Summary in a picture

According to the famous adage "*a picture is worth a thousand words*", Figure **1**.2 depicts a graphic summary of the thesis in form of word cloud.

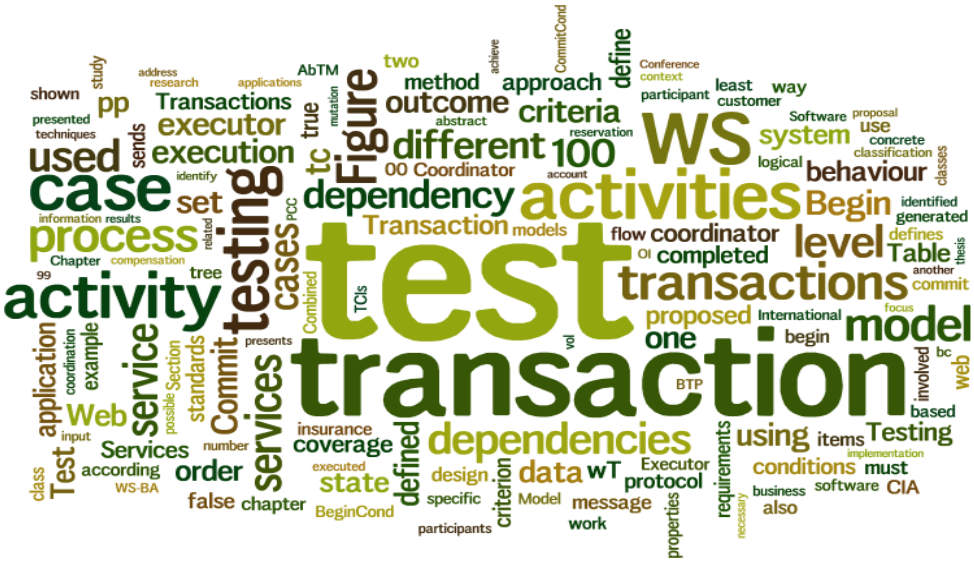


Figure 1.2. Thesis word cloud

Chapter 2

Background and research review

You have to know the past to understand the present

Carl Sagan

This chapter reviews the main concepts involved in this thesis: transactions and testing in the Service Oriented Architectures (SOA), especially in its implementation as Web Services (WS). The first part explains the evolution of the transaction model from the ACID properties to the current WS transaction standards. It critically analyzes the Advanced Transaction Models (ATM) in relation to the requirements and characteristics of WS environment. The second part analyzes the existing efforts in testing web services as well as the works on verification and validation of transactions.

2.1. Service Oriented Architecture

This section outlines the concept of *services* and the associated architectural paradigm, Service Oriented Architecture (SOA). It focuses on the transactional management of the service-based applications.

SOA is a style of software architecture that is modular, distributed and loosely coupled. SOA-style applications use business components that are designed to be reusable across applications and enterprise boundaries. These components are invoked through services that are based on well-defined interface definitions, and are independent of the underlying hardware and software platforms, as well as the development language [7].

According to [8], the definition of SOA can be perceived differently by different people:

- *Business executive and business analyst*: a set of services that constitutes information technology (IT) assets (capabilities) and can be used for building solutions and exposing them to customers and partners.
- *Enterprise architect*: a set of architectural principles and patterns addressing overall characteristics of solutions: modularity, encapsulation, loose coupling, separation of concerns, reuse, composability, and so on.
- *Project manager*: A development approach supporting massive parallel development.
- *Tester or quality assurance engineer:* a way to modularize, and consequently simplify, overall system testing.
- *Software developer:* a programming model complete with standards, tools, and technologies, such as Web Services.

Independently of the perspective used, the kernel of SOA is the services. According to the OASIS SOA RM [9], a service is:

A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.

As an architectural concept, SOA permits multiple approaches for the realization and deployment of an IT system that has been designed and built around its service-based principles. This thesis focuses on a specific technology that, arguably, has the most significant academic and industrial visibility and attraction, and that is Web Services (WS) [10].

2.1.1. Web Services

The World Wide Web Consortium(W3C) [11], which has managed the evolution of some of the most influential standards such as SOAP [12] and WSDL [13] specifications, defines Web Services (WS) as follows [14]:

A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with XML serialization in conjunction with other Web-related standards.

A key aspect of WS is the use of an open, standards-based approach in which every WS specification is eventually standardized by an industry-wise organization (such as W3C [11] or OASIS [15]). The WS community has done significant work to address this interoperability issue, and since the introduction of the first WS, various organizations have introduced other WS-related specifications. Figure 2.1 illustrates the most current and widely accepted standards, forming the WS standard stack [10].

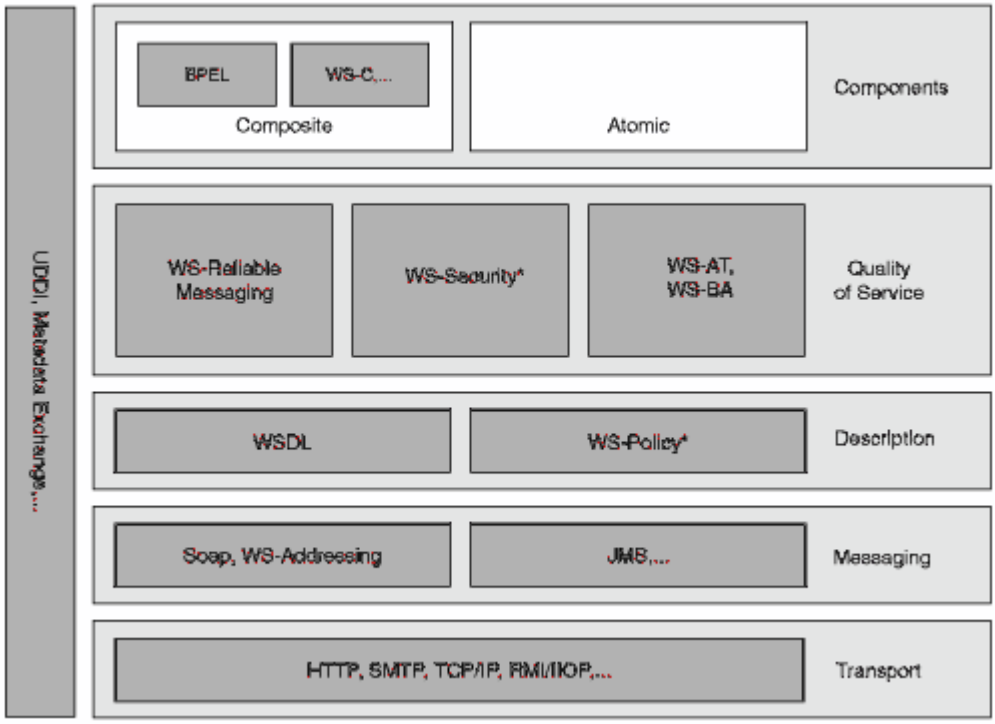


Figure 2.1. WS standards stack

Modern and complex business scenarios necessitate the development of applications that consist of multiple WS in order to create composite services that can perform efficient and cost effective functions. However, it is important to ensure that such applications are correctly executed and the underlying data is consistent and correct even in the case of failures of component services or communication failures. A coordinated orchestration of the outcome of the participating services that make up the business application is essential so that a coherent outcome of the whole business application can be agreed upon and guaranteed. Transactions have commonly been used in order to meet such requirements as correctness, consistency, and coordinated orchestration of complex business applications.

2.2. Transactions in SOA

In order to ensure reliable execution of WS applications it is crucial that their activities are modelled as transactions such that they achieve a mutually agreed outcome. 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 since their execution requires careful coordination, accounting for fault-tolerance, correct process termination and cancelation, without undesirable consequences at any stage of the execution.

This section outlines the evolution transaction models and explains why the classical models are not suitable for the WS environment. A depth analysis of the history of transactional management can be consulted in [16].

2.2.1. Classical transaction models

Transactions are a fundamental concept in building reliable distributed applications. A transaction is a mechanism to ensure that all the participants in an application achieve a mutually agreed outcome [17, 18].

Often, we mnemonically refer to the collection of reliability guarantees for transactions as the ACID properties:

- *Atomicity*: either the all operations of a transaction are successfully executed, or none are.
- *Consistency*: a transaction must bring the system from one consistent state to another.
- *Isolation*: the effects of the operations are not visible outside the transaction´s scope until it completes successfully. Each transaction appears as if it executes in isolation.
- *Durability*: once a transaction has successfully completed, the changes it has made to the data are made permanent in order to survive failures.

In order to ensure the ACID properties, distributed transaction processing systems use the Two Phase Commit protocol (2PC) [19] for all parties to reach an agreement on the global outcome of a transaction. In the first preparation phase, the participants are asked to vote for the transaction to be committed or aborted. Those who vote to commit log their updates to

a reliable backup medium. An actual commit is not carried out until the second, completion phase, when a joint commit decision has been reached. Some variations of 2PC have been developed. For example the presumed abort (PA) [20] or presumed commit (PC) [21] take an assumption about the outcome of transactions and it allow optimizing the protocol. In PA when a coordinator decides to abort a transaction, it does not have to log its decision. It just sends abort messages to all the participants that have voted positively and discards all information about the transaction. That is, the coordinator does not wait for acknowledgments and therefore, the participants are not required to log such decisions. After a coordinator or participant failure, if the participant inquires about the transaction, the coordinator, not remembering the transaction, will direct the participant to abort it (by presumption). As opposed to PA, the Presumed Commit (PC) aims to reduce the cost of committing transactions. Instead of interpreting missing information about transactions as abort decisions, in PC coordinators interpret such information as commit decisions.

ACID transactions have proven to be very useful in traditional database applications where the execution time is relatively short, the number of concurrent transactions is relatively small and transactions execute on only one database system. However, they lack the flexibility to meet the requirements of complex applications such as WS-based applications where the workflow system needs to support long-living transactions [16]. Advanced Transaction Models (ATM) [22] have been used to deal with such needs.

2.2.2. Advanced Transaction Models

WS transactions are based on various models ranging from classical ACID models to Advanced Transaction Models (ATM). Two Phase Commit (2PC) protocol and its variants [19] 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 [23].

The distributed Two Phase Commit protocol (2PC) is a well-formed ACID protocol. While the strict ACID behaviour is desirable in a database environment or a short-lived distributed transaction, it can be too expensive to secure in a long-running distributed transaction environment. 2PC protocol cannot be completely applied in some distributed transactions where remote entities may interact by performing complex activities that may take longer processing. Such increased processing time results in unnecessary locking of resources thus, making roll-back activities impossible. Advanced (or Extended) Transaction Models (ATM) [22] have been designed in order to relax some of ACID properties and to meet the requirements of long running transactions.

The fundamental logic of ATM is to divide a transaction into smaller activities according to the semantics of the applications. The advanced transactions can perform more complex and longer-lasting tasks.

The Nested transaction model [24] was the first model using the idea to decompose a transaction into activities and allow them to commit independently. In this model a transaction is decomposed into a hierarchy of cooperating activities (called subtransactions) forming a transaction tree. A child transaction may start after its parent has started and a parent transaction may terminate only after all its children terminate. If a parent transaction is aborted, all its children are aborted. These strategies are applied recursively throughout the transaction tree.

The SAGAS model [25] introduced the concept of compensation in managing long-lived transactions. In this model a transaction is decomposed into independent activities and each activity has associated a compensation activity that semantically undoes the effects of committed activity. The compensation-based transactions allow managing consistency of data across applications without locking resources.

Both nested transaction and SAGAS models provided the basis for developing various other models such as open-nested [26] , split-join [27], ConTracts [28], Flex [29], WebTram [30] and so on. Further analysis of

ATM is presented in [30-32]. Some of such ATM based models have been adopted in the WS environment.

2.2.3. Web Services Transactions

In a WS environment, transactional applications are constructed from the composition of one or more services, each of which might manipulate shared data and be party to an agreed overall coordinated outcome. However, in a WS environment, the services that are the component parts of an application are typically loosely coupled and distributed across various independent systems spanning a network [33]. Therefore, in some scenarios, the ACID properties might have to be applied less strictly in order to allow more flexible forms of outcome coordination processing. It is necessary more relax forms of transaction to accommodate collaborations, workflow, real-time processing, and so on. Consequently, the Advanced Transaction Models (ATM) are more suitable for the WS environment.

With the shift in interest toward Internet-based applications, web service transactions have received growing attention from both industry and academia.

Industrial research

Three competing standardization efforts have been taken in this area, which are described below. This thesis does not want to develop a new transaction protocol, but we need to understand the behaviour of the existing standards in order to develop suitable testing methods for them. For this reason we do not present in-depth comparison of the standards. Further comparison between the three WS transaction standards can be consulted in [16, 34, 35].

The Business Transaction Protocol (BTP) [36] was proposed in 2001 by a consortium of companies including Hewlett-Packard, Oracle and BEA. It is an XML based protocol for representing and seamlessly managing complex, multi-step business-to-business (B2B) transactions over the Internet. BTP ensures consistent outcomes for parties that use applications that are disparate in time, location and administration and participate in

long running business transactions. Although BTP was not exclusively designed for WS, it has been widely used in such environment.

To ensure atomicity between multiple participants, BTP uses the 2PC protocol: during the first phase (*prepare*), an individual participant must make durable any state changes that occurred within the scope of the transaction, such that those changes can either be undone (*cancelled*) or made durable (*confirmed*) later once consensus has been achieved. Although BTP uses a 2PC protocol, it does not imply ACID semantics. The specific implementation of the *prepare*, *confirm* and *cancel* phases depends on the application business logic. The management of consistency and isolation issues of data are also back-end choices and not imposed or assumed by BTP [37].

Because the traditional 2PC protocol does not impose any restrictions on the time between executing the first and second phases, BTP took the approach of using this to allow business-logic decisions to be inserted between the phases. This means that users have to drive the two phases explicitly in what BTP terms an open-top completion protocol. The application has complete control over when transactions prepare, and using whatever business logic is required, later determine which transactions to confirm or cancel. *Prepare* becomes part of the service business logic, for example.

BTP specifies two extended business transactions types:

- *Atoms*. If the transaction is configured as an atom, it is guaranteed that the transaction outcome of all the involved activities is atomic, meaning that either all participants confirm or all participants cancel. That is, Atoms meet the conditions set by ACID properties.

- *Cohesions*. If the transaction is configured as cohesion, the atomicity property is relaxed. The application itself determines (using business logic) which participants to confirm or cancel. Participants of a cohesion that are confirmed, i.e., should commit their results form a *confirm set*. The *confirm set* itself is in turn an atom, as all members of this set

should complete successfully (they are confirmed). Cohesion, therefore, follows a nested transaction based model. Cohesions are used to model long running transactions in which participants enrol in atoms which may be cancelled or prepared, depending on certain conditions. It may take considerable time for the cohesion to arrive at a confirm set.

The Web Services Composite Application Framework (WS-CAF) [38] provides an interoperable, easy-to-use and easy-to-implement framework for composite WS applications. It is composed of a series of specifications.

- *WS Context* defines a generic context management mechanism for sharing common system data (i.e., transaction context) across multiple web services.
- *WS Coordination Framework* provides a coordination service that is plugged into transaction context. It manages and coordinates multiple Web services that are grouped together in one or more activities to perform some task together.
- *WS Transaction Management (WS-TXM)* defines three protocols to manage transaction that are plugged into the coordination framework.
  - *ACID transaction (TXACID).* A 2PC protocol to enforce the ACID properties.
  - *Long Running Action (TXLRA).* A protocol designed to cover transactions that have long duration. Compensations are used to ensure the data consistency. This protocol uses the SAGAS model.
  - *Business Transaction Process (TXBP).* The aim of this protocol is to integrate different heterogeneous transaction systems (e.g., using ACID transactions and messaging) from different business domains into one overall business to business transaction.

Each specification covers a certain level of the overall architecture required to build reliable business applications that span multiple systems and use Web service technology.

WS-Coordination, WS-AtomicTransaction and WS BusinessActivity. In 2004 Microsoft, IBM and BEA released a new set of specifications aimed at the reliable and consistent execution of web based business transactions using different interconnected web services. In this set of specifications, the steps to execute a distributed application are regarded as series of activities which are created, run and completed. The Web Services Coordination (WS-COOR) [39] specification provides a generic foundation for web service coordination while WS-Transactions was the first specification of a protocol that is based on that coordination protocol. Later WS Transaction was split in two different standards: Web Services Atomic Transactions (WS-AT) [40] and Web Services Business Activity (WS-BA) [41]. The last versions (1.2) of these standards have been released in 2009 by OASIS. Figure 2.2 depicts the accomplish of the three standards [42].

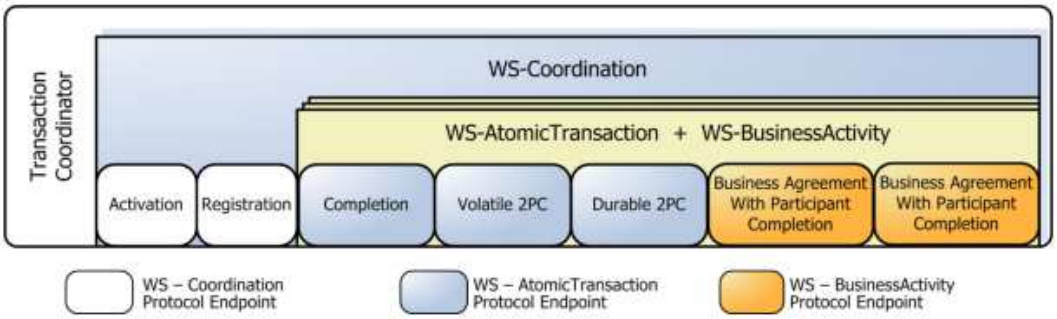


Figure 2.2. WS-AT and WS-BA dependency on WS-COOR

- *WS-COOR.* It provides a generic coordination infrastructure for web services, making it possible to plug in specific coordination protocols (such as WS-AT and WS-BA.) which work between clients, services and participants. Usually one actor, the coordinator, spreads information to a set of participants to guarantee that all participants obtain a particular message. It defines a coordinator, which is an entity that provides services to applications that want to participate in a coordinated activity. Those services are:

  - *The Activation Service.* It is responsible for instantiating a new coordinator and its associated context on behalf of a specific

coordination protocol. It also creates a Registration Service which is implicitly associated with the current coordinated activity.

- *The Registration Service*. It is used by participants to enrol with the coordinator. It acts on behalf of a specific coordination protocol and a specific instance of a coordinator.

- *WS-AT*. It is focused on the existing transaction systems and protocols with strict ACID requirements. Existing transaction systems, that require an all or nothing outcome, form an important part of the companies' back-end infrastructure. WS-AT defines two coordination protocols:

  - *Completion*. This protocol is used by the application that controls the atomic transaction. The application instructs the coordinator to commit or to abort the transaction after all the necessary application work has been done. The coordinator will receive either a Commit or Rollback and then executes the *Volatile 2PC* protocol first before proceeding through to the execution of the *Durable 2PC* protocol.

  - *Two-Phase Commit*. The 2PC protocol used in WS-AT is the same as the traditional 2PC protocol. So it is used to coordinate a group of participants that all need to reach the same decision, either a commit or an abort. The protocol uses the well-know two phases, the prepare phase and the commit phase.

- *WS-BA*. This specification defines protocols that enable existing business process and workflow systems to interoperate. It allows managing long-lived transaction by using compensations. WS-BA defines two coordination protocols

  - *BusinessAgreementWithParticipantCompletion*. In this protocol, a participant of a business activity must know when it has completed all work for a business activity.

- *BusinessAgreementWithCoordinatorCompletion*. In this protocol, a participant of a business activity relies on its coordinator to tell it when it has received all requests to perform work within the business activity.

WS-BA, for each coordination protocol, defines two coordination types:

- *AtomicOutcome*: all participants will be directed by the coordinator to either close or compensate their work.
- *MixedOutcome*: the coordinator should be able to choose which participants should close their work and which ones should compensate.

Academia research

Despite the standardization of some transaction models, the academics have identified open issues and proposed new ideas related to the transactional management of web services.

Charfi et al. [43] observed the lack of support for non-functional requirements (included the transactional requirements) in BPEL-based WS services compositions. They proposed a container framework based on AO4BPEL in order to define and ensure such non-functional requirements. The idea of integrate composition and coordination is also used by Chang-ai et al. [44]. Unlikely Charfi, Chang-ai proposes the use of the standardized transaction protocols WS-COOR, WS-BA and WS-AT. Both Charfi and Chang-ai´s works are aligned with the approach of WS-CAF where the composition and transaction coordination are integrated in the same framework.

Younas et al. [45] proposed a new commit protocol for managing transactions in composite web services. Their main goal is to improve the performance by reducing network delays and the processing time of transactions. The proposed protocol is based on the concept of tentative commit that allows transactions to tentatively commit on the shared data of

web services. Their work, therefore, is aligned with the idea of optimization protocols such as the PA or PC discussed in Section 2.2.2.

Schäfer et al. [46] claim that classical compensation based models are not enough for the WS environment. Instead, they propose a contract-based approach to deal with flexible advanced compensations. They separate the compensation logic from the coordination logic. In this way, their approach allows using different compensation strategies defined on top of basic compensation activities and complex compensation types. In this way, Zhao et al. [47] identify a number of issues with compensation-based extended transaction protocols and describe a new reservation-based extended transaction protocol that addresses those issues. In the reservation-based protocol, an application has full control over the reservation activity, as well as over how long the resource. The protocol defines two steps. The first step involves an exclusive blocking reservation of the resource. The second step involves a confirmation or cancellation of the reservation. Associated with each reservation is a fee, which is proportional to the duration of the reservation should be reserved. On the way of flexible advanced compensations, Ferreira et al. [48] propose a protocol which provides transactional recovery through incremental evolution of exception handling by combining both backward and forward recovery mechanisms. A similar approach is proposed by Jiuxin et al [49] where they propose flexible compensations. All these works disagree with the lack of flexibility in the compensation methods presented in the current transactions standards.

Choi et al. [50] argue that the isolation relaxation mostly used in the ATM (and therefore in the WS transaction standards) introduces a serious inconsistency problem. They proposed a mechanism to ensure the consistent execution of the isolation-relaxing transactions based in end-state dependencies. The notion is a relationship between two transactions in which one transaction's failure incurs the inconsistent state of the other transaction. Once the inconsistent transactions are identified, the mechanism recovers those transactions to the previous consistent states and optionally further re-executes them. The problem to ensure the consistency is also addressed by Alrifai et al. [51]. They propose a protocol that applies a

commit-differing policy to ensure the consistency of concurrent executions. An edge chasing algorithm is used to detect potential global waiting cycles spanning several service providers.

Montagut et al [52] claim that existing WS transaction standards does take into account the workflow requirements of the WS composition. In this way, they propose an adaptive transactional protocol for the pervasive workflow model to support the execution of business processes in the pervasive setting. The execution of this protocol takes place in two phases. First, candidate business partners are assigned to tasks using an algorithm wherein the selection process is based on both functional and transactional requirements. Then the resulting workflow execution is compliant with the defined consistency requirements, and the coordination decisions depend on the transactional characteristics offered by the partners assigned to each task. A similar lack is argued by von Riegen et al [53]. They claim that there is little support of dynamic aspects of transactional management in WS compositions. They propose a set of rules for deciding on the ongoing confirmation or cancelation status of participants' work and protocol extensions of WS-BA for monitoring the progress of a process. Various types of participant vitality for a process are distinguished, facilitating the controlled exit of non-vital participants as well as continuation of a process in case of tolerable failures. Another rule-based approach is proposed by Cao et al. [54], in this case, focus on the recovery mechanism.

On the other hand, Zhang et al.[55] focus on the fulfilment of the ACID properties on WS transaction that require such restrictions. They propose to improve the WS-AT standard to protect the services against Byzantine faults. Khachana et al. [56] also focus on ACID properties but their approach´s distinctive feature is that any or all of the ACID properties may be maintained or relaxed depending on user need.

After the reviewing the work on different transaction models for WS, we have achieve a number of relevant conclusions to the research work presented in this thesis.

- The WS transaction standards share the same notion of a transaction coordinator, participants and a transaction context (Figure 2.3 [42]) and resemble mechanisms which are used in traditional transaction systems.
- The client application interacts with the coordinator to obtain a transaction context which is propagated to all services that are used within the scope of a transaction. Services may then enlist their participants with the coordinator.
- Participants respond to transaction messages and interact with the coordinator according to the specifics of the transaction protocol [2].

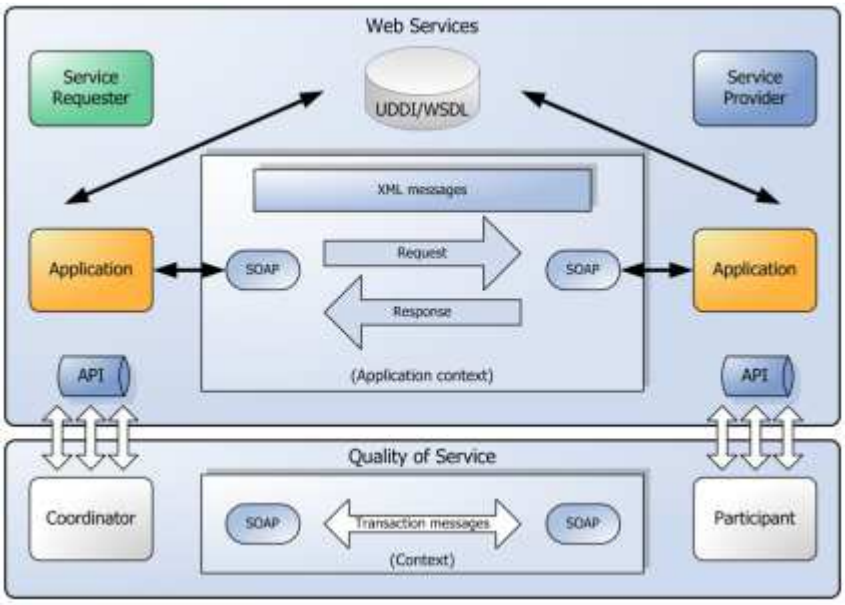


Figure 2.3. Web Services, transactions and contexts

The WS transaction standards are summarized and analysed in Table 2.1. 'Coordination' represents whether a particular standard provides coordination facilities. 'Short' and 'Long' represent that the underlying model is respectively based on ACID properties and advanced transaction models. 'Related' represents the remaining standards which belong to a same family.

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

Table 2.1. WS transaction standards

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

2.3. Software testing in SOA

According to [57], "*one of the main technological barriers to enterprises' transition to SOA is denoted by the heightened importance of the issue of trust. Web services are introduced into systems that require high-reliability and security. Using services in these systems raises the importance of establishing trust between the service provider and the consumer. The trust issue is not just about correct functioning of a service. It has many other dimensions such as reliability. Testing provides one potential solution to the issue of establishing trust.*"

Many efforts have been made in the area of testing web services. The topic of web service testing was surveyed by Canfora and Di Penta [58] and Bozkurt et al. [57]. Also Endo and Simão conducted a systematic review of web services testing but focused on the formal approaches [59] while Palacios et al. [60] focus on composition with dynamic binding.

As stated by Canfora and Di Penta [58], four testing levels can be identified in testing web services:

- *Unit Testing*: Similar to component testing in traditional software. If the service is tested by its developer, structural-based techniques such as code coverage can be applied. In the other case, specification-based techniques must be used.

- *Integration Testing*: Test how independent services work together. The big issues are (i) the lack of information about the integrated components, which makes the production of stubs very difficult, and (ii) the impossibility of executing the integrated components in testing mode.

- *Regression Testing*: A software exposed as a service undergoes maintenance and evolution activities. Maintenance and evolution strategies are out of the system integrators control, and any changes to a service may impact all the systems using it. This changing nature of the web services make this level of testing more important than in the centralized architectures.

- *Non-functional Testing*: Focus on Quality of Service (QoS) attributes such as performance, robustness and reliability.

WS transactions involve the collaboration of different services to meet the requirements of a common bigger process. So testing WS transaction is included in the level of integration testing. Furthermore, transaction is a key issue in building reliable WS based applications [61]. Reliability is identified as a QoS attribute [62, 63], so testing WS transaction also affect the level of non-functional testing.

Some of the issues encountered in integration testing of services are investigated by Bucchiarone et al. [64]. Many efforts have been published to test WS compositions [65-72]. Those works focus the test effort on aspects such us the internal behaviour, services coordination, control flow execution or composition robustness. On the other hand, testing QoS attributes in web services have been addressed in [73, 74]. But none of the previous works address specifically the issue of testing the transactional requirements of the WS compositions.

Existing literature contains various strategies and solutions to test classical transactions, in the areas of databases [75, 76] or system-on-chips (SoCs) at electronic system level [77, 78]. Similarly, work on performance testing and evaluation of transaction models has been received significant attention from existing research [45, 79, 80].

Although the aforementioned literature presents an interesting work on addressing various issues, it lacks research on testing the WS transactions.

2.3.1. Formal verification of transaction processing

On contrary of testing WS transactions, work on formal verification of classical, as well as WS transactions, has been carried out in the literature.

Lanotte et al. [81] develop a model for communicating hierarchical timed automata in order to describe long-running transactions. This approach allows the verification of properties by model checking. Their approach takes in account the compositions but is limited to sequence and parallel execution patterns. A similar work is presented by Kokash and Arbab [82]. They suggest using the channel-based coordination language Reo to model the long-lived transactions and verify their properties using model checking technology.

Aligned with the previous works, Emmi and Majumdar [83] presented a work to translates programs with compensations to tree automata. They argue that usual trace-based semantics for web services compositions leads to

an undecidable verification problem, but a tree-based semantics gives an algorithm that runs in time exponential in the size of the business process. It allows a safety verification of compensating transactions.

Gaaloul et al. [84, 85] use event calculus to validate the transactional behaviour of WS compositions. The transactional behaviour verification is done either at design time to validate recovery mechanisms consistency, or after runtime to report execution deviations and repair design errors, and therefore, formally ensure service execution reliability. This work takes into account the execution of the transaction but does not define how the tester has to do it (the test case design).

Saleh et al. [86] present a data modelling and contracting framework for WS that help formally verify data integrity properties in transactions. Although the work addresses the important issue of data verification, their approach is limited to ACID transactions.

In addition Li et al. [87] propose a formal model to verify the requirement of relaxed atomicity whilst Bhiri et al. [88] propose using the Accepted Termination States (ATS) to achieve the same goal.

All the above works deal with verification transaction-related properties from a theoretical point of view. Thus, they are focus in the design phase. None of them take into account the possible failures included during the implementation (code developing) phase.

2.4. Discussion and Summary

Transactions in WS environment are complex as they can have duration, involve independent providers and go through different network, hardware and software configurations. While ACID properties are desired in restrictive scenarios, new transaction models inspired by the ATM have been designed to deal with the SOA characteristics. Some of those proposals have been standardized and included in the WS standards stack.

On the other hand, very interesting effort have been done in the area of testing web services. But we have noticed that although transactions are a key issue in developing reliable SOA based applications, the current literature does not pay attention to the issue of testing transactions. Testing transactions should be included in both integration and non-functional testing levels.

Existing works have addressed the formal verification of WS transactions. These works mainly propose formal approach to verify the consistency and correctness of the process. However, these approaches do not ensure that the implementation satisfies the properties since there is no formal link between the design model and their implementation. Thus, it is difficult to predict that the software fulfils those constraints since the implementation phase may include faults.

According to the conclusions achieved after the current literature review, we identify the following open issues in the area of WS transactions:

1. Web Services transactions require a flexible transaction model since the business logic could be complex and dynamic. The existence of different WS transaction standards does not help to the interoperability of the applications and also make more difficult the design, implementation and testing phases. Currently there is no a generic model capable of representing the transaction independently of the standard used.

2. Transaction is a key issue to ensure the reliability of a WS based application. In consequently, the transactional requirements shall be taking into account in the test process. Currently there is no specific method to test WS transactions.

3. A WS transaction is composition of web services with extra conditions. Currently the business logic of composition and coordination (transaction management) are separately in different standards. It would be useful to integrate both protocols.

4. Classic compensation based mechanism are not enough for WS transaction. An advanced flexible compensation protocol should be standardized and included in the WS standard stack.

In this thesis, we address the open issues 1 and 2. Our main goal is to define specific test methods for WS transactions (open issue 2) but because of the variety of transaction models, we took the decision of defining a generic model (open issue 1). Chapter 3 presents the proposed Abstract Transaction Model (AbTM) to represent the existing transactions protocols and standards. Then, chapter 3 also presents the Framework for Testing Transactions (F2T) which has been designed and developed for testing WS transactions.

Chapter 3

The proposed model and framework

To win without risk is to triumph without glory

Pierre Corneille

This chapter presents the two main components of the research work presented in this thesis: the Abstract Transaction Model and the Framework for Testing Transactions. The objectives are to address the research issues identified in Chapter 2.

As discussed in Chapter 2, there exist different WS transaction models and standards. Such diversity of models makes difficult the process of testing the transactions. Consequently, it is necessary to develop a generic or abstract model that can represent all such models and standards. The first part this chapter presents the design and development of the proposed Abstract Transaction Model (AbTM).

In the second part this chapter presents the Framework for Testing Transactions (F2T) which has been designed and developed for testing WS transactions under the proposed Abstract Transaction Model. F2T has been devised to organize all the concepts involving in the process of test case design. The framework is inspired by the risk-based methodologies and is hierarchically organized in four levels.

3.1. Abstract Transaction Model

This section describes the design and development of the proposed Abstract Transaction Model (AbTM). The goal of the AbTM is to abstractly represent existing WS transaction models and standards. In other words, it captures the behaviour of a WS transaction independently of the underlying standard or model. It therefore, serves as a template for existing transactions model and standards and provide an easy and uniform way for testing different WS transactions.

The scope of the AbTM is defined by the existing approaches (discussed in Section 2.2) to manage distributed transactions. In this way, it is assumed that the WS transaction either follows a standardized protocol (BTP [36], WS-COOR [39], WS-AT, WS-BA[41], WS-CAF[38]) or is managed as a WS composition (WS-BPEL [89])

3.1.1. Fundamental Concepts and Definitions

This section first describes the main definitions related to a WS transaction. Then it is presented the different roles used by the AbTM to abstractly define the transactional process.

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. The set of functional information and transaction configuration shared by the activities of the wT is called *transaction context*.

The outcome of wT is the final decision of the process. In other words, if the logical unit of work defined in the transaction is performed or not. The outcome is called *atomic* if all its activities are either successfully completed or compensated. Alternatively, if activities can differ (some completed and some not), then the outcome is called *mixed*.

Activities represent the elements of work that form a wT. That is, the work performed by each involved service. An activity can be atomic (so-called *task*) or non-atomic (so-called *subtransaction*). An activity is compensatable if a compensating activity exists within the wT to undo its actions. An activity is retriable if it can be re-executed without causing any data inconsistency. An activity is replaceable if there is an alternative that can perform the same work.

A **subtransaction** is an activity which can be a transaction itself. Thus the structure of wT follows the nested transaction structure. The WS transaction tree, made up of these nested relationships, can be arbitrarily wide or deep – there are no fixed limits to how many activities a transaction can have, or how many levels of subtransactions there are between the top-most coordinator and the bottom-most leaf executor. The actual creation of the tree depends on the behaviour and requirements of the transaction. Figure 3.1 shows such relationship wherein wT_p, is a parent of a_1, a_2 and a_3. The activity a_1 (or wT_c) is in turn a parent of a_{c1} and a_{c2}.

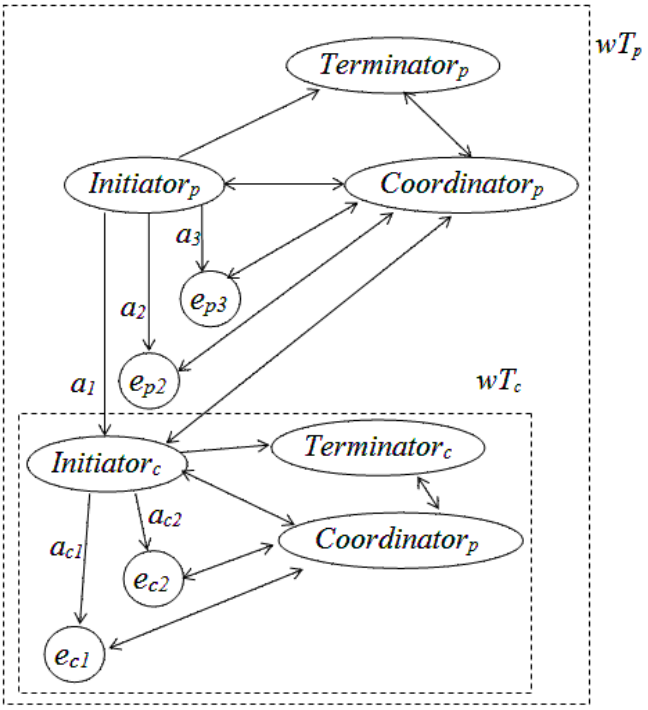


Figure 3.1. Nested transactions

Compensation is an activity that undoes from a semantic point of view the actions performed by another activity.

Dependencies are constraints on the processing produced by the concurrent execution of activities. A dependency defines a relationship between a set of activities. The types of dependencies are explained in Section 3.1.4.

There are four activities always present during the life-cycle of a WS transaction: creation, coordination, execution and finalization. Creation refers to the process to create a new wT, that is, to generate a transaction context that will be used for the involved services during the transaction execution. Coordination refers to the management of the transaction from its beginning until the end. Mainly it is related to the communication between the services in order to achieve an agreed outcome. Execution refers to the actions of each service to perform the work specified in the activities that form the transaction. Finalization refers to decision about the final outcome of the transaction taking in account the partial results of each service.

According to the activities commented above, the execution of a wT involves different participants, each of which plays a certain role. As shown in Figure 3.2 the AbTM defines four different roles of the participants involved in processing wT:

- *Executor:* It represents a participant which is responsible for executing and terminating an activity.
- *Coordinator:* It communicates with individual participants, coordinates the wT and manages failures and compensations. It also collects the results from the executors in order to maintain consistency of data after the execution of wT.
- *Initiator:* It represents a participant which starts the wT. First it requests the coordinator for a transaction context. Then it asks others participants to participate in wT. Accordingly, the initiator is a simple participant that does not offer any services to the transaction coordinator or executors.

- *Terminator:* It represents a participant which decides when and how wT has to be terminated. It also participates in the coordination tasks. In some situations, it can play the role of a sub-coordinator. It performs the next tasks: (i) decide whether the context's overall goals can be achieved by querying the executor. In mixed outcome contexts, to decide for each executor its suitable end, and, in atomic outcome contexts, decide whether to confirm or to cancel/compensate all work. And (ii) send the suitable message to the coordinator about the final transaction outcome.

Each wT is associated with one *Coordinator* and one *Termintor* while each activity, a_i, $\in$ A, is executed by an *Executor*, e_i. The behaviour of an Executor is further detailed in Section 3.1.2.

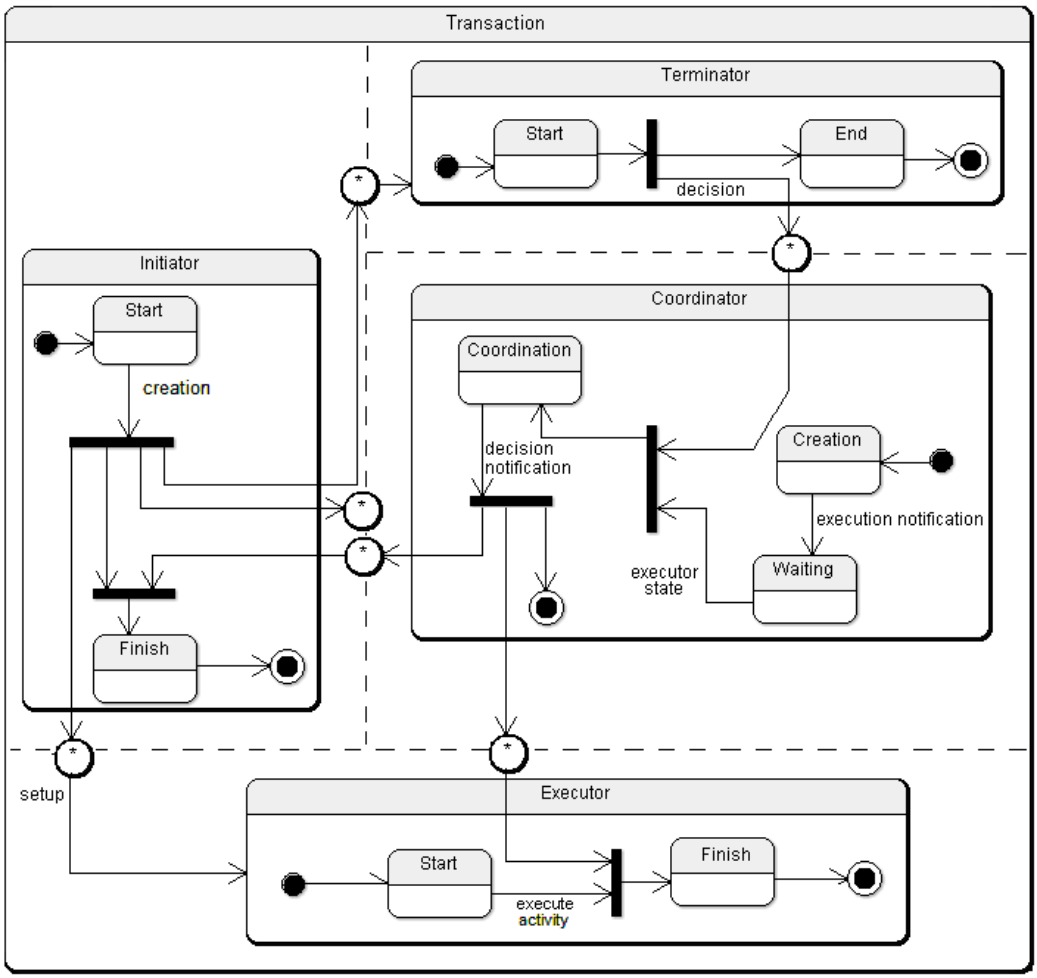


Figure 3.2. Roles in WS transactions

3.1.2. Executor

An executor is a web service enhanced to be able to take part in a transaction. So it is composed by web service itself plus the transactional interface (Figure 3.3).

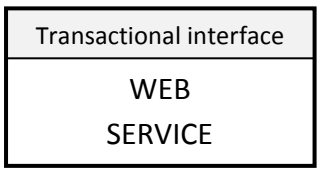


Figure 3.3. Executor

Each executor is in charge of executing an activity. This is a key role in the processing of transaction so their model is it as well. The model resides at various levels of abstraction. The most abstract version maps each possible input (*start* state) to the output *end* (Figure 3.2). But the model must be sufficiently precise to serve as a basis for the generation of test cases according to the test goals. The executor´s test goals, as is further explained below, are related to (a) its behaviour and standard performance, and (b), the dependencies (relationships) with other activities. So the AbTM defines more detailed models for the executor according to the level of abstraction.

Executor Model

WS transactions do not have a homogeneous transaction model such as the ACID (Atomicity, Consistency, Isolation and Durability) model. Instead they are characterized by a diversity of transaction models such as BTP [36], WS-BA [41] and WS-TXM [38] reviewed in Chapter 2. Such diversity of models also complicates the process of testing the executors. 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 messages, processing of services, etc. and (ii) service level failures such as service acquisition failures wherein services cannot be acquired because of unavailability of the desired services, payment problems, or service cancellation. The test process of testing the executor shall take into account the previous situations when defining the test conditions.

All the above failures affect the reliability of WS transactions. Thus it is important to have a model in order to analyze different transaction models, generate test case specifications and test their reliability in terms of failures. A specific model for each transaction standard should be necessary. But if we define a model for each standard and base the test case design process on it, we would be defining tests for the implementation, rather than taking into account the transactional nature of the process. An abstract (generic) model for an executor independently of the protocol used would be useful from a testing point of view. The Executor Model (EM) is devised with those purposes. It have to be able to capture the behaviour of a executer running under a specific transaction protocol, but also the dependencies (relationships) between the activities involved in the wT. The EM, therefore, is presented in two versions. The first version of the model focuses on the dependencies between different executors while the second one is focus on the behaviour of each executor during its cycle-life. The EM versions are represented using the well-known UML statecharts [90] notations which reflect the event-driven (message communication) nature of the WS transactions

Executor Model for dependencies

The EM defines the dependencies between the activities in terms of the actions carried out by the executors. For example, an executor can start executing its activity only once another executor has finished its execution. The goal of this model, therefore, is to capture the different actions that an executor can do during its life-cycle. It is defined as follow:

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 Figure 3.4. 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 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.

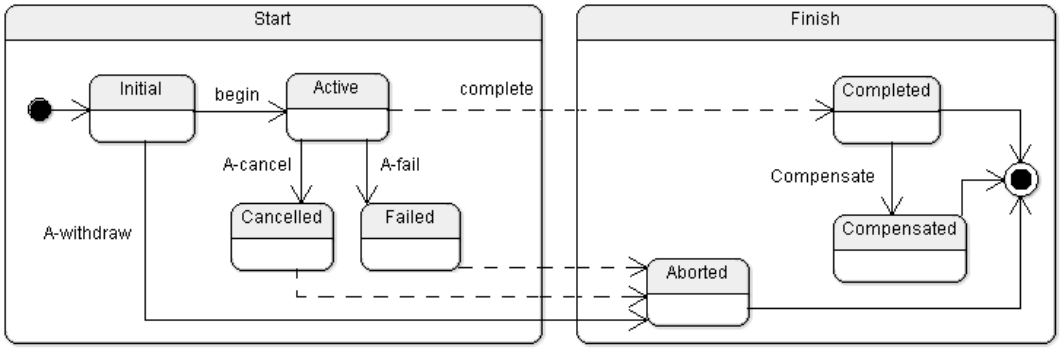


Figure 3.4. Executor Model for dependencies

Executor Model for behaviour

When the executor is implemented according to the web services paradigm, it must follow a concrete transaction model defined in a protocol. We need, an abstract executor model focus on the behaviour that serves as a template for modelling (capture the standards special features) and testing executors running under different WS transactions standard. That is the EA for behaviour version, depicted in Figure 3.5.

The sequence of messages from the behaviour part of the EA can be automatically translated to a particular syntax of a WS transaction standard (see Section 4.2.1). How the states and transitions of this abstract model can model (or pattern) existing WS transaction standards is also shown in Section 4.2.1.

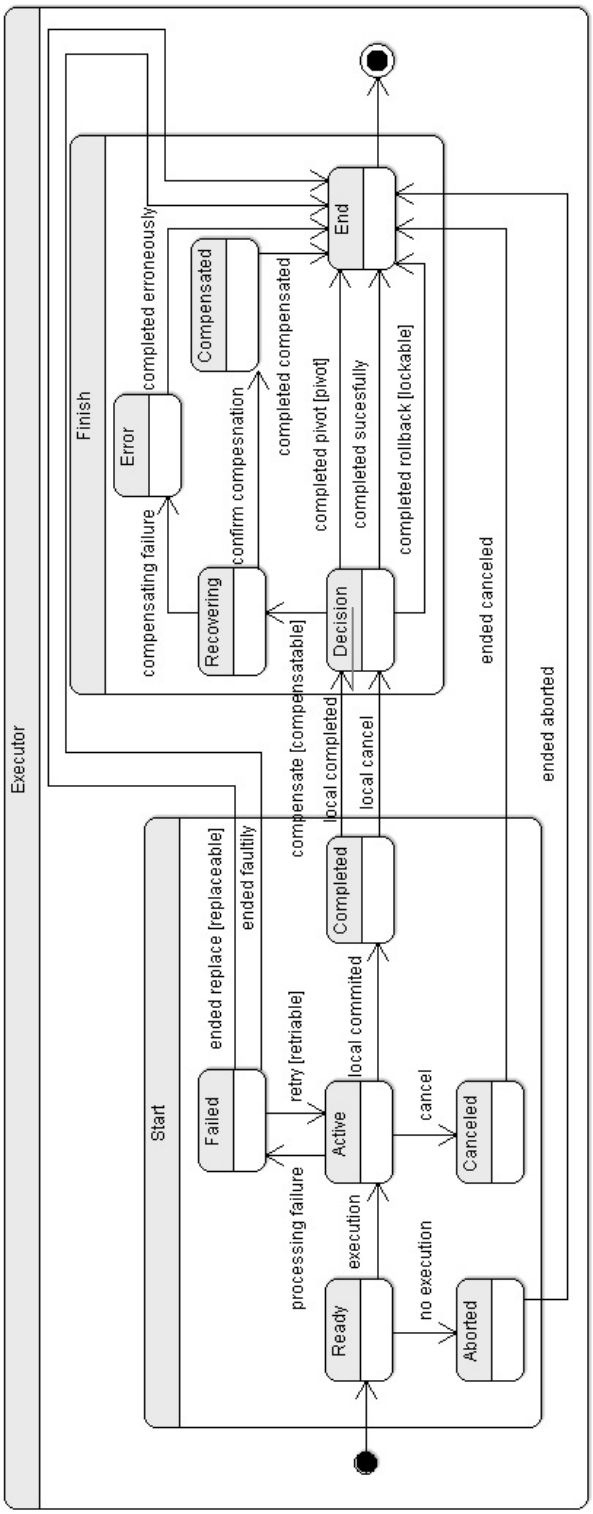


Figure 3.5. Executor Model for behaviour

3.1.3. Coordinator

The coordinator is in charge to communicate with the executors pursuant to the standard in order to ensure the correct execution of the transaction. In the same way that the executor, models with different level of abstraction can defined. Figure 3.6 shows a model of a coordinator focus on its actions (*creation*, *waiting* and *coordination*). Here we defined the Coordinator Model (CM) focus on the details of its possible behaviours during its life-cycle. It is shown in Figure 3.6. The relationships between coordinator and other roles have been shown in Figure 3.2, so here we focus on its behaviour.

The coordinator is active to manage a specific transaction. It waits until has the enough information to communicate the final decision to the executors. That behaviour depends on the concrete protocol used to manage the transaction. In addition, the coordinator can bear different faults during the transaction management. Those scenarios are relevant from a testing point of view. How the states and transitions are used to model the standards is shown in Section 4.2.1.

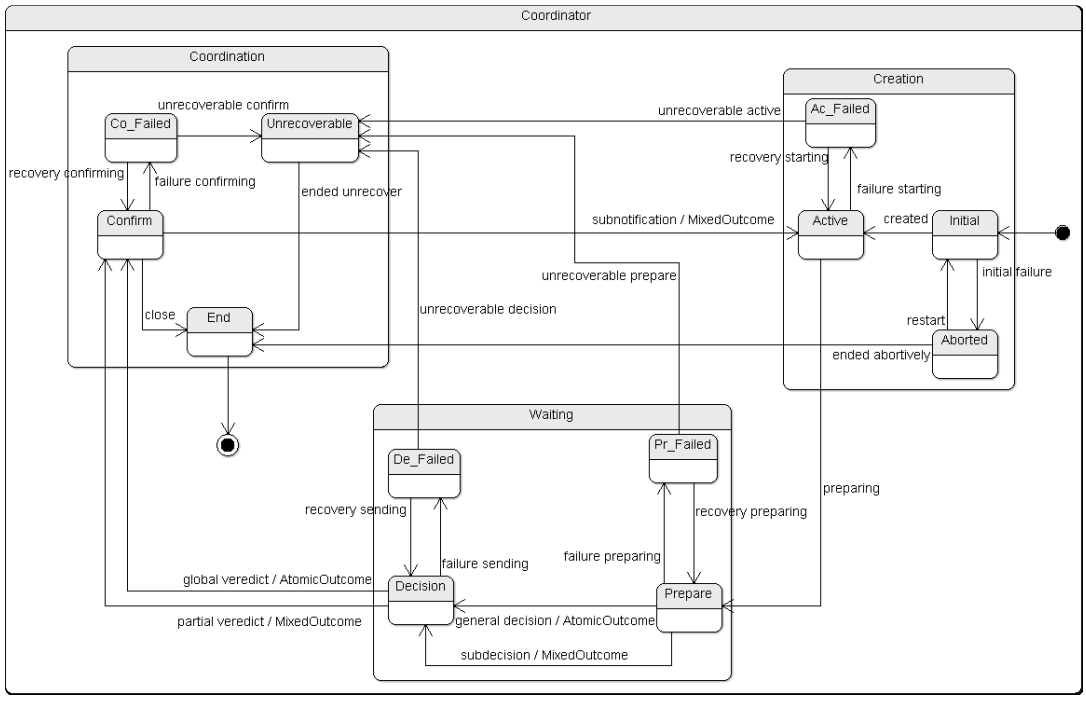


Figure 3.6. Coordinator model

3.1.4. Dependencies

The AbTM presents a wT is a collection of existing WS working together to offer an agreed combined outcome. The process modelled as a transaction is composed by a set of activities and a set of dependencies (relationships) between such activities. Each activity is executed by an executor and the AbTM uses the EM to model the executor´s behaviour. While the EM and CM focus on the behaviour of the participants, this section presents a complementary approach to model the dependencies between them.

The dependencies specify how services are coupled and how the behaviour of certain services influences the behaviour of other services. The AbTM identifies three kinds of dependencies in WS transaction: flow, data and control. 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). The three kinds of dependencies allow capturing the necessary requirements to derive the test conditions and test coverage items (see Section 3.2.1). They are presented as follow:

- **Flow dependencies** define constraints on the workflow in terms of the order of execution of activities.
- **Data dependencies** define relationship between the data used by the activities. These specify relations according to read and write operations on shared data.
- **Control dependencies** are hybrid dependencies (a mix of flow and data dependencies). These dependencies refer to *feature dimension* described in Section 3.2.6.

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 updates (writes) it to 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 in the input of 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 has been completed. For example in *Exclusive*, the dependency is completed if exactly on activity is completed. In Join, the dependency is completed if all activities have been completed

The dependencies used by our method are graphically represented using a BMPN based notation as shown in Table 3.1. Table 3.2 presents some examples for further explanation of the dependencies.

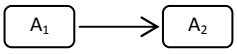
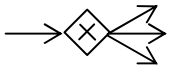
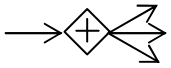
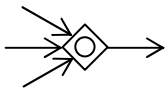
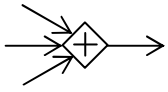
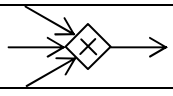
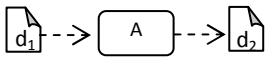
Name	**Notation**	**Description**
Sequence		The activity A_1 must complete before activity A_2 can begin
Alternative		Only one activity can begin.
Fork		All the activities begin.
Merge		At least one activity must complete before another can begin. Extra conditions can be specified.
Join		All activities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

Table 3.1. Dependencies

Dependency	Example
Sequence	The item is sent to the customer once the payment has been confirmed
Alternative	In a purchasing process, the customer selects one method (credit card, bank transfer, *Paypal*) to pay for the item.
Fork	In a journal review process, the editor sends the email to all the reviewers.
Merge	At least one means of transport (car, train, plane) has to be available before continuing the package holiday reservation.
Join	All the bookings (flight, hotel, car rental) have to completed before paying for the package holiday
Exclusion	When different hotel providers are consulted, only the cheapest one has to complete
Write	The tax to be paid depends on the number of items sold in a day

Table 3.2. Example of dependencies

3.2. Framework for Testing Transactions

This section presents the proposed Framework for Testing Transactions (F2T). Firstly we introduce in Section 3.2.1 some fundamental concepts about the process of test case design and risk-based testing. The F2T, presented in Section 3.2.2, has been devised to organize all the concepts involving in the process of test case design for WS transactions. It leverages such structure for the generation of specific suitable test suites for WS transactions. The elaboration of the F2T was inspired by the risk-based testing methodologies. The elaboration of the framework is described in sections 3.2.3 to 3.2.6.

3.2.1. Foundations

Test case design

According to the International Software Testing Qualifications Boards (ISTQB) [91], testing is defined as:

The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

Thus, one aim of testing is to systematically explore the behaviour of a system or a component in order to detect unexpected behaviours. 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 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, test techniques can be used to ensure that testing is carried within the constraints of available resources. Test techniques provide guidance to design test cases using some information about the SUT, for example, the workflow specification or the data usage. They allow the systematic identification of the most relevant conditions and most important values to test. Below we introduce some definitions about the test case design process. Figure 3.7 shows the relation between these concepts.

Test basis: It represents all sources from which the requirements of a component or system can be inferred.

Test items: Test basis is broken down into *test items* that are the minimal functional unit that can be tested in isolation.

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 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 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. A *test case* is a set of input values, execution preconditions, expected results and execution postconditions, that cover and exercise a set of test coverage items.

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

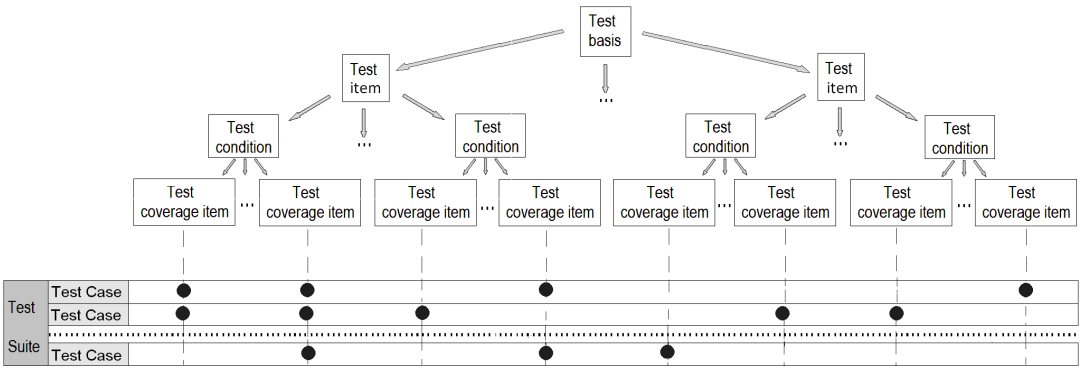


Figure 3.7. Test case design concepts

To manage the above concepts, i.e., which test basis to use, how to identify the test items, how to derive the test condition and test coverage items, and how to define the test cases, it is necessary a test strategy. We have used a strategy inspired in the risk-based methodologies to achieve a whole test case design process for WS transactions.

Risk-based testing

The seed for the whole test case design process is the test basis (Figure 3.7). Very common used test basis are specifications where is defined what the software (or a component) is expected to do [92]. There are different requirements specifications according to the development phase: user

requirements, functional specification, physical design, and program specification. Many testing techniques, called specification-based techniques [93], have been proposed to achieve test cases using such test basis. But the set of specifications is not the only approach that can be used as test basis. Another approach is to base the test process on the risks related to the system rather than then software functionalities. The strategies that consider both specification and risks are called risk-based testing.

The main goal of risk-based testing is to find the most important defects as early as possible. A defect in the system may lead to undesirable effects for both the development company and the system users. The composition of likelihood and consequence of such unwanted defect is referred as a risk exposure. By identifying and analyzing the risks related to the system, it is possible to focus the test effort on the most critical areas of the system. In fact, the testers have always used risk-based testing but using and ad-hoc fashion since they based their decision on their personal expertise [94]. Hence, the contribution of the risk based techniques to the field of software is to provide risk assessment methods to the test process.

Risk analysis is a set of techniques used to investigate problems created by uncertainty and to assess their effects. Originally it was used in areas like nuclear, chemical and space industries, and nowadays it is used in software development where safety is very important too [95]. But risk-based testing can also be used to give directions for which test strategies to use such as selecting test basis and the most important test items or evaluating the testing techniques to derive test conditions [96, 97]. This approach of using a risk-based strategy inspired the research methodology used in this thesis.

Several risk-based methods have been proposed to the software testing process [98-103]. They are mainly adaptations from more generic risk methodologies such as HAZOP (HAZard and OPerability study) [104] and FMEA (Failure Mode and Effect Analysis) [105]. The high-level common steps of all methods are the following:

1. *Area of study definition*. Specify the elements and context where the method is applied.

2. *System division.* To decompose the whole scope into smaller subsystems,
3. *Risk identification and analysis.* Identify what could go wrong and assign scores for the both probability and consequence.
4. *Risk response planning.* To propose ways to mitigate the each identified risk.

The above steps have provided guidance to devise the F2T as is described in Section 3.2.2.

3.2.2. Conceptual framework

As was earlier discussed in Section 2.3, there are no specific methods to test WS transactions. So the main goal of the Framework for Testing Transactions (F2T) is to organize the concepts related to test WS based applications with transactional requirements. This planning allows applying the most suitable techniques as appropriate. F2T is hierarchically organized in four levels, inspired by the four generic steps of the risk-based methodologies described in Section 3.2.1. Figure 3.8 illustrates the framework.

A previous stage before defining the F2T levels was to achieve depth knowledge of the transaction and web services related fields. Key issues are the different models and standards published to manage the transactions and the existing works about verification and validation focused on services compositions. The starting point is, therefore, the literature review analyzed in Chapter 2.

- The first level of the F2T defines the transaction using the AbTM. It focuses on the activities involved in the transaction and the relationships between them.
- The second level aims to divide the objective of testing WS transactions in smaller goals. Following the approach of ISO 9126 software quality model [62], F2T defines a set of system properties (or characteristics)

that should be tested to evaluate the correct behaviour of a WS transaction.

- In the third level, F2T identifies, for each system property, a set of hazards that will compose the test goals.
- The fourth level defines the approach to mitigate the risks, in our case, the test techniques to address the test goals previously identified in the third level. Firstly we define the dimensions of testing such goals with the objective of organize all the concepts. That is, we analyze what could be tested and how could be done. Then, we propose specific methods for test case generation in order to address the testing goals. For this purpose, each system property is isolated analyzed in order to adapt or create suitable test techniques to be used for achieving specific test cases.

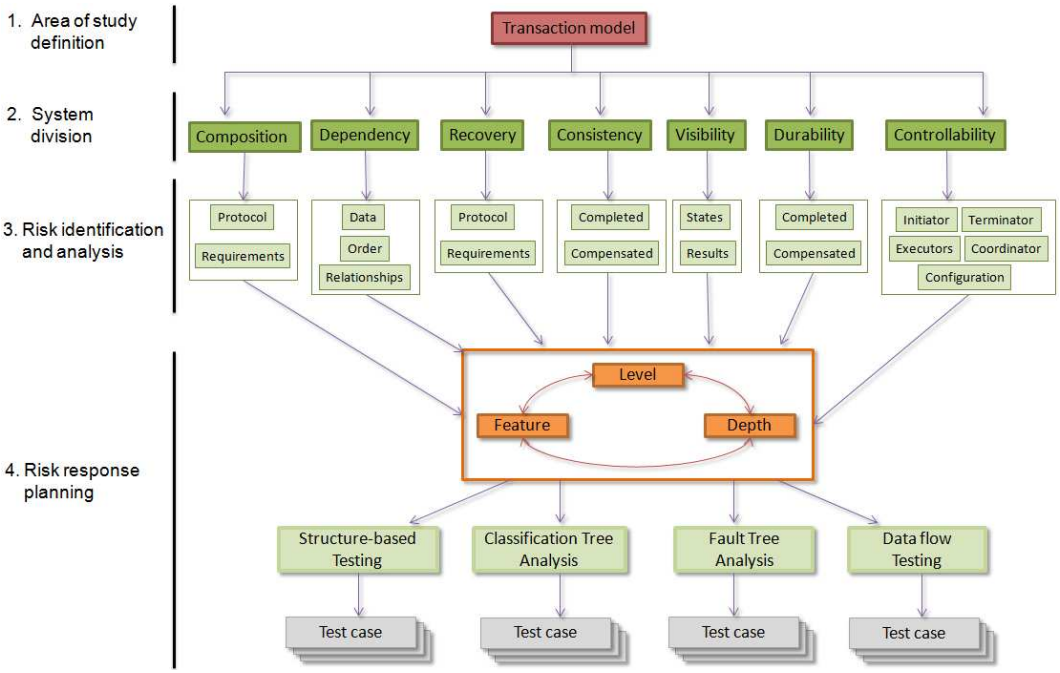


Figure 3.8. Framework for Testing Transactions (F2T)

3.2.3. First level: Area of study definition

According to the risk-based method, the first step is to define the system where the method is applied. In our context, the system is the WS

transaction model. As was commented in Section 2.2, both academic and industrial worlds have proposed different models and standards to manage transaction in web services environments. It makes more difficult the test process. Some activities are always presented in the transaction management, such as creation or termination. Also the same roles al always presented independently of the model used. For example, in an application that allows booking different services for a night out (e.g. theatre tickets, restaurant and taxi), the client side of the application starts the transaction because it has the customer information. Also, it finishes the transaction because it knows the customer's requirements about the whole reservation process.

Since different models can be used to manage the process, we propose to use a generic (or abstract model) to define (or pattern) the behaviour of the transaction running under the different standards discussed in Section 2.2. Therefore, the F2T uses the AbTM (section 3.1) since it is capable of capture the semantic of the transaction process independently of the standard and protocol used.

3.2.4. Second level: System division

The goal of this level is to identify the particular test objectives in a WS transaction. This is, therefore, related to the test types and quality properties applicable. A common approach to formulating a model for software product quality is to first identify a small set of high-level quality attributes and then, in a top-down fashion decompose these attributes into sets of subordinate attributes [106]. The ISO 9126 software quality model [62] is typical of this approach. Here we derive a specific quality model for the WS transactions composed of a set of so-called system properties.

Existing works [24, 25, 30, 46, 50, 107-110] have addressed the features of advanced transactions models in distributed and service-based environments. We focus on the relevant properties from a testing point of view.

A WS transaction is *composed* of activities [24] that have autonomy to commit or abort unilaterally. So the atomicity is relaxed and isolation is violated because the results of the committed activities are *visible* to other transactions. This new concept of atomicity (called semantic atomicity [30]) means that if any of the activity is aborted then the effects of the committed transactions must be compensated [25]. Hence the system must be able to *recover* [46] its previous state maintaining the *consistency* [50]. There are relationships between the activities involved in a transaction. These relationships, called *dependencies*, define constraints on the processing produced by the concurrent execution of interdependent activities in terms of control flow and data [108, 111]. To manage these operations is necessary a specific *coordination* [109]. As in the ACID model, the *durability* of the results of a completed transaction is a desired property [110]. Consequently, we divide the scope of testing WS transaction in addressing the following system properties:

Composition: A wT is composed by a set of activities, each executed by a service. It is necessary all services to provide the desired functions achieving the correct results.

Dependency: There are dependencies between the activities involved in a wT. These dependencies specify constraints in the flow of execution and define relations with the data shared during the transaction processing.

Recovery: A wT has to be able of re-establish its level of consistency and recover the data directly affected in case of a failure or a user requirement of cancelation of any of their activities.

Consistency: Activities and their compensations must maintain the required consistency of services when used under any aborted, compensation of completed conditions.

Visibility: A wT allows their activities and other transactions to see the partial results of its activities.

Durability: Once a wT is finished successfully the results will remain permanent in the system.

Controllability: A wT requires a participant to take the role of coordinator with the goal of coordinate the process ensuring the Composition, Consistency, Durability, Dependency and Recovery properties.

The system properties describe a software quality model for applications that rely on WS transactions. They define aspects related to the characteristics such as functionality or reliability defined in the ISO 9126 software quality model [62]. Therefore, we can define a mapping between the system properties and the ISO 9126 characteristics. This relationship is shown in Table 3.3, where column *Characteristic* means one of the six general categories defined in the ISO 9126, and column *Section* refers to the sub-characteristics defined in the standard. In the rest of this work, we use the system properties because are specific for the context of WS transactions, unlike the most of testing approaches that follow the generic ISO 9126 characteristics.

System property	Characteristic	Section
Composition	Functionality	Accuracy Suitability Compliance
Dependency	Functionality	Interoperability
Recovery	Reliability	Fault-tolerance Recoverability
Consistency	Functionality	Security
Visibility	Maintainability	Analyzability Testability
Durability	Maintainability	Stability
Controllability	Efficiency	Compliance

Table 3.3. System properties and ISO 9126 characteristics

3.2.5. Third level: Risk identification and analysis

Once the system to be tested is divided in smaller areas, i.e. the system properties, the next step is to define what should be tested in each one. We identify the aspects that could cause failures, drawing, in this way, an analogy with the risk identification process specified in the risk-based

methodologies. So we define the risks in the form of hazards that concern to our test goals (system properties). Test techniques will be defined to address those hazards in the next level of the framework.

The identified hazards for the system properties are presented below. A hazard is described as an expected requirement that, if it was not meet, it would likely cause a failure in the transaction processing. An ID (in brackets), name and description are provided for each hazard.

Composition

- [COM1] *Requirements*: Each service shall fulfil its requirement specification when it executes their activity.
- [COM2] *Protocol*: Each service shall meet the expected transaction protocol during the execution of its activity.

Dependency

- [DEP1] *Order*: The activities shall be executed fulfilling the defined order of execution.
- [DEP2] *Relationship*: The activities shall be executed fulfilling the constraints defined on the processing of the concurrently executing activities.
- [DEP3] *Data*: The relationships between the share data used by the activities shall be correctly handled.

Recovery

- [REC1] *Requirements*: Each service shall fulfil its requirement specification when it executes the compensation of a previously completed activity.
- [REC2] *Protocol*: Each service shall meet the expected transaction protocol during the execution of compensations.

Consistency

- [CON1] *Completed*: The successfully execution of any activity shall bring the transaction from one valid state to another according to the specified requirements.
- [CON2] *Compensated*: The execution of any compensation shall bring the transaction from one valid state to another according to the specified requirements.

Visibility

- [VIS1] *Results*: The results reached by each finished service shall be visible for other activities of the same or another transaction.
- [VIS2] *State*: The current state, according to the transaction model used, shall be externally visible for coordination and testing purposes.

Durability

- [DUR1] *Completed*: Results shall remain in the system once the transaction has been successfully completed.
- [DUR2] *Compensated*: Results shall have been semantically undone once the transaction has been compensated.

Controllability

- [COT1] *Configuration*: The transaction need to be configurated (type, protocol, shared information)
- [COT2] *Coordinator*: Coordinator shall work in presence of networks failures.
- [COT3] *Initiator*: The initial process shall create correctly the transaction in collaboration with the coordinator.
- [COT4] *Executors*: All the suitable services shall be enrolled in the transaction.

- [COT5] *Terminator*: The transaction shall achieve the right agreed outcome according to the requirements specified.

3.2.6. Fourth level: Risk response planning

In this level of the framework we address how the identified hazards can be addressed. First we define the scope (dimensions) of testing such elements to organize the concepts involved in the test case design; in this way, we can adapt or create the most suitable test techniques in order to derive the test conditions to address such hazards.

Testing dimensions

Typically, the test case design process includes the dimensions of *test level*, *test type* and *test depth* [112]. Test level defines the specificity of the test such as unit, integration and system levels. Test type refers to the quality attributes that those tests are focus on (see Section 3.2.4). And finally, test depth refers to the test effort required. In this work we propose a characterization of those dimensions to achieve a specific test case design process for WS transactions.

We identify three orthogonal dimensions for testing WS transactions. *Level* dimension defines the granularity level of testing, i.e., testing WS transactions at different levels such as activity (web service), nested subtransaction or at whole process (transaction) level. *Feature* defines the source used to identify the situations to be tested (test conditions), for example the flow of execution or the data elements shared by the services. *Depth* is related to how to combine such conditions (test coverage items) to design the test cases in order to achieve a cost-benefit trade-off.

Level dimension. The level dimension refers to the granularity level of testing. It defines the test items which be used. Three values for the level dimension are defined: executor, transaction, business process.

Participant. Different roles are involved in a WS transaction. A key role is the executor due to the activities that compose a WS transaction are carried out by executors. In fact an executor is a role entrusted to a web

service. So a first level of testing should consider each service as a test item. 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. The test cases for this level have to exercise the different situations that a service (e. g. an executor) has to manage during its life-cycle.

Transaction. A WS transaction is represented as a set of related activities that have to achieve an agreed outcome. So they form a logical unit of work. The integration and coordination of the participants forming the transaction as a whole should be considered as test item.

Business process. A business process can rely on one or more WS transactions to fulfil the whole business process requirements. So the integration of the specific business logic of the process and the WS transaction should be considered as test item.

Recursive application of levels. A WS transaction is composed by activities where each activity can be an atomic task or another WS transaction itself (subtransaction). On the other hand, a WS transaction can be part of a bigger process too. Therefore, previous levels can be applied recursively to both upper and lower nested process. In order to depict the recursive relations, Figure 3.9 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 participant 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 participant level to test *E* and *F* and the transaction level to check their relationships.

The scope of this thesis includes the participant and transaction levels. At both levels we mainly focus in the executor role. To deal in depth with the other roles as well as the business process is proposed as future work.

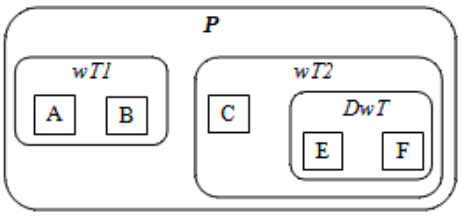


Figure 3.9. Recursive test levels

Feature dimension. The feature dimension refers to the source used to derive the test conditions and, consequently, the test coverage items. Three feature are defined: flow, data and control

Flow. An executor passes through different states during the execution of an activity. The dependencies in a WS transaction define the order and constraints of the execution of the activities. Thus, for both executor and transaction level, a control flow analysis can be derived to identify the test conditions. In the executor level the flow is defined by the states/transitions model that it follows. Therefore, the test conditions can be defined in terms of the coverage of a particular set of elements in the structure of such model.

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. By looking for patterns of data usage, risky situations are identified and more test conditions can be defined.

Control. The decision of an executor moving from one state to another may depend on the value of one or more data elements. 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 when more than one service are available to execute the same task, 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.

Depth dimension. The depth dimension refers to how combine the identified test coverage items. So it is related to the test effort required.

Test techniques 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, but this can be achieved in different ways, depending on the required test effort. Thus different strategies are applicable to combine the test coverage items that will be exercised by the test cases. In this way, 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 simply to cover all the test coverage items using the lowest number of test cases. Thus the test criteria propose different test efforts ranging from the minimum to the maximum effort.

Test techniques

Finally each hazard is analyzed taken into account the three testing dimensions in order to define the most suitable test technique for each case. The risk response planning in our context is a specific test suite. The next chapters of this thesis present the test techniques and methods developed to achieve the suitable set of test cases for the identified hazards.

3.3. Summary

In this chapter we have presented the two main components of this thesis: the Abstract Transaction Model (AbTM) and the Framework for Testing Transactions (F2T).

AbTM addresses the issue of the wide range of existing transaction model to manage service-based transactions that make the test process more difficult. It defines a transaction as a set of activities and a set of dependencies between those activities. AbTM also identifies four roles that are always present in the transaction life-cycle: initiator, executor, coordinator and terminator. Different levels of abstraction are used to define

the models for the coordinator and executor roles. These different models will be used by the F2T according to the test goals to address.

F2T is designed inspired by the risk-based methodologies and addresses the issue of testing service-based transactions. It encompasses the concepts from the transaction definition to the test case generation. F2T identifies a set of hazards and defines how the test techniques can addressed them.

In this thesis, we have addressed the Composition, Controllability and Dependency system properties. Table 3.4 summarizes the hazards and the techniques proposed in this thesis. To deal with the Composition and Controllability properties, Chapter 4 presents a structure-based testing technique. Two different testing techniques have been proposed to deal with the Dependency property. Chapter 5 uses a condition-based approach whilst Chapter 6 develops a method based on the systematic analysis of the possible situations [113].

Chapter 4	**System properties and hazards**	*Composition*	COM1 Requirements COM2 Protocol
		Controllability	COT1 Configuration COT2 Coordinator COT3 Initiator COT4 Executor COT5 Terminator
	Testing Dimensions	*Level*	Participant
		Feature	Flow
		Depth	Transition coverage criteria
	Test case generation		Structure-based testing
Chapter 5	**System properties and hazards**	*Dependency*	DEP1 Order DEP2 Relationship
	Testing Dimensions	*Level*	Transaction
		Feature	Flow
		Depth	Action and condition coverage criteria
	Test case generation		Control-flow based criteria
Chapter 6	**System properties and hazards**	*Dependency*	DEP1 Order DEP2 Relationship DEP3 Data
	Testing Dimensions	*Level*	Transaction
		Feature	Flow, Data, Control
		Depth	Combination criteria
	Test case generation		Classification-Tree

Table 3.4. Summary of hazards addressed

Chapter 4

Testing at participant level

Size matters not. Look at me. Judge me by my size, do you?

Yoda, Star Wars

This chapter presents an in-depth study of the issue of testing at participant level of a WS transaction. It focuses mainly on the executor role, as was defined in the F2T presented in the previous chapter. The executors are the services in charge of executing the activities that compose the transaction. In this chapter the Abstract Transaction Model (AbTM, Section 3.1) is used to derive concrete models in order to automatically generate test cases for different WS transactions standards.

The first part of this chapter presents the proposed testing process. The second part evaluates the approach using a case of the Jboss Transaction. The evaluation shows that the proposed system has the capability to automatically generate test cases and detect possible failures of executors running under different web services transactions standards.

4.1. Introduction

The Framework for Testing Transaction (F2T) (section 3.2) identified the behaviour of each executor (*Composition* property) and their coordination (*Controllability* property) as testing targets: the executors are in charge of performing the activities that compose a WS transaction, while the coordinator manages their collaboration in order to achieve an agreed outcomes of a WS transaction.

The hazards identified for the Composition property in Chapter 3 suggests defining tests for checking the functional behaviour of the service (hazard COM1) and the compliance of the transaction model used to manage the transaction (COM2). In the same way, the hazards of the Controllability property shows the need of defining test case for checking the configuration of the transaction (COT1), but also the behaviour of the rest involved roles (COT2, COT3, COT4, COT5).

As WS transactions do not have a homogeneous transaction model, the process of testing the participants of WS transaction is more complicated. We therefore use the proposed AbTM to encode intended behaviour of the transaction. The focus of testing in this chapter is to detect possible faults or failures in WS transactions participants running under different models or standards (e.g., BTP [36], WS-BA [41]). The main goal is the executor role since they are in charge of executing the activities that actually forms the transaction. The objective is to identify the observable differences between the behaviours of implementation and what is expected on the basis of specification of WS transaction models and standards.

Table 4.1 shows how the goals of this chapter fit into the F2T presented in Section 3.2. The test case design process of this chapter focus on the behaviour of the participant, so we address the issue of testing at *participant level*. This includes testing the behaviour of executor and coordinator roles. We use the part of the Executor Model (EM) that captures the executor´s behaviour. This model defines the executor life-cycle in terms of states and transitions which relates to the *flow feature*. The

criteria for managing (*depth* dimension) the test effort use the structure-based testing techniques.

System properties and hazards	*Composition*	COM1 Requirements COM2 Protocol
	Controllability	COT1 Configuration COT2 Coordinator COT3 Initiator COT4 Executor COT5 Terminator
Testing Dimensions	*Level*	Participant
	Feature	Flow
	Depth	Transition coverage criteria
Test case generation		Structure-based testing

Table 4.1. Relationship of chapter 4 with F2T

The first part of this chapter (Section 4.2) specifies the model used to achieve the concrete test method for participants in WS transactions. The method relies on the AbTM to capture the behaviour of the transaction independently of the protocol used. So firstly we show how the AbTM can model existing WS transaction standards. We then define a test process using the abilities of the AbTM. The method allows automatically defining test cases but also compares the actual system outcome with the expected system outcome.

The second part of this chapter (Section 4.3) presents the implementation and validation of the proposed method. A prototype tool has been developed to implement the method using a case study.

4.2. Specification and theoretical model

This section specifies the theoretical model proposed to achieve the testing method. The method uses the AbTM´s abilities of modelling existing WS transactions standards in order to define a generic testing method. First we show how the AbTM models the BTP and WS-BA standards given that

these are the most widely accepted standards in WS transactions. We then present the generic testing process for WS transactions at participant level.

4.2.1. Modelling the WS transaction standards

This section shows how the BTP and WS-BA transaction standards can be modelled using the proposed model. The modelling process is composed of the following activities:

1. *Role identification and modelling*: it identifies the roles of participants in a target WS transaction standard and models them using the roles defined in the abstract transaction model.

2. *State transitioning*: it captures the important states of a target WS transaction standard and maps them to the state transitions of the abstract transaction model.

3. *Messages mapping*: it maps the messages between AbTM and a specific WS transaction standard.

Modelling of BTP

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

Roles identification and modelling

This activity models the roles of the BTP participants involved in executing wT and its activities. BTP implements nested transaction model, that is, a parent transaction, wT, is composed of activities, and each activity can be another wT itself (subtransactions). BTP defines a dependency called *superior:inferior* between the parent (*superior*) and its activities (*inferiors*). The *superior* makes the decision and the *inferior* abides such decision in order to complete the transaction. That *superior:inferior* relationship can be

recursively extended to define a transaction tree having intermediates nodes as superior and inferior. For example, the process of booking a holiday package (flight, hotel and car rental reservations). The *holiday package reservation* would take the role of superior and each reservation (flight, hotel and car) would be the *inferiors*. Let assume that the hotel reservation is provided by a service that manage many hotel chains. The service would have to do some actions such as find a suitable hotel, make a pre-booking and sent information to the customer. So the *hotel reservation* activity takes the role of superior of another wT composed by the activities (*inferiors*) *find*, *pre-booking* and *information*.

Figure 4.1 depicts the modelling of BTP using the Abstract Transaction Model (AbTM). Figure 4.1 (a) represents the BTP coordination of wT and its activities using the *superior:inferior* relationship, and (b) represents the coordination of the same wT using the AbTM. The superior of BTP is modelled as *Initiator* in AbTM since it starts the process. Also the superior can be modelled as *Coordinator* and *Terminator* as it decides on the outcome of the activities. *Inferior* (in BTP) executes a concrete activity and is therefore modelled as *Executor* in the AbTM.

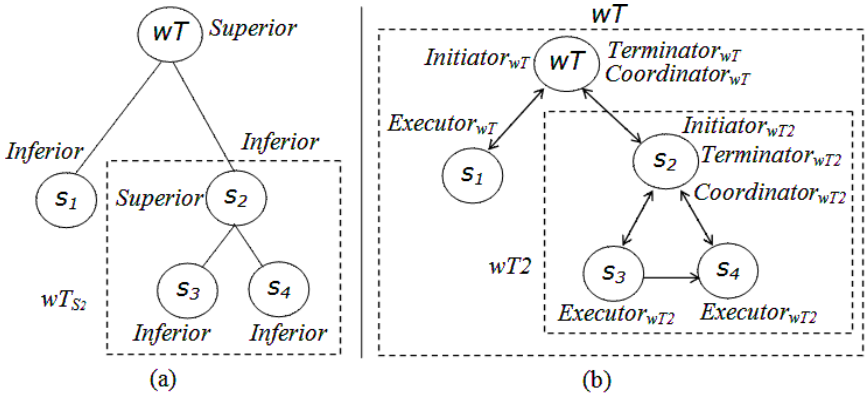


Figure 4.1. BTP relationship modelling

State transitioning

The Executor Model (Figure 3.5) and the Coordinator Model (Figure 3.6) are used to model the BTP (as well as WS-BA) states and transitions. When a wT is started at the initiative of an initiator it causes the creation of

a context for a new transaction. The role of initiator has finished its life-cycle so the initiator moves from START state to FINISH state through *creation* (see Figure 3.).

After receiving a context creation request from an initiator, the coordinator replies the context (it moves from INITIAL state to ACTIVE state). Executors receive a context, enrol with the Coordinator and are already active in that transaction so they move from READY to ACTIVE state. Coordinator moves to PREPARE state awaiting decisions from Executors.

Once an Executor (*inferior* in the BTP model) has processed its activity, it moves to COMPLETED. The Executor sends its outcome to Coordinator and moves to DECISION state. The Coordinator collects the outcomes from all Executors and takes the final decision. It moves from PREPARE state to DECISION state. Note that in the BTP model, the terminator role is taken by the coordinator as was previously commented.

The final decision is sent to each Executor and the Coordinator then moves to CONFIRM state. Executor sends acknowledgement and changes its state to END state through the transition (either completed rollback or completed successfully) according to the decision. Once the Coordinator has received all confirmation, it moves to END state. Note that an Executor can leave the wT before confirming the activity. So it can move from ACTIVE state to CANCEL state.

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

Message syntax

Table 4.2 presents the mapping of messages from the abstract transitions to BTP specific message syntax. The table shows that the abstract model captures all the messages required to complete a transaction using BTP.

Abstract model	**BTP**
Creation	Initiator sends BEGIN to coordinator.
Created	Coordinator sends BEGUN to initiator.
Setup	Initiator sends the context to the executors
Execution	Executor sends ENROL to coordinator. It responses with ENROLLED. If the executor is a superior of a new wT, it response with CONTEXT_REPLY.
Local committed	Coordinator sends PREPARE to executor. Due a protocol optimization, this transition could be omitted.
Local completed	Executor sends PREPARED to coordinator
Local cancel	Executor sends CANCEL to coordinator
Completed successfully	Coordinator sends CONFIRM to executor and it responses with CONFIRMED.
Completed rollback	Coordinator sends CANCEL to executor and it responses with CANCELLED.
Preparing	It receives CONFIRM_TRANSACTION from the terminator and sends PREPARE to all executors.
General decision	Coordinator receives all executor messages
Global verdict	Coordinator sends the suitable message for each executor.
Close	The coordinator receives all the responses from executors and sends TRANSACTION_CONFIRMED/ TRANSACTION_CANCELLED to initiator.
Cancel	Executor sends RESIGN to coordinator.

Ended cancelled	Coordinator sends RESIGNED to executor.
Completed rollback	Coordinator wants confirm but there is a contradiction. Coordinator sends CONTRADICTION to executor, and/or executor sends HAZARD to coordinator.
Completed pivot	Coordinator cancels but there is a contradiction. Coordinator sends CONTRADICTION to executor, and/or executor sends HAZARD to coordinator.
Processing failure	The executor is not working. Coordinator knows it by receiving a FAIL message or due to a timeout (non response message).
Ended replaceablity	Coordinator sends REDIRECT with the address of the new executor.

Table 4.2. BTP message mapping

Modelling of WS-BA

Web Service Business Activity (WS-BA) manages transactions that apply compensations to handle exceptions which occur during the execution of activities. Compensation is an activity that semantically undoes the work performed by other completed activity. In the example of the *holiday package reservation*, if the holidays are cancelled, the flight company, for example, would execute a compensation that would set as available the booked seat and would refund to the customer part of the paid money.

WS-BA works with WS-COOR standard protocol to coordinate the transactions. WS-BA supports two coordination types, *MixedOutcome*, and *AtomicOutcome*, and two protocol types. *MixedOutcome* allows each activity to achieve a specific outcome while *AtomicOutcome* requires all the activities to finish in the same way. In *holiday package reservation* example, with *AtomicOutcome* type either all reservations are confirmed or any on them is confirmed. On the other hand, *MixedOutcome* would allow confirming the reservation of the holiday package even if, for example, the rent car reservation cannot be done. The protocols types differ according to the participant's roles in processing activities. In other words, who has the ultimate say about the state of an activity; Executor (*BusinessAgreementWithParticipantCompletion*, *BAWPC*) or Coordinator (*BusinessAgreementWithCoordinatorCompletion*, *BAWCC*).

Roles identification

Figure 4.2 depicts the modelling of WS-BA using the abstract transaction model. Figure 4.2 (a) shows the *AtomicOutcome* protocol, whilst (b) shows *MixedOutcome* protocol. In both protocols the role of Initiator is taken by the first participant who interacts with a Coordinator. In *AtomicOutcome* the role of Terminator is taken by the Coordinator. This is due to the fact that coordinator is the participant that knows all Executors's output and, therefore, it knows the final outcome: close or terminate if all executors have successfully executed their activities, or compensate otherwise. In *MixedOutcome*, the Initiator is the Terminator since each Executor may have its specific decision so the outcome depends on the business logic.

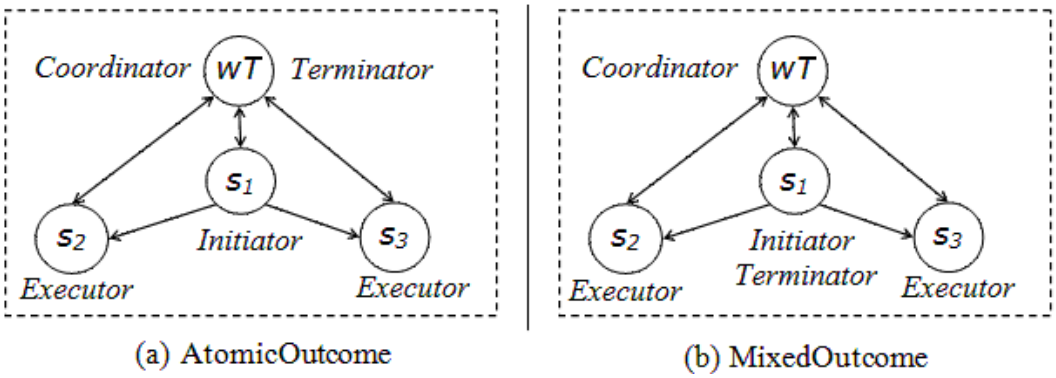


Figure 4.2. WS-BA relationship modelling

State transitioning

Similar to BTP, it is used the Executor Model (Figure 3.5) and the Coordinator Model (Figure 3.6) to model the WS-BA. The Initiator creates a new transaction by requesting a context to the transaction. The executor finishes its life-cycle (it moves from START to FINISH). The Coordinator responds with a context (from INITIAL to ACTIVE state). That context is sent to the Executors by the Initiator.

The Executors join the current wT and it is modelled as the transition from READY to ACTIVE state. Once each Executor makes a decision, it

moves from ACTIVE to COMPLETED state and the Coordinator moves from ACTIVE to PREPARE state.

When the transaction is *MixedOutcome*, the decision for each activity is taken alone. The Coordinator moves from PREPARE to DECISION state when it receives an Executor's notification. The Coordinator decides about its outcome and moves from DECISION to CONFIRM. The Coordinator receives the confirmation and goes back to wait for the rest of Executor's notifications (from CONFIRM to ACTIVE state).

In the A*tomicOutcome* type, the Coordinator moves from PREPARE to DECISION state when it has a global outcome about the transaction. The Coordinator then sends the global decision and moves from DECISION to CONFRIM state. Finally it waits for the confirmations and moves to END state. When an Executor is not able to start executing its activity it moves from READY to ABORTED state. If the activity was cancelled while it was under execution, the Executor moves from ACTIVE to CANCELLED state. In case of failure it moves from ACTIVE to FAILED state.

Messages mapping

Table 4.3 presents the transformation from the abstract transitions to WS-BA specific message syntax.

Abstract model	**WS-BA**
Creation	Initiator sends CREATECOORDINATIONCONTEXT to coordinator.
Created	Coordinator sends CREATECOORDINATIONCONTEXTRESPONSE to initiator. The initiator sends the context for the executors.
Setup	Initiator sends the context to the participants
Execution	Each executor, after receiving the context, sends a REGISTER message to its chosen coordinator. The coordinator responses with a REGISTERRESPONSE message.
Local committed	If the coordination type is BAWCC, coordinator sends COMPLETE to executor. In the other coordination type this transition is omitted.
Local completed	Executor sends COMPLETED to the coordinator.

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

Table 4.3. WS-BA message mapping

4.2.2. Test design and execution process

In general, testing aims at showing that the intended and actual behaviours of a system differ, or at gaining confidence that they do not. The main goal of testing is failure detection, i.e., in the current scope, the

observable differences between the behaviours of implementation and what is expected on the basis of the specifications of WS transaction standards. We exploit the model-based testing approach that encodes the intended behaviour of a system and the behaviour of its environment. Model-based testing is capable of generating suitable test cases and it has also been successfully used in others WS domains [68]. Specifically, we use the structure elements of such models, so it fit in the category of structure-based testing [93]. We have designed a test process which comprises test design, test implementation, test execution and outcome evaluation. These phases are depicted in Figure 4.3.

Definitions

The abstract model can be used to generate test cases for different WS Transactions. The test basis used is the abstract model of executor and coordinator, therefore, the *level dimension* is the participant. Since the model captures the behaviour of the participants in terms of flow of states/transitions, the *feature dimension* used is the flow. Also as the model is based on states and transitions, in order to manage the *depth dimension*, we propose the well known family of transition coverage criteria [114]. By applying a test criterion over the proposed model, AbTM, we obtain a set of abstract test cases. Each abstract test case is mapped to a concrete test case which is composed by the test scenario and the expected system outcome. The concepts used in this test process are defined as follows.

- *Transition coverage criterion*: The set of test cases must include tests that cause every transition between states in a state-based model.
- *Abstract test case*: A sequence of states and transitions of a participant using the abstract transaction model. The notation $S_i \xrightarrow{t} S'_i$ is used to denote that the participant p_i changes its current state S to S' executing the transition labelled, t. If the participant is the Coordinator, it is denoted by K. We use $S^a{}_i \xrightarrow{t^x} S^b{}_i$ – … – $S^c{}_i \xrightarrow{t^y} S^d{}_i$ to denote a sequence of states/transitions.

- *Test scenario*: A sequence of actions in a human-understandable way to provide guidance to the tester to execute a test case.
- *System outcome*: The internal state of the process defined by a sequence of exchanged messages between participants using a specific WS Transaction standard. The notation $i[m_1]j$ is used to denote that the participant p_i sends message m_1 to participant p_j. We use $i\ [m_1]j - l[m_2]o - \ldots - v\ [m_n]z$ to denote a sequence of messages.

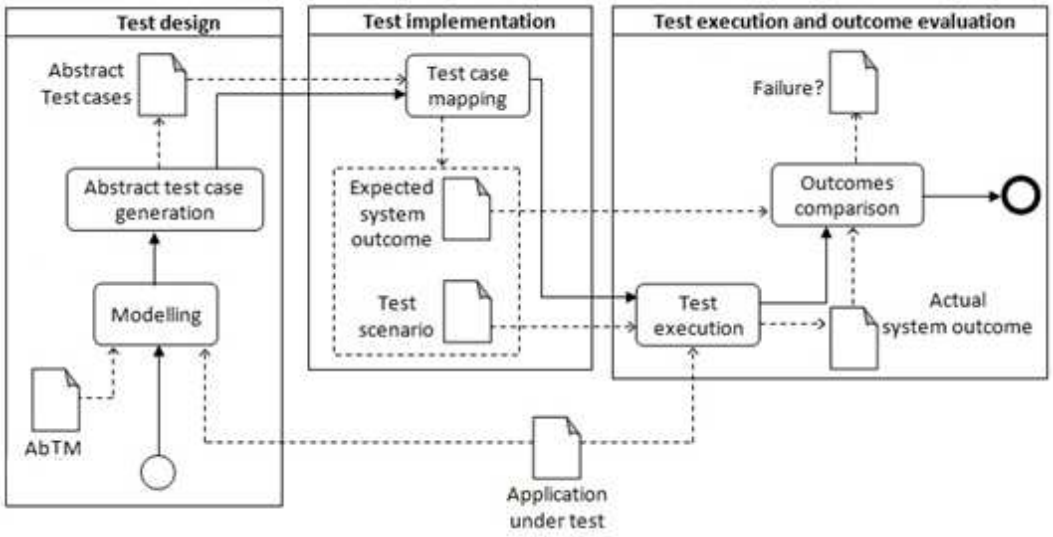


Figure 4.3. Test process using the AbTM

Test design

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

Test implementation

The sequence of states and transitions specified by the abstract test cases generated in the test design phase are mapped to a specific WS

transaction standard as is shown in Section 4.2.1. As discussed above the proposed AbTM has the ability to capture the behaviour of a WS transaction standard as well as mapping the abstract cases to a specific WS transaction standard. These features provide the capability of automatically obtaining the test scenario and the expected system output.

Test execution and outcome evaluation

Once the test cases are implemented, they are executed over the system under test (i.e. an application that uses a specific WS transaction standard) and the actual outcome is obtained. Finally, for each test case, the expected outcome is compared to the actual outcome to find differences in behaviour and to detect failures. Two outcomes are considered: (i) *user outcome* refers to what the user perceives, for instance, to reserve theatre tickets whether the number of booked tickets is correct. (ii) *system outcome* refers to the non-visible process that the system has carried out to achieve the requirements - e.g., the correct exchange of messages between the services according to the transaction standard.

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

4.3. Implementation and validation

This section explains the implementation of the proposed test process described in Section 4.2.2. A prototype system has been developed to evaluate the method through a case study: the JBoss Night Out open source application.

4.3.1. Prototype system

We have developed a prototype system that implements the main phases of the proposed test method (Figure 4.3)

- *Modelling*: the prototype system prompts the tester to provide information (e.g. services, roles, transaction standard, etc) and to create the WS transaction.
- *Abstract test case generation*: the abstract test cases for all the participants (Coordinator, Executor, etc) are automatically generated by the prototype system.
- *Test case mapping:* abstract test cases are mapped to WS transaction standards (e.g., BTP or WS-BA). That is, the prototype system automatically generates the concrete test cases (for each WS transaction standards) which are composed of the test scenario and the expected system outcome. A test scenario is defined as a sequence of actions in a human-readable way to provide guidance to the tester to execute a test case.
- *Outcomes comparison:* test cases are executed in order to produce the actual system's outcome. The prototype system automatically compares the actual system's outcome with the expected system's outcome in order to detect any fault or failure.

The prototype system is implemented in Java 1.5. It includes three components: *Model*, *Tests* and *Outcome*. The *Model* implements the generic transaction model. It also includes a graphic interface to allow the tester to enter all the necessary information such about the system under test such as

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

4.3.2. Case study: Jboss Night Out

In order to evaluate the proposed testing approach, we utilise the *Night Out* case study of the Jboss WS-BA standard [115]. This section firstly describes the case study and its testing process. Then the result of their execution is discussed and an example of test case and detected failure is described for a better understanding.

***Night Out* specification**

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

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

to select the reservations and then book a night out by invoking each of the services within the scope of a WS transaction. For example, if seats are not available in the restaurant or the theatre, the taxi will not be necessary. Each service, exposed as Java API for XML Web Services (JAX-WS) [116] endpoint, has a GUI with state information and an event trace log. The application provides logs for step of its activity. As the transaction proceeds, each of the WS pops up a window of its own in which its state and activity log can be seen. Some events in the service code are also logged.

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

Test design

The transactional process included in the *Night Out* application has been modelled according to the roles identified in the AbTM as is shown in Figure 4.4. *Night Out* (client side) takes the role of Initiator since it starts the transaction and asks the other web services to participate. *Restaurant*, *Theatre* and *Taxi* services are modelled as Executors since they execute a specific activity. The role of Terminator is taken by the *Night Out* application since some activities (e.g. *Theatre*) are independent of others services (e.g. *Restaurant*). Thus even if one service can not complete its action the others are allowed to commit. The *Taxi* activity is dependent. For instance, if a table is not available in the restaurant, the customer still needs a taxi to go to the theatre. The role of Coordinator is taken by an external service, *WSCoor11*, provided by the server. It follows the WS-COO [39] and WS-BA[41] standards to exchange suitable messages.

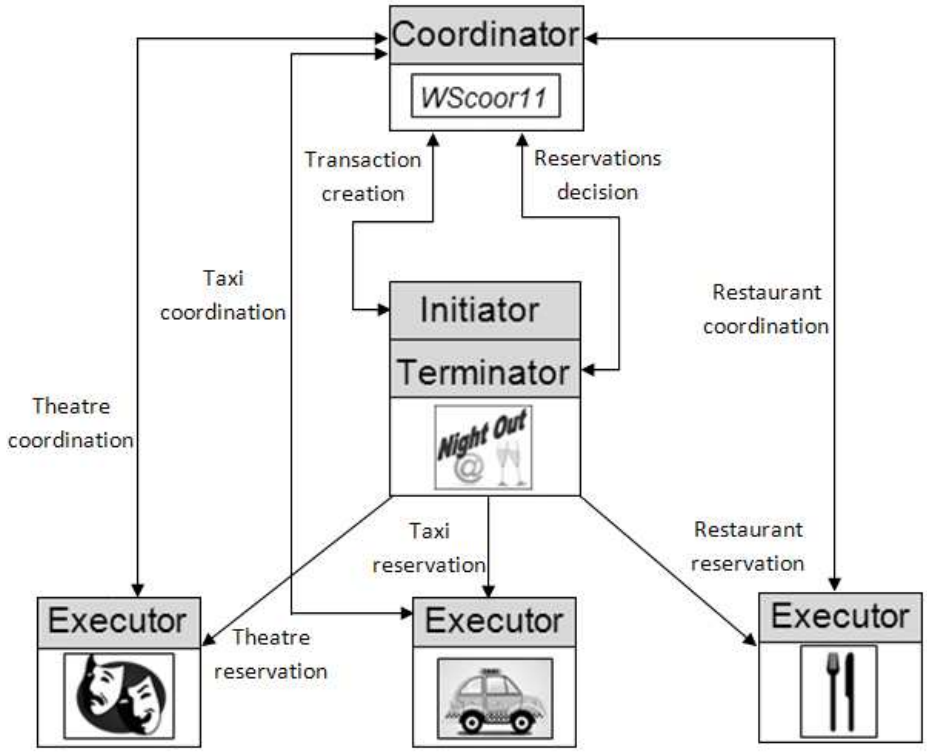


Figure 4.4. ***Night Out*** **case study modeling**

Test implementation

This phase generates various abstract cases for each Executor, i.e., *Restaurant*, *Theatre* and *Taxi*. According to testing approach explained in Section 4.2.2, eight abstract test cases are generated for each Executor. Those abstract test cases were automatically mapped to generate test cases, i.e., the test scenario and the expected system outcome for *Restaurant*, *Theatre* and *Taxi* services.

Table 4.4 contains eight test cases for the Restaurant, Theatre and Taxi. Res_1, Thr_1, and Tax_1 respectively mean test case 1 for Restaurant, Theatre and Taxi services. Res_2, Thr_2, and Tax_2 mean test case 2 and so on.

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

Table 4.4. Test cases for *Night Out* services

Test execution and outcome evaluation

The generated test cases have been executed over the case study and Table 4.5 summarizes the results. 'Pass' means that a test case did not detect any failure. 'Fails' means that the actual outcome differs from the expected outcome (i.e. a failure has been detected). 'Blocked' means that a test case cannot be executed the application does not have the interface to achieve the required actions. In this section we use a number to identify each test case according to the Table 4.4. For each number there are actually three test cases, one for each executor (e.g. Rest_3, The_3, Tax_3).

Two of the designed test cases were blocked due to the following reasons; test case 1 requires cancelling the activity (`Cancel` message) once the Executor has started and has not finished yet, but the application does not allow cancelling a booking. Test case 8 defines a scenario where the Executor is not able to complete its activity (`CanNotComplete` message), but retried executing its action. The application neither allows resending the data nor registering again the Executor without starting a new transaction.

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

Table 4.5. Tests execution results

The test case 5 detected an important transaction-related failure in the compensation process. This test case and the detected failure are further explained below. During the execution of test cases 3 and 4 interface-related failures were detected: the application, which shall allow changing manually the capacity of each resource (i.e. number of tables and number of seats in the theatre), either crashes or does not update the capacity when the button is pressed.

A test case in detail

As an example of test case and detected failure we consider the test case Thr_5 generated using the following abstract test which was obtained applying the transition coverage criterion over the executor abstract model:

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

The abstract test case was mapped (see Section 4.2.1) to a specific sequence of WS-BA message as depicted in Figure 4.5. From this sequence of messages, our prototype system automatically generates the test scenario shown in Figure 4.6. Note that the transaction creation and participant register processes are defined by the Initiator as was shows in Figure 3.2 (*creation* and *setup* transitions).

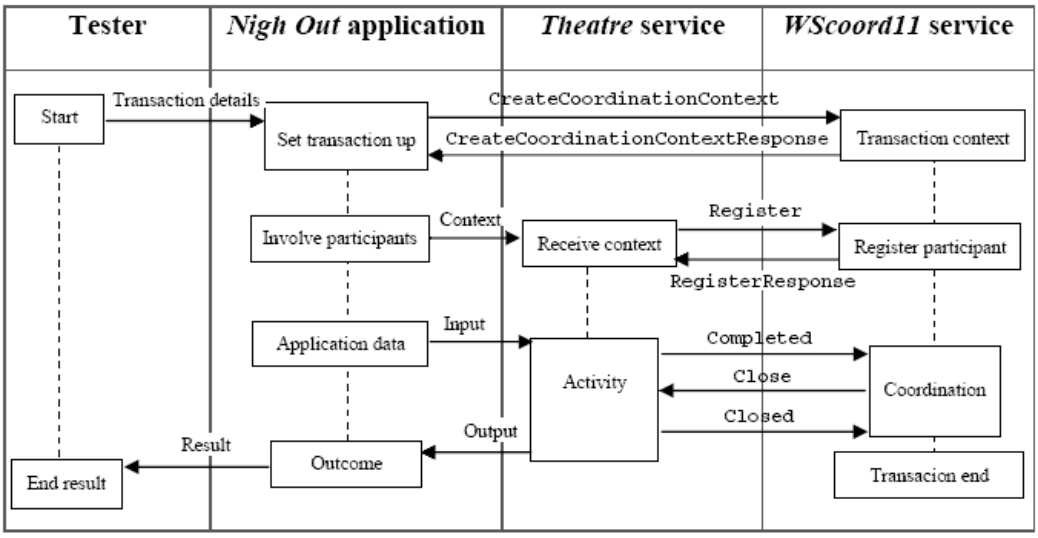


Figure 4.5. Sequence diagram of a test scenario for *Theatre* service

STEP 1: *NighOut* starts the process. It sends a context request (`CreateCoordinationContext` message) to the coordinator *WScorr11*
STEP 2: WScorr11 sends the transaction context (`CreationCoordinationContextResponse` message) to *NighOut*
STEP 3: *Theatre* receives a transaction context from the initiator *NighOut*
STEP 4: *Theatre* accepts to participate in the process. It requests to be registered in the transaction, thus it sends `Register` message to *WScorr11*
STEP 5: *WScorr11* receives Register message from *Theatre* and registers *Theatre* in the transaction. It sends `RegisterResponse` message to *Theatre*
STEP 6: *NighOut* sends the application data to *Theatre*
STEP 7: *Theatre* completes successfully its activity. *Theatre* sends `Completed` message to notifies its outcome to the coordinator *WScorr11*
STEP 8: Theatre has successfully completed its activity. *Theatre* notifies the results and leaves the transaction. *Theatre* sends `Close` message to notifies to the coordinator *WScorr11* and it responses with a `Closed` message

Figure 4.6. Test scenario for test case Thr_5

As described in Table 4.4, the goal of the test case Thr_5 is to successfully confirm the theatre tickets booking when the other service reservation has been undone through compensating transaction. As commented in the case study specification, the theatre service is independent of the restaurant service, so if the restaurant reservation is cancelled, the theatre tickets booking have not to be compensated.

After the execution of the test case, we obtain the expected system outcome. By comparing the expected system outcome and the actual system outcome, a failure is detected by the prototype system. This is shown in the code snippet in Figure 4.7. The expected system outcome requires receiving a CLOSE message once the *Theatre* service has successfully completed its activity (see sequence diagram in Figure 4.5). However, the actual outcome has a COMPENSATED message since *Restaurant* service was not able to commit. As a result, the *Theatre* reservations were automatically undone. The fault which causes such failure is also found by the prototype system as there is a difference (or discrepancy9 in the 'Register' message – the way *Theatre* service is registered in the *Night Out* under the WS-BA specification. That is, the application registers the *Theatre* service as an atomic outcome when a mixed outcome was expected (Figure 4.8). In other words, if *Taxi* or *Restaurant* services are not able to make their reservations, the *Theatre* service will automatically undo the reservation even if the customer would wish to keep the theatre tickets.

```
<soap:Envelope
xmlns:soap="http://schemas.xmls
oap.org/soap/envelope/">
<soap:Header>
<Action
xmlns="http://www.w3.org/2005/0
8/addressing">
http://docs.oasis-open.org/ws-
tx/wsba/2006/06/Close
</Action>
```

(a) Expected outcome

```
<soap:Envelope
xmlns:soap="http://schemas.xm
lsoap.org/soap/envelope/">
<soap:Header>
<Action
xmlns="http://www.w3.org/2005
/08/addressing">
      http://docs.oasis-
open.org/ws-
tx/wsba/2006/06/Compensate
</Action>
```

(b) Actual outcome

Figure 4.7. Fault in message exchange

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

Figure 4.8. Fault in registration process

The results obtained from the test comparison are also useful for a debugging process. In the above tests, the faults mean that the transaction was not correctly configured or coded. This can help in identifying the faults in the code. For example, the above fault was found in *BasicClient.java* file, at line number 496 in the code shown in Figure 4.9. The configuration of the transaction is made using the class *UserBusinessActivityImple*, through the factory pattern using *UserBusinessActivityFactory* class. By looking at the implementation of that class we found (in Figure 4.10) that the transaction is defined as an *AtomicOutcome*.

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

   UserBusinessActivity uba =
   UserBusinessActivityFactory.userBusinessActivity();
```

Figure 4.9. Fault identification: transaction setup

```
public class UserBusinessActivityImple
extends UserBusinessActivity
{

  public void begin(int timeout) throws
  WrongStateException,SystemException
  {

    try {

      if (_contextManager.currentTransaction() != null)

          throw new WrongStateException();

      CoordinationContextType ctx = _factory.create(
      BusinessActivityConstants.WSBA_PROTOCOL_ATOMIC_OUTCOME,
      null, null);
```

Figure 4.10. Fault identification: protocol implementation

4.4. Summary

This chapter investigated into the issue of testing WS transaction at the *participant level*. In it we developed and evaluated the coordinator and executor roles in the Abstract Transaction Model which is capable of dynamically modelling different WS transaction models and standards. The model exploits structure-based testing technique in order to automatically generate test cases for testing the reliability of participants running under WS transaction standards such as BTP and WS-BA. The proposed test process is implemented as a prototype system with which various test cases were automatically generated and mapped to different standards. The evaluation was performed using the case study of *Nigh Out*, which is an open source application provided by Jboss [117]. The experiments show that our approach can effectively be used to define different test cases for the executor level as well as test the reliability (or failure detection) of different WS transactions standards.

This chapter was focused on the behaviour of each executor and their coordination. Therefore, it addressed the testing goals derived from the

Composition and *Controllability* properties defined in F2T. Another property related to the executors is *Dependency* that refers to the relationships between the different executors. That property and its testing goals are addressed in the next chapters.

Chapter 5

Testing at transaction level: control-flow based approach

There is no dependence that can be sure but a dependence upon one's self

John Gay

This chapter presents a control-flow based approach to address the hazards indentified in the Dependency property. The method is focused on the flow feature – as described in the Framework for Testing Transactions (F2T), Chapter 3.

Firstly the chapter describes how the dependencies between activities can be defined in terms of task relationships. Using the logical expression derived from those relationships, the chapter later presents a new family of test criteria to exercise the implementation of the dependencies.

In order to show the viability of the proposed method, it is applied to an example used in the literature. Furthermore, the chapter presents a real feedback of the method through a real industrial case study.

5.1. Introduction

As defined in the AbTM (Section 3.1), a WS transaction comprises a group of a smaller and partially independent activities executed by different web services. To manage the execution of the various activities, a set of dependencies (relationships) are specified among them. Dependencies are constraints on the processing produced by the concurrent execution of activities. Activities dependencies represent a key component in ensuring the flexibility required to support exceptions, alternatives, compensations and so son, which all are the basis of the Advanced Transaction Models [19].

In Section 3.2.4, the Framework for Testing Transactions (F2T) identified the relations between the activities involved in a WS transaction (*Dependency* property) as testing target. Dependency property refers to the execution of the activities under certain constraints (such as order of execution, hazard DEP11) in order to maintain application reliability, correctness and data consistency (DEP2 and DEP3). This chapter addresses such property. It focuses on the overall transaction level (or composite web service level) and it takes into account the relationships between the different activities of WS transaction. To deal with these test goals, this thesis presents two different approaches. In this chapter we define the flow related dependencies using logical expressions. Test cases are generated using control-flow based techniques. Chapter 6 presents a different approach to address the hazards: we analyze a set of widely used high level dependencies using the Classification Tree approach [118].

Table 5.1 shows how the goals of this chapter fit in the F2T. As stated above, this chapter focuses on the *transaction level* according the levels identified in F2T. The control-flow based approach presented here uses the flow of execution of the whole transaction, so the *flow feature* is used to derive the test cases. This approach, therefore, addresses the hazards DEP1 and DEP2. Note that since the data is not taking into account, the hazard DEP3 is no addressed in this chapter. The test effort (depth dimension) is managed by the proposed new family of test criteria based on control-flow testing techniques.

System property and hazards	*Dependency*	DEP1 Order DEP2 Relationship
Testing Dimensions	*Level*	Transaction
	Feature	Flow
	Depth	Action and condition coverage criteria
Test case generation		Control-flow based criteria

Table 5.1. Relationship of chapter 5 with F2T

5.2. Flow definition

This section presents the proposed method to define the dependencies between activities in terms of flow of tasks. This flow definition will be the basis for the test case generation explained in Section 5.3.

According to the AbTM (Section 3.1), a web service transaction is defined as $wT =< A, D >$ where $A = \{a_1, \dots, a_n\}$ is a set of activities and $D = \{d(a_b, a_c)_1, \dots, d(a_w, a_z)_m\}$ is a set of dependencies between the activities.

The Executor Model (EM, Figure 3.4) defined the set of possible actions for an executor as *begin*, *complete*, *compensate*, *A-withdraw*, *A-cancel* and *A-fail.* These actions can be classified in three primitive tasks according to its semantic (meaning) [119, 120]. Table 5.2 shows these tasks. The primitive tasks are the lowest level of granularity in which a dependency can be defined. Note that according to this classification, an activity is not compensated by itself, instead the compensation is modelled as a related activity that commit once the former is aborted.

Primitive Task	**Action**
Begin	Begin
Commit	Complete (Compensate)
Abort	A-withdraw A-cancel A-fail

Table 5.2. Actions categories

In this way, any activity a_i has three (primitive) tasks that we assume are atomically executed:

- $B(a_i)$: The activity a_i begins executing.
- $C(a_i)$: The activity a_i successfully commits.
- $A(a_i)$: The activity a_i aborts.

An abortion may occur due to either a fault during the execution or an explicit cancellation. When an activity aborts, its compensatory action will be executed if it exists. The compensatory action is defined as another activity part of the same wT. The original activity and their compensatory action are, therefore, related by concrete dependencies as is shown later.

5.2.1. Dependencies

Each dependency $d(a_x, a_y)$ defines a relationship between two activities as_x and a_y. The formal definition of the possible dependencies is presented below. The dependencies are divided in three groups (necessary, sufficient, and composite) according to their constraints:

Necessary conditions dependencies: In order to be able to execute any task P, an activity a_y may require the execution of other task Q of an activity a_x. So a_y cannot execute P until a_x has executed Q. Formally, $P(a_y) \Rightarrow Q(a_x) < P(a_y)$. These dependencies are labelled as $abc - on - xyz$ (abbreviated as *ax*) where $abc, xyz \in \{begin, commit, abort\}$. Due to there are three different tasks and all combinations are possible, nine dependencies are defined as is shown in Table 5.3. For example *begin-on-begin* dependency, $bb(a_x, a_y)$, specifies that the beginning of a_x is a necessary condition to enable the beginning of a_y.

Sufficient conditions dependencies. The execution of any task P of an activity a_x may force the execution of another primitive task Q of an activity a_y. So if a_x executes P, then a_y also executes Q. Formally, $\mathrm{P}(a_x) \Rightarrow Q(a_y)$. These dependencies are labelled as $force\ abc - on - xyz$ (abbreviated as *fax*). The nine possible dependencies of this kind are

presented in Table 5.4. For example *force begin-on-abort* dependency, $fba(a_x, a_y)$, defines that if a_x abort then a_y has to begin.

Composite dependencies. This group is composed by the dependencies where more than one relationship are taken into account. They are shown in Table 5.5.

	Begin	**Commit**	**Abort**
Begin	$bb(a_x, a_y)$	$bc(a_x, a_y)$	$ba(a_x, a_y)$
Commit	$cb(a_x, a_y)$	$cc(a_x, a_y)$	$ca(a_x, a_y)$
Abort	$ab(a_x, a_y)$	$ac(a_x, a_y)$	$aa(a_x, a_y)$

Table 5.3. Necessary conditions dependencies

	Begin	**Commit**	**Abort**
Begin	$fbb(a_x, a_y)$	$fbc(a_x, a_y)$	$fba(a_x, a_y)$
Commit	$fcb(a_x, a_y)$	$fcc(a_x, a_y)$	$fca(a_x, a_y)$
Abort	$fab(a_x, a_y)$	$fac(a_x, a_y)$	$faa(a_x, a_y)$

Table 5.4. Sufficent conditions dependencies

Name	Description	Definition	Example
Weak commit dependency, $wc(a_x, a_y)$	If both a_x and a_y commit, then the commitment of a_x precedes the commitment of a_y.	$C(a_x) \Rightarrow \{C(a_y) \Rightarrow [C(a_x) < C(a_y)]\}$	If a paper is accepted in a conference then it was sent before the deadline
Weak abort dependency, $wa(a_x, a_y)$	If a_x aborts and a_y has not been committed, then a_y aborts	$A(a_x) \Rightarrow \{\neg[C(a_y) < Aax \Rightarrow Aay$	If the user cancels the information request process, the query is not sent to the database
Termination dependency, $t(a_x, a_y)$	a_y cannot commit or abort until a_x either commits or aborts	$C(a_y) \vee A(a_y) \Rightarrow C(a_x) \vee A(a_x)$	The final outcome of a process cannot be sent until other process has finished
Exclusion dependency, $e(a_x, a_y)$	Only one of both a_x and a_y can commit	$[C(a_x) \Rightarrow A(a_y)] \wedge [C(a_y) \Rightarrow A(a_x)]$	When two hotel providers have been queried, only one can confirm the reservation.
Strong exclusion dependency, s$e(a_x, a_y)$	One of both a_x and a_y must commit	$[A(a_x) \Rightarrow C(a_y)] \wedge [A(a_y) \Rightarrow C(a_x)]$	If there are two possible means of transport, one of them has to be booked for finishing the travel reservation

Table 5.5. Composite dependencies

5.2.2. Modelling wT using dependencies

Using the above dependencies we can define aspects related to the management of the transactional process. A *compensatory action* associated to an activity is defined as two dependencies fca and ba. A a_x *replaceable* by a_y can be defined as a dependency $e(a_x, a_y)$, $se(a_x, a_y)$ or a combination of both, depending of the specific context.

Control flow patterns [61], such as *AND-join*, *AND-split*, *OR-join*, *XOR-split, parallel-overlapping, parallel-including* and so on, can be modelled with these dependencies.

AND-join pattern defines that a group of activities have to execute a task before another(s) activity(s) can execute a task. Since it defines necessary conditions to execute a task related to the execution of others activities' task, it is modelled as a set of *necessary conditions dependencies*. For example $bc(a_x, a_z)$ and $bc(a_y, a_z)$ define a *AND-join* pattern between a_x, a_y, a_z where the commitment of a_x, a_y is needed to begin a_z.

OR-join pattern defines a relationship between a group of activities, say a_x, a_y, and another one, say a_z. The execution of the task of any activity a_x, a_y is a sufficient condition to execute the task of a_z. So this pattern is modelled as two *sufficient conditions dependencies* $fbc(a_x, a_z)$ and $fbc(a_y, a_z)$

AND-split pattern defines that once an activity has executed a task, another(s) activity(s) can execute a task. A common use is the serial execution, defined as $bc(a_x, a_y)$, where the activity a_y has to wait until a_x has committed before it can begin.

XOR-split pattern defines a relationship between a group of activities, say a_x, a_y, and another one, say a_z. This relationship specifies that one and only one activity must commit in order to enable a_z to begin. According to the definition, *XOR-split* pattern is defined by a *composite dependency* $e(a_x, a_y)$ and two necessary conditions dependencies $fbc(a_x, a_z)$ and $fbc(a_y, a_z)$.

Two different activities, say a_x, a_y, follow the *parallel overlapping* pattern if and only if the begin of a_x precedes the begin of a_y, the begin of a_y precedes the commitment of a_x, and the commitment of a_x precedes the commitment of s_y. This pattern is defined as three dependencies (a_x, a_y) , $cb(a_y, a_x)$ and $cc(a_x, a_y)$. In a similar way, they follow the *parallel including* pattern if and only if the begin of a_x precedes the begin of $a_y$but the commitment of a_y precedes the commitment of a_x. This pattern is defined as two dependencies $bb(a_x, a_y)$ and $cc(a_y, a_x)$.

5.2.3. From a wT to tasks relationships

The dependencies involved in a wT can be specified in terms of tasks relationships. Let assume as example the WS transaction depicted in Figure 5.1.

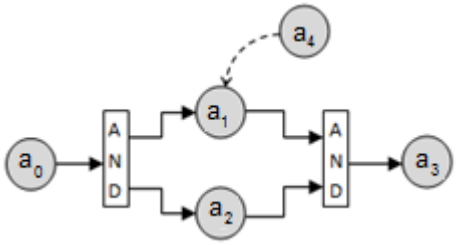


Figure 5.1. WS transaction example

The initial step is to define the activities involved in the process. According to the figure, we partially define the process as

$wT = \{A, D\}$, $S = \{a_0, a_1, a_2, a_3, a_4\}$.

The next step is to identify the control flow patterns (e.g. *AND-split*) and the transaction management aspects (e.g. *replaceable* activities). The example shows a workflow where a_0 is the first activity to be executed. When a_0 has committed, a_1 and $a_2$can begin (*AND-split*). Both a_1 and a_2 are required to commit before as_3 can begin (*AND-join*). If a_1 is aborted after it had committed, it is necessary to execute a_4 to undone its action (*compensatory action*, denoted by the broken line). Those relationships are modelled using the dependencies as had been shown before. So we define the set of dependencies as

$$D = \{bc(a_0, a_1), bc(a_0, a_2), bc(a_1, a_3), bc(a_2, a_3), fca(a_1, a_4),\ ba(a_1, a_4)\}$$

Logical conditions are specified by tailoring the dependencies. They define a logical expression that fire a task once it is evaluated as true. In other words, they specify a precondition to be enforced before the activities can execute the task. $BeginCond(a_i)$ defines the logical expression, derived from a_i´s dependencies, that controls the activity a_i beginning. It is structured as

$$BeginCond(a_i) = (N_1 \wedge \ldots \wedge N_i) \vee (S_1 \vee \ldots \vee S_j)$$

where N is a necessary condition and S a sufficient condition. In a similar way we can define $CommitCond(a_i)$ and $AbortCond(a_i)$. In this way, the last step in the transaction modelling is to define the $BeginCond$, $CommitCond$ and $AbortCond$ expressions for all the activities. To define those expressions is necessary to check all the dependencies where the task is involved. If the dependency defines a necessary condition, it will be added to the left part of the expression (N_{i+1} , linked by ∧). If it is a sufficient condition, it will be added to the right part of the expression (S_{j+1} , linked by ∨). The logical expressions for the example are presented in Table 5.6. The symbol * means that there are no conditions, in other words, the logical expression is always true.

	$BeginCond(a_i)$	$CommitCond(a_i)$	$AbortCond(a_i)$
a_0	*	*	*
a_1	$C(a_0)$	*	*
a_2	$C(a_0)$*	*	*
a_3	$C(a_1) \wedge C(a_2)$	*	*
a_4	$A(a_1)$	$A(a_1)$	*

Table 5.6. Logical expressions in the example

5.3. Test criteria

The goal of this section is to define test criteria for testing the dependencies based on the flow previously defined. We base our approach on the activities tasks relationships.

As described in Section 3.2.2, a test criterion is defined as a set of rules that impose test requirements and must be fulfilled by the test cases. A coverage criterion provides guidance for tests definition making this process more efficient and effective. Many test coverage criteria have been proposed such as path coverage, branch coverage, data flow coverage and so on [121]. These criteria are applied over some kind of model of the software under test. For example path coverage can be used on a graph that represents the

states and transitions of a software component. We define test criteria to be applied on the dependencies model explained in the previous section.

We propose a set of criteria based on two set of criteria: *task-based* and *conditions-based*. *Task-based* refers to the task(s) that are checked in the activities. *Conditions-based* refers to the criteria used to check the conditions that compose the logical expressions$BeginCond$, $CommitCond$ and $AbortCond$. Finally, these two primitive criteria are combined to define a family of test criteria.

5.3.1. Task-based criteria.

These criteria are regarding the activities tasks to be exercised. Three criteria are defined:

- *All-begin criterion (ABC):* All the activities must begin at least once.
- *All-commit criterion (ACC)*: All the activities must commit at least once.
- *All-commit-abort criterion (ACAC):* All the activities must commit and abort at least once.

ACC subsumes *ABC* since any activity needs to begin before committing. Obviously *ACAC* includes *ACC* and, therefore, also includes *ABC*. A more exhaustive criterion requires more primitive tasks to be executed and therefore, a higher testing effort.

A test suite T was defined as a set of test cases, $T = \{tc_1, \dots, tc_n\}$, where each tc_i is a test case that describes which tasks have to be executed (and which not) in an execution of a web transaction $wT = \{A, D\}$. We can formally define the previous criteria as follow:

T satisfies the *all-begin* criterion for wT if $\forall\, a_i \in A,\ \exists\, tc_j \in T / BeginCond(a_i) = true$.

T satisfies the *all-commit* criterion for wT if $\forall\, a_i \in A,\ \exists\, tc_j \in T / CommitCond(\mathrm{a}_i) = true$.

T satisfies the *all-commit-abort* criterion for wT if $\forall\, a_i \in A$, $\exists\, tc_j \in T/\ CommitCond(a_i) = true \wedge \exists\, tc_k \in T/\ AbortCond(a_i) = true$.

5.3.2. Conditions-based criteria

These criteria are used to check the conditions that compose the logical expressions *BeginCond*, *CommitCond* and *AbortCond*:

- *Decision criterion (DC):* Every logical expression has taken true and false outcome at least once.
- *Decision/Condition criterion (DCC):* Every logical expression has taken true and false outcome and all conditions in each logical expression have taken true and false outcome at least once.
- *Modified condition/decision coverage (MCDC):* Every logical expression has taken true and false outcome at least once, all conditions in each logical expression have taken true and false outcome at least once, and each condition has been shown to independently affect the logical expression´s outcome (both true and false).

DCC subsumes *DC* and *MCDC* subsumes both *DC* and *DCC*. In the same way as *task-based* criteria, a deeper criterion requires a higher testing effort.

These criteria are formally defined as follow. Let define a transaction $wT = \{A, D\}$, a test suite $T = \{tc_1, \dots, tc_n\}$ and a logical expression $E \in \{BeginCond,\ CommitCond,\ AbortCond\}$.

T satisfies DC for wT if $\forall\, a_i \in A$, $\exists\, tc_j \in T/\ E(a_i) = true \wedge \exists\, tc_k \in T/\ E(a_i) = false$.

T satisfies the DCC for wT if $\forall\, a_i \in A$, $(\exists\, tc_j \in T/\ E(a_i) = true \wedge \exists\, tc_k \in T/\ E(a_i) = false) \wedge (\forall\, cond \in E(a_i), \exists\, tc_l \in T/\ cond = true \wedge \exists\, tc_l \in T/\ cond = false)$

T satisfies the MCDC for wT if $\forall\, a_i \in A$, $(\exists\, tc_j \in T/\ E(a_i) = true \wedge \exists\, tc_k \in T/\ E(a_i) = false) \wedge (\forall\, cond \in E(a_i), \exists\, tc_o \in T/\ E(a_i) = true$

$\Rightarrow (\neg cond \Rightarrow E(a_i) = false) \ \wedge \ \exists\, tc_p \in T/\ E(a_i) = false \Rightarrow (\neg cond \Rightarrow E(a_i) = true)$

5.3.3. Dependency-based criteria

Combining both primitive criteria, we define a family of criteria for testing dependencies in web services transactions. For each *task-based* criterion any *conditions-based* criteria can be applied. So we define nine criteria labelled as *T-C* where *T* is a *task-based* criterion and *C* is a *condition-based* criterion. *T* defines what primitive task will be exercised and, therefore, what logical expressions will be used. *C* defines what criterion will be used to exercise the conditions in such logical expressions. The proposed criteria are *ABC-DC*, *ABC-DCC*, *ABC-MCDC*, *ACC-DC*, *ACC-DCC*, *ACC-MCDC*, *ACAC-DC*, *ACAC-DCC*, *ACAC-MCDC*.

For example, in the *ACC-DCC* criterion, *ACC* requires all the activities to commit, so the logical expressions to be used are $CommitCond(s_i)$. *DCC* requires all the conditions in each logical expression to take true and false outcome at least once. So *ACC-DCC* criterion is defined as follow:

ACC-DCC: All the activities must commit at least in one test case, all activities must not commit at least in other another test case and all conditions in the committing logical expression have taken true and false outcome at least in one test case. Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A$, $(\exists\, tc_j \in T/\ CommitCond(a_i) = true \wedge \exists\, tc_k \in T/\ CommitCond(a_i) = false) \ \wedge \ (\forall\, cond \in CommitCond(a_i),\ \exists\, tc_l \in T/\ cond = true \wedge \exists\, tc_l \in T/\ cond = false)$

In the same way, the rest of criteria are defined as follow:

ABC-DC: All the activities must begin at least once and all activity must not begin at least once. Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A, \exists\, tc_j \in T/\ BeginCond(a_i) = true \wedge \exists\, tc_k \in T/\ BeginCond(a_i) = false$

ABC-DCC: All the activities must begin at least once, all activity must not begin at least once and all conditions in the beginning logical expression

have taken true and false outcome at least once. Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A$, $(\exists\, tc_j \in T/\ BeginCond(a_i) = true \wedge \exists\, tc_k \in T/\ BeginCond(a_i) = false) \wedge (\forall\, cond \in BeginCond(a_i), \exists\, tc_l \in T/\ cond = true \wedge \exists\, tc_l \in T/\ cond = false)$

ABC-MCDC: All the activities must begin at least once, all activity must not begin at least once, all conditions in the beginning logical expression have taken true and false outcome at least once, and each condition has been shown to independently affect the final outcome (both true and false). Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A$, $(\exists\, tc_j \in T/\ BeginCond(a_i) = true \wedge \exists\, tc_k \in T/\ BeginCond(a_i) = false) \wedge (\forall\, cond \in BeginCond(a_i), \exists\, tc_l \in T/\ BeginCond(a_i) = true \Rightarrow (\neg cond \Rightarrow BeginCond(a_i) = false) \wedge \exists\, tc_m \in T/\ BeginCond(a_i) = false \Rightarrow (\neg cond \Rightarrow BeginCond(a_i) = true)$

ACC-DC: All the activities must commit at least once and every activity must not commit at least once. Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A$, $\exists\, tc_j \in T/\ CommitCond(a_i) = true \wedge \exists\, tc_k \in T/\ CommitCond(a_i) = false$

ACC-MCDC: All the activities must commit at least once and all activity must not commit at least once, all conditions in the committing logical expression have taken true and false outcome at least once, and each condition has been shown to independently affect the final outcome (both true and false). Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A$, $(\exists\, tc_j \in T/\ CommitCond(a_i) = true \wedge \exists\, tc_k \in T/\ CommitCond(a_i) = false) \wedge (\forall\, cond \in CommitCond(a_i), \exists\, tc_l \in T/\ CommitCond(a_i) = true \Rightarrow (\neg cond \Rightarrow CommitCond(a_i) = false) \wedge \exists\, tc_m \in T/\ CommitCond(a_i) = false \Rightarrow (\neg cond \Rightarrow CommitCond(a_i) = true)$

ACAC-DC: All the activities must commit and abort at least once. Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A$, $\exists\, tc_j \in T/\ CommitCond(a_i) = true \wedge \exists\, tc_k \in T/\ CommitCond(a_i) = false \wedge \exists\, tc_l \in T/\ AbortCond(a_i) = true \wedge \exists\, tc_m \in T/\ AbortCond(a_i) = false$

ACAC-DCC: All the activities must commit at least once, all activities must not to commit at least once and all conditions in the committing logical expression have taken true and false outcome at least once. Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A$, $(\exists\, tc_j \in T/\ CommitCond(a_i) = true \wedge \exists\, tc_k \in T/\ CommitCond(a_i) = false) \wedge (\forall\, cond \in CommitCond(a_i), \exists\, tc_l \in T/\ cond = true \wedge \exists\, tc_l \in T/\ cond = false)$

ACAC-MCDC: All the activities must commit and abort at least once, all conditions in both committing and aborting logical expressions have taken true and false outcome at least once, and each condition has been shown to independently affect the final outcome (both true and false). Formally, let $wT = \{A, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall\, a_i \in A$, $(\exists\, tc_j \in T/\ CommitCond(a_i) = true \wedge \exists\, tc_k \in T/\ CommitCond(a_i) = false) \wedge (\exists\, tc_l \in T/\ AbortCond(a_i) = true \wedge \exists\, tc_m \in T/\ AbortCond(a_i) = false) \wedge (\forall\, cond \in CommitCond(a_i), \exists\, tc_o \in T/\ CommitCond(a_i) = true \Rightarrow (\neg cond \Rightarrow CommitCond(a_i) = false) \wedge \exists\, tc_p \in T/\ CommitCond(a_i) = false \Rightarrow (\neg cond \Rightarrow CommitCond(a_i) = true) \wedge (\forall\, cond \in AbortCond(a_i), \exists\, tc_p \in T/\ AbortCond(a_i) = true \Rightarrow (\neg cond \Rightarrow AbortCond(a_i) = false) \wedge \exists\, tc_q \in T/\ AbortCond(a_i) = false \Rightarrow (\neg cond \Rightarrow AbortCond(a_i) = true)$

5.4. Example of use

In this section we use the proposed method to test the dependencies in an example. In order to show the complimentary aspect of our approach with existing verification-based techniques, we will use the example presented in [111]. In that work, the authors presented a method to ensure the correctness of WS compositions. Here, we use the test criteria to check those identified requirements in the design phase regarding the implementation.

5.4.1. PC purchase

The example is an application dedicated to the online purchase of personal computer (OCP). This application is carried out by a composite service as illustrated in Figure 5.2. We assume the process design has been

correctly verified so our goal is to find faults in the implementation. Services involved in this application are: the Customer Requirements Specification (CRS) service used to receive the customer order and to review the customer requirements, the Order Items (OI) service used to order the computer components if the online store does not have all of it, the Payment by Credit Card (PCC) service used to guarantee the payment by credit card, the Computer Assembly (CA) service used to ensure the computer assembly once the payment is done and the required components are available, and the Deliver Computer (DC) service used to deliver the computer to the customer (provided either by Fedex (DF) or TNT (DT)).

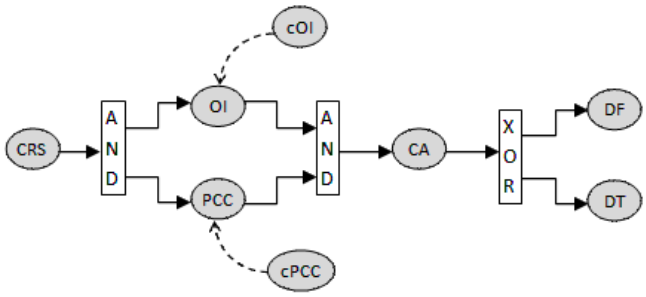


Figure 5.2. OCP application

The whole purchase process is identified as a WS transaction. As is identified in [111], several dependencies are necessary between the activities. Some dependencies are directly defined by the flow patterns (e.g. AND-split pattern). On the other hand, some dependencies are required due to the relationship between activities. If OI service does not complete its activity, the payment service PCC has to be compensated. In the same way, OI is compensated by cOI since if PCC fails, the order must be undone. Also there is a dependency between the delivery services since one and only one must commit. The WS transaction is modelled as is shown in Section 5.2. The logical expressions derived from the dependencies in the OCP example is shown in Table 5.7.

$wT_{OCP} = \{A_{OCP}, D_{OCP}\}$

$A_{OCP} = \{CRS, OI, cOI, PCC, cPCC, CA, DF, DT\}$

$D\_OCP = \{bc(CRS, OI), \quad bc(CRS, PCC), \quad bc(OI, CA),$

$bc(PCC, CA)$, $bc(CA, DF)$, $bc(CA, DT)$,

$fca(OI, cPCC)$, $bc(PCC, cPCC)$, $fca(PCC, cOI)$,

$bc(OI, cOI)$, $e(DF, DT)$, $fe(DF, DT)$ }

	$BeginCond(s_i)$	$CommitCond(s_i)$	$AbortCond(s_i)$
CRS	*	*	*
OI	$C(CRS)$	*	*
cOI	$A(PCC) \wedge C(OI)$	$A(PCC)$	*
PCC	$C(CRS)$	*	*
$cPCC$	$A(OI) \wedge C(PCC)$	$A(OI)$	*
CA	$C(OI) \wedge C(PCC)$	*	*
DF	$C(CA)$	*	$C(DT)$
DT	$C(CA)$	*	$C(DF)$

Table 5.7. Logical expressions in OCP example

5.4.2. Test case design

Since there are infinite possible test cases, it is necessary to define a subset of all possible tests. A test criterion will provide guidance for test cases generation. A test case is a specific way of executing the application in order to cover one or more requirements defined by the test criterion. To our field, such requirements are the value of the conditions that compose the logical expressions. So a test case describes which tasks have to be executed (and which not) in an execution of a web transaction.

Once the dependency-based criterion is chosen, the next step is to systematically apply it over the model. Let assume we want to apply *ABC-MCDC* for OCP application:

- The task-based (*ABC*) criterion specifies that all subtransactions have to begin at least in one test case and not to begin in at least another different test case, so the *BeginCond* expressions will be used.
- Since the condition-based criterion is *MCDC*, every condition of each *BeginCond* expression has to take a true outcome in at least one test

case and a false outcome in at least another different test case and, in both case, the value has been shown to affect the final expression´s outcome.

For example the *BeginCond* for CA activity is $BeginCond(CA) = C(OI) \wedge C(PCC)$, as is shown in Table 5.7. *MCDC* criterion applied over $BeginCond(CA)$ requires one test case where the expression takes the false outcome due to $C(PCC)$ is false. $C(PCC)$ may be false because it has not begun. In order to make true *C(OI)*, it requires CRS activity to commit. So the conditions are defined (*T*=true, *F*=false) as *B(CRS)=T, C(CRS)=T, B(OI)=T, C(OI)=T, B(PCC)=T*. It defines a situation where CRS receives and successfully reviews the customer requirements and then contacts with OI and PCC. While the OI service correctly achieves its goal (begin and commit the subtransaction), the PCC service does not execute its activity. In this way, according to the defined dependencies, CA service must not begin and thus, the rest of processes are not executed. The rest of test cases according to the criteria can be defined in the same way. As example, we present in Appendix A the algorithm to apply the ABC-DC criterion and to automatically obtain the test conditions according to such criterion.

The application of the proposed test criteria allows deriving *positive* and *negative* test cases. A *positive test case* exercises the application in a right way, in other words, according to the specification. For example the test scenario TC1 identified in Figure 5.3 achieved using *ABC-DC* criterion. Dash means that it does not matter what is the value. The test scenario defines the following execution: The Customer Requirements Service (CRS) successfully receives and reviews the customer order. The Order Items service (OI) has successfully ordered the required items and the payment has been successfully done using the Payment service (PCC). These two actions have begun in parallel. Later, the computer is successfully assembled. Finally the two delivery services are notified to check their availability. This test case could detect failures of extra dependency implementation; for example, if OI waits to order the items until PCC has charged the payment, the whole process will take longer time keeping the resources busy and maybe rejecting new orders where they are actually available.

A *negative test case* exercises the application in a wrong way. It means that the execution tries to break the specification. This kind of test case can detect fault of dependencies implementation omission. For example the test scenario TC2 identified in Figure 5.3, achieved using the *ABC-DC* criterion too. This test case tries to order and to charge without reviewing the customer requirements. If the scenario can be executed, a failure will be detected: the constraints of successfully committing of CRS before OI and PCC can begin are not implemented. So a purchase of incompatible items for a personal computer can be allowed.

TC1	Begin	Commit	Abort
CRS	T	T	F
OI	T	T	F
cOI	F	F	F
PCC	T	T	F
cPCC	F	F	F
CA	T	T	F
DF	T	–	–
DT	T	–	–

TC2	Begin	Commit	Abort
CRS	T	F	F
OI	T	T	F
cOI	F	F	F
PCC	T	T	F
cPCC	–	–	–
CA	–	–	–
DF	–	–	–
DT	–	–	–

Figure 5.3. Test case design examples

The Appendix B shows the test cases generated for the OPC example following the *ABC-DC* (6 test cases), *ACAC-DC* (9 test cases) and *ACC-MCDC* (9 test cases).

5.4.3. Evaluation

In order to evaluate the test cases generated guided by our test technique, we follow the method proposed in [122]. The method, based on specification-based mutation, allows measuring completeness, adequacy and coverage of test sets.

Mutation analysis is a fault-based testing technique that uses mutation operators to introduce small changes into a specification, producing faulty versions called mutants. For instance, an insertion mutation operator can replace a boolean condition with a disjunction of the condition and another boolean condition. By systematically applying the set of operators we obtain a set of mutated specifications. If a test set can distinguish a specification

from each slight variation, the test set is exercising the specification adequately. When a test set identifies a mutant, it is said that the mutant was killed. Better test sets are those which kill more mutants. Here we apply mutation operator over the logical expressions defined by the dependencies. We generate first order mutants of the specification, in others words, only one fault is injected in each mutant. We use a subset of the mutation operations proposed in [123]:

Mutation of actions

Action Replacement Operator (ARO): It replaces a subtransaction action by another. For example, it replaces $BeginCond(a_i) = C(a_j) \wedge B(a_k)$ with $BeginCond(a_i) = A(a_j) \wedge B(a_k)$

Missing Action Operator (MAO): It omits an action. For instance, it replaces $BeginCond(as_i) = C(a_j) \wedge B(a_k)$ with $BeginCond(a_i) = C(a_j)$

Action Insertion Operator (AIO): It inserts an action, that is, it replaces a condition c with $c * d$ where d is another action of any subtransaction involved in the expression, $*$ is either conjunction o disjunction. For example, it replaces $BeginCond(a_i) = C(a_j) \wedge B(a_k)$ with $BeginCond(a_i) = C(a_j) \wedge B(a_k) \wedge C(a_l)$

Mutation of logical operators

Logical Operator Replacement (LOR): It replaces a logical operator ($\wedge$, $\vee$) by another logical operator. For example, it replaces $BeginCond(a_i) = C(a_j) \wedge B(a_k)$ with $BeginCond(a_i) = C(a_j) \vee B(a_k)$

Mutation of subtransactions

Subtransaction Replacement Operator (SRO): It replaces a activity involved in an action by another. For example, replace $BeginCond(a_i) = C(a_j) \wedge B(a_k)$ with $BeginCond(a_i) = C(a_l) \wedge B(a_k)$

5.4.4. Results

Our method allows automatically deriving test conditions for validating the dependencies implementation. As a first approach, the test sets for OPC application are defined using ABC-DC, ACAC-DC and ACC-MCDC criteria. They are shown in Appendix B.

As we explained above, the test conditions define two kinds of test scenarios. *Positive test scenarios* exercise the application in a right way, in other words, according to the specification (e.g TC1.2). *Negative test scenarios* exercise the application in a wrong way. That is mean that the execution try to break the specification (e.g. TC1.6).

The evaluation carried out shows that all mutated specifications were killed by the test cases generated using our approach. Some faulty specifications, achieved using the mutation operators, are shown in Appendix C. For example MUT1 introduces a relaxation in cPCC begin conditions due to the original specification requires OI to be aborted while MUT1 only requires OI to be begin. This mutation is killed with the test scenario defined in TC3.2. In that case, the expected result is that cPCC does not begin since OI begins and commit but not aborts, but according to MUT1 cPCC would begin. In a similar way MUT2 and MUT3 can be killed by different test scenarios.

5.5. Industrial case study

The method proposed in this chapter has been used in a real industrial case study. This section describes the case study, the application of the method and the obtained feedback.

5.5.1. Cajastur insurances application

Cajastur is a financial institution from Asturias (Spain). For more than 130 years, it has been one of the pillars of the Asturian economy. Today it conducts its banking business through Liberbank, entity which has 66 percent of the capital.

Cajastur Insurances Application (CIA) is a software application of the Cajastur´s systems used to contract personal and car insurances. CIA contacts with different insurances providers and presents the customers with different alternatives. Currently, Cajastur collaborates with three car insurance providers and one life insurance provider.

CIA contacts with the insurance providers by using their web services. Also CIA use the private Cajastur Customer Service (CCS), that is allocated to the host that allows querying the personal data of the Cajastur´s customers. The architecture of the CIA is shown in Figure 5.4.

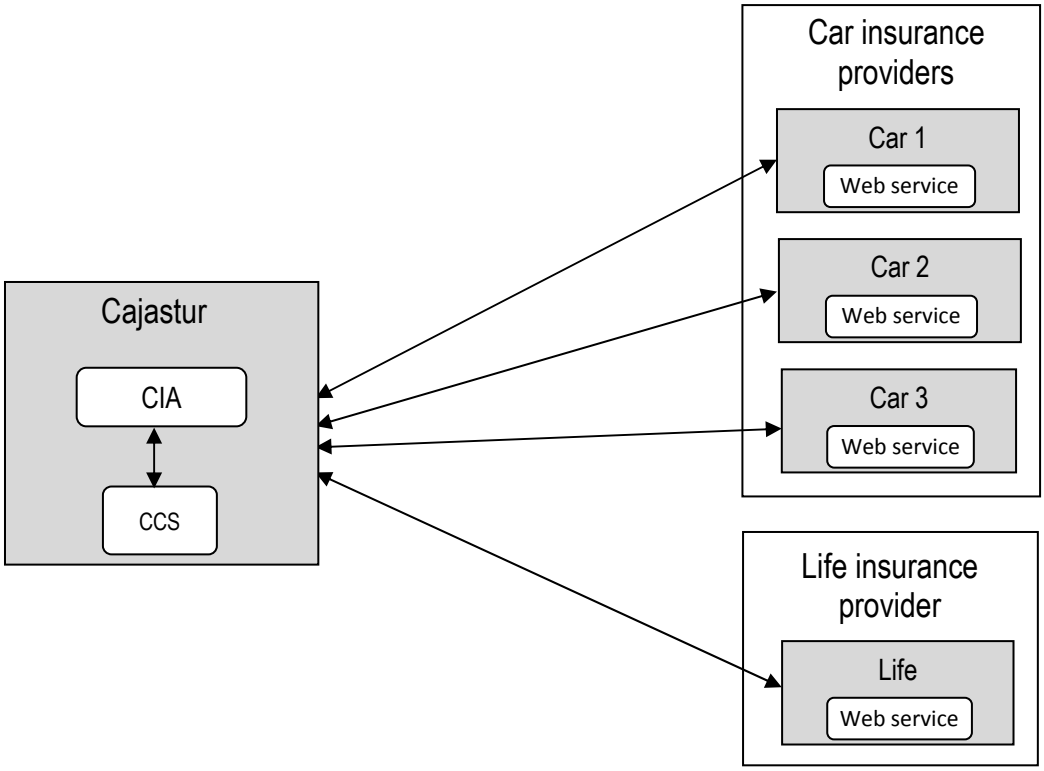


Figure 5.4. Cajastur Insurance Application (CIA)

The process of contracting insurance is described as follow. Firstly the CIA gets the customer´s personal data from the CCS. The additional required information is manually introduced in the application such as type of insurance (car or life insurance), car registration number or healthy problems. Secondly, CIA contacts with the suitable service(s) in order to get insurance proposals. The service(s) receive the information, generate a

insurance proposal and send such proposal back to the CIA. There are two type of insurance proposal. A *concrete* proposal means that all data are provided (price, insurance policy, etc) and the customer can accept it. The other type of proposal is an *offer* proposal. It means that the insurance provider have to contact the customer (by phone) in order to negotiate a concrete proposal. After the negotiation, the offer proposal becomes a concrete proposal. The customer can accept any of the concrete proposals or reject all of them. If no concrete proposal is accepted, the process is finished. In the other case, the customer accepts one concrete proposal and, automatically, the rest of proposals are rejected. Note that the process is a long-lived transaction as a customer can stop the process of contracting an insurance once he/she has some concrete proposal and can continue it later (e.g. the next day he come back to the Cajastur branch). Once a concrete proposal is accepted, the payment of such insurance is charged to the customer´s bank account. The payment information is sent by the CIA to the insurance provider using its web service. Finally, if there are no problems, the insurance provider confirms the contracting of the insurance and sends back to the CIA the customer documentation.

5.5.2. Transaction modelling

The transactional process carried out by the CIA, is modelled according to the method explained in 5.2. The model is composed of a set of activities and a set of dependencies.

In order to present the approach to the industrial partner, we decided to use a graphic notation of the model. Figure 5.5 depicts such model. Each activity is shown as a rectangle. The flow after an activity is completed is shown as a bottom arrow. To show the possibility of abort we use a t-shaped line in the right side of an activity. The AND-join, AND-split, OR-join, OR-split, XOR-join and XOR-split patterns (relationships) have been illustrated using the BPMN notation. Multiple instances of a same process (the 3 car insurance providers in this case) are also depicted using the BPMN notation. The transitions in the model have been enumerated to make the traceability easier between the graphic model and the test cases generated.

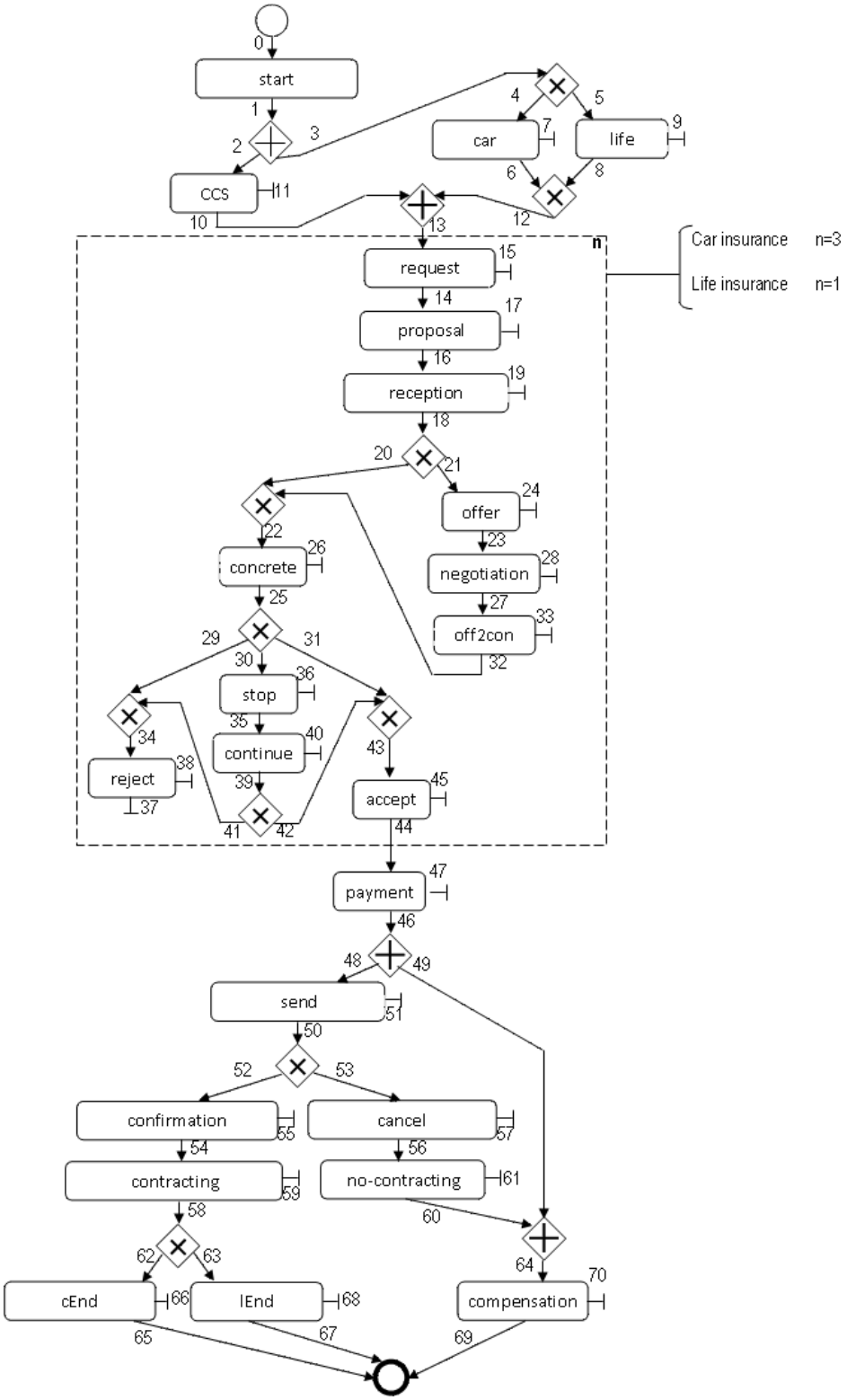


Figure 5.5. CIA model

Activities

- *Start*: The insurance contracting starts by selecting the type of insurance to contract.
- *Customer information (CCS):* Consult the CCS in order to get the customer personal data saved in the Cajastur´s sytems.
- *Life information (life)*: Introduce in the application the required information to obtain a life insurance proposal.
- *Car information (car)*: Introduce in the application the required information to obtain a car insurance proposal.
- *Proposal request (request)*: Send from CIA to the suitable web services the customer information in order to get the insurance proposals.
- *Proposal*: The service(s) get the information from CIA, prepare an insurance proposal and send it back to the CIA.
- *Reception:* CIA receives an insurance proposal.
- *Concrete*: CIA has received a concrete insurance proposal.
- *Offer*: CIA has received an offer insurance proposal. The insurance provider contacts to the customer by phone.
- *Negotiation*: The insurance provider and the customer negociate the offer proposal.
- *Offer to concrete (off2con)*: The customer and the insurance provider have agreed to a concrete insurance proposal. The insurance provider sends the concrete proposal to the CIA.
- *Accept*: The customer accepts a concrete proposal.
- *Reject:* The customer rejects an insurance proposal.
- *Stop:* The customer stops the process without accepting or rejecting the insurance proposals.

- *Continue:* The customer continues the process of contracting an insurance policy. The customer already has concrete proposal to accept or reject.

Each service of an insurance provider executes the following activities: *request*, *proposal*, *reception*, *concrete*, *offer*, *negotiation*, *offer2concrete*, *accept*, *reject, stop* and *continue*.

- *Payment*: Execute the internal transaction to transfer the money from the customer´s bank account to the insurance provider´s bank account. Also Cajastur charges a tax as part of intermediary (agent)..
- *Send*: CIA send the confirmation of the payment to the insurance provider.
- *Confirmation*: The insurance provider responds with the confirmation of the insurance contracting.
- *Cancel*: The insurance provider responds rejecting the contracting.
- *Contracting confirmation (contracting)*: CIA receives the confirmation of the insurance contracting.
- *Contracting rejecting*: *(no-contracting)*: CIA receives the cancellation of the insurance contracting.
- *Car end (cEnd):* CIA prints the documentation related to the car insurance contracting.
- *Life end (lEnd)*: CIA prints the documentation related to the life insurance contracting.
- *Compensation*: Undo the payment activity.

Dependencies

This section defines the dependencies exist in the CIA applications. Here we use the notation presented in Section 5.2.

bc(CCS, start) bc(car, start) bc(life, start)

bc(request, CCS) e(car, life) bc(request, fe(car, life))

bc(proposal, request) bc(reception, proposal)

e(concrete, offer) fe(concrete, offer) bc(concrete, reception)

bc(offer, reception) bc(accept, continue)

bc (negotiation, offer) bc(off2con, negotiation)

bc(reject, concrete) bc(stop, concrete) bc(continue, stop)

e(accept, reject) e(accept, stop) e(reject, stop)

fbc(accept, continue) fbc(accept, stop) fbc(reject, continue)

fbc(accept, reject) fbc (reject, stop) bc(payment, accept)

bc(send, payment) bc(confirmation, send)

bc(cancel, send) e(confirmation, cancel)

bc (contracting, confirmation) bc(cEnd, contracting)

bc(lEnd, contracting) bc(no-contracting, cancel)

bc(cEnd, car) bc(lEnd, life) e(cEnd, lEnd)

c(compensation, no-contracting) fcc (no-contracting, compensation)

fcc(accept*X*, reject*Y*) `X €{1,2,3}, Y € {1,2,3}-X`

5.5.3. Logical expressions

The logical expressions that define the task relationships derived from the dependencies (see Section 5.2.3) are shown in Table 5.8.

Activity	**BeginCond**	**CommitCond**	**AbortCond**
start			
CCS	C(start)		
car			
life			
request	C(car) && [C(car) || C(life)]		
proposal	C(request)		
reception	C(request)		
concrete	C(reception)		C(offer)
offer	C(reception)		C(concrete)
negotiation	C(offer)		
off2con	C(negotiation)		
accept	C(concrete) || C(continue)	!C(reject) && !C(stop)	C(reject) || C(stop) || C(acceptY)
reject	C(concrete)|| C(continue)	!C(accept) && !C(esp)	C(accept) || C(esp)
stop	C(continue)	!C(accept) && !C(reject)	C(accept) || C(reject)
continue	C(stop)		
payment	C(accept)		
send	C(payment)		
confirmation	C(send)	!C(cancel)	C(cancel)
cancel	C(send)	!C(confirmation)	C(confirmation)
contracting	C(confirmation)		
no-contracting	C(cancel)		
cEnd	C(contracting) && C(car)	!C(life)	C(life)
lEnd	C(contracting) && C(life)	!C(car)	C(car)
compensation	C(payment) && C(cancel)		

Table 5.8. Logical expressions in CIA

5.5.4. Test case generation

To generate the test cases we selected the **A**ll-**C**ommit-**A**bort **M**odified **C**ondition **D**ecision **C**overage (ACA-MCDC) criterion.

Thirty-two (32) test cases were generated according to the test criterion. Eight of them are classified as *positive* test cases while twenty-four were *negative*. Positive test cases check whether the system do what it is supposed to do. Negative test cases are designed to test the system in ways it was not intended to be used.

5.5.5. Results

When we applied our test method to the CIA, it was already in the production environment. In order to evaluate the quality of our test case we obtained the feedback from Cajastur regarding the failures detected in the CIA during its testing phase and other potential failures. The actual failures had been logged by the CIA project manager. The testing phase that Cajastur had done was composed of two phases:

1) The development team tested the software system themselves after the coding phase using a black box approach.

2) Once the development team released a beta version, it was tested by non-technical people who were experts on the business logic. In the case of the CIA, it was tested by the insurance brokers. They based the tests on their own knowledge and experience about the field.

The actual failures detected during the above testing phases and other potential failures identified by the Cajastur team were the following:

i. CIA allowed starting the contracting process to a person whose personal data was not stored in the CSS.

ii. CIA allowed to request a car insurance proposal without adding the car registration number

iii. One of the life insurance service crashed when an invalid XML message was sent as a request message

iv. CIA was blocked if any of the services did not response

v. CIA crashed if an invalid XML message was received as a response of any service

vi. CIA did not take into account that the insurance provider can reject a paid proposal because the characteristics had changed.

vii. CIA was blocked if unexpected data were sent in the contracting confirmation

viii. CIA crashed if the CSS was unavailable

ix. CIA did not allow re-printing the information in case of problem with the printer.

x. CIA did not check the customer´s bank account balance before executing the transaction payment.

xi. CIA did not check that the customer personal data sent to the service was similar to the data received in the confirmation.

xii. CIA did not check that the payment was correctly received by the service. In case of the network failure, CIA did not undo the payment.

xiii. CIA did not take into account the possible network message during the confirmation process. If CIA did not receive the confirmation/cancel message, it undid the payment even if the service had confirmed the contracting and the message was lost. CIA should contact again to the service in order to have the real decision.

xiv. If CIA received an invalid XML message as confirmation, it undid the payment but the insurance was actually contracted.

xv. If CIA had to execute the compensation (i.e. to undo the payment) and the CSS was unavailable, CIA marked the process as compensated but the money was not refunded to the customer.

We delivered the 32 test cases generated by our method to the CIA project manager. He and his test team analyzed the proposed test cases. The result of such analysis is summarized as follows:

- All failures would been detected by the proposed test cases
- Such systematization of the test case design would have saved effort and time to the project
- Some of the proposed test case were identified as very interesting and had not been achieved by the Cajastur testing team
- It would have been useful to have a tool to help in the modelling phase and to automate the test case design process

5.6. Summary

This chapter has presented a condition-based approach to address the *flow feature* of the Dependency property according to the Framework for Testing Transactions (F2T) defined in Chapter 3.

The flow execution of a WS transaction is defined in terms of relations between its activities´ tasks (begin, commit and abort). The logical conditions specified by tailoring those relations to define a logical expression that fire a task once is evaluated as true. In other words, they specify a precondition to be enforced before the activities can execute the task. Taking into account the different tasks and the existing condition-based testing criteria, this chapter have presented a new family of test criteria for testing the dependencies in WS transactions.

The explanation of the proposed approach has been shown through the *PC purchase* application, an example previously used in different application areas. In order to evaluate quality of the test case and the practical use of

the proposed method, this chapter presented the application of the approach in an industrial case study: *Cajastur Insurances Application (CIA).* The results have shown the viability of our method.

However, we have identified some open issues in this method. Firstly it only addresses the *flow* feature, so the *control* and *data* features are out of its scope. Also the existence of alternative activities increases several times over the task relationships. It makes the flow definition difficult and, sometime prone to make mistakes.

In order to address the open issues commented above, Chapter 6 presents a different approach. That approach uses the Classification-Tree [118] method to deal with *flow* but also *control* and *data* feature, as well as the existing of alternative activities.

Chapter 6

Testing at transaction level: Classification-Tree based approach

The experimenter who does not know what he is looking for will not understand what he finds

Claude Bernard

This chapter presents a different testing approach to address the hazards indentified in the Dependency property. The method proposed in this chapter identifies, analyzes and classifies the possible relationships between services at transaction level. The Classification-Tree technique is used to derive the test conditions and test coverage items for each kind of dependency (relationship). A family of test criteria are proposed to generate the test cases based on the generated tree. Those criteria have been designed, implemented and evaluated through a case study and a number of experiments have been performed.

6.1. Introduction

The management of transactional activities complicates the business logic of web services since their execution requires careful coordination, accounting for fault-tolerance, correct process termination and cancelation, without undesirable consequences at any stage of the execution. In a WS transaction some information about the internal behaviour of a particular service is disregarded since they follow a general pattern during execution [88]. The pattern is defined in terms of the individual behaviours (i.e. the possible states and transitions of each participant) and also in terms of relationships between the activities (e.g. execution flows, nested subtransactions, etc). In summary, testing of WS transactions involves different challenges related to the relationships between services and the shared data and consistency issues.

Testing at the transaction level involves many factors and dependencies such as type of relationship (e.g. merge, union, etc), external conditions derived from the business logic or cardinality of the union. Due to these factors, there is a large set of possible situations to analyse for each dependency. In Chapter 5 we described the dependencies in terms of logical conditions between the activities´ tasks. A test method was proposed and evaluated to address some of the hazards that threaten the Dependency property. Despite the viability of such proposed method, some related issues still require attention such as the alternative activities to perform the same work or the use of shared data during the execution. This chapter proposes a different approach to include all the identified Dependency hazards according to the Framework for Testing Transactions (F2T, Section 3.2).

To deal with all the factors involved in the dependencies, this chapter proposes to identify, organize and classify those situations. A well-known method to classify situations for testing purposes is the Classification-Tree (CT) approach [118]. CT has been successfully used in both academic and industrial sectors [127, 128]. In this approach, the test basis is the whole transaction while the test items are the dependencies. For each dependency

we identify the test conditions and test coverage items relevant for such relationship by elaborating a CT.

Table 6.1 shows how the goals of this chapter fit into the F2T. Similar to Chapter 5, this chapter focus on the *transaction level* according to the different levels identified in F2T. The *flow*, *data* and *control* (*feature dimension*) are all taken into account during the tree generation. This approach, therefore, addresses the hazards DEP1, DEP2 and DEP3. The test effort (*depth* dimension) is managed by the proposed new family of test criteria based on different combination of test coverage items identified in the tree analysis.

System property and hazards	*Dependency*	DEP1 Order DEP2 Relationship DEP3 Data
Testing Dimensions	*Level*	Transaction
	Feature	Flow, Data, Control
	Depth	Combination criteria
Test case generation		Classification-Tree

Table 6.1. Relationship of chapter 6 with F2T

6.2. 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 activities' behaviours. 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 method proposed in this chapter, 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 identified classes: orthogonal and exclusive.

   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:

   - *Activities*: refers to the aspects related to the activities involved in the dependency such as its behaviour or the cardinality.

   - *Situations*: defines if the situation is possible or impossible. A situation is possible if it meets the expected behaviour. For example, in a Sequence dependency, if the former activity does not achieve the complete state, the later activity does not begin. In

other case, the situation is defined as impossible (e.g. the later activity begins when the former activity had been aborted).

- *Finished*: refers to set of activities that are in the completed state.
- *Not finished*: refers to the set of activities that are not in the completed state.
- *Cardinality*: number of activities involved in the situation defined in the parent node.
- *Behaviour*: current state of the involved activities.
- *Selected*: specific activity involved in the situation
- *Definer*: the set of activities that act as input in the dependency
- *User*: the 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:

- *Continue:* the requirements of the dependency are fulfilled so the flow execution continues. For example in a Merge dependency, at least one of the input activities is in the completed state.
- *Stop:* the requirements of the dependency have not been fulfilled so the flow execution stops. For example, in Join dependency, at least one the input activities has not achieved the completed state.
- *Finished Behaviour:* the activity either has completed or it has been compensated once the flow of execution had passed the dependency. In other words, the activity was completed when the flow achieved the dependency but it was later compensated.
- *Not Finished Behaviour:* the activity is not in the completed state. It can be still running, aborted or could been compensated before the flow of execution achieved the dependency.

- *Aborted:* the activity has achieved the aborted state so it was either withdrawn, cancelled or it failed.

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 classified into classes and/or values. Each time a class is classified in further classes or values, a new deep level can be defined. These deep levels allow defining CTs which require different degree of test effort as discussed in Section 6.4.

4) A set of constraints between the values/classes is specified. Those constraints avoid the generation of impossible scenarios.

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 6.4.

Figure 6.1 shows the generic structure of a dependency classification-tree according to the concepts presented above. The graphic notation used is the following: orthogonal classes are shown within a rectangle and exclusive classes are shown without rectangle. The concrete values are the leaf nodes in the tree.

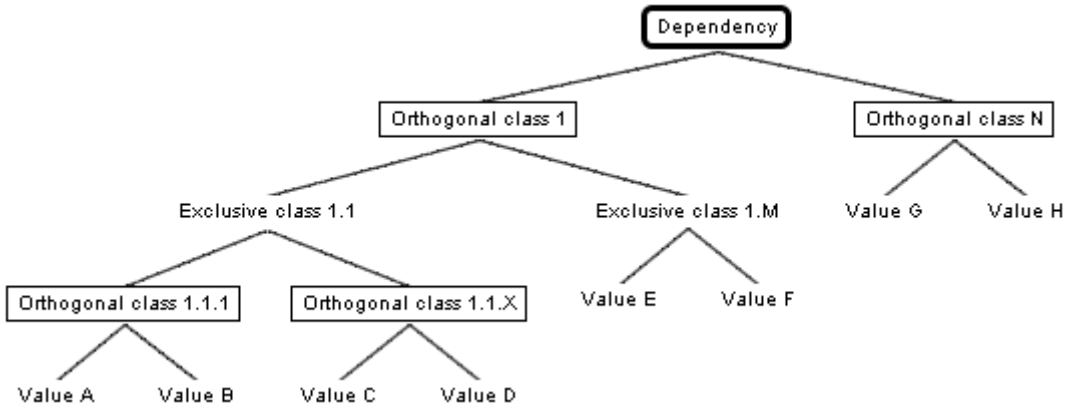


Figure 6.1. Concepts in a dependency classification-tree

6.3. Dependencies classification trees

This section presents the classification tree generated for each dependency. The trees have been elaborated according to the steps explained in the previous Section 6.2.

The classification trees are organized into three families according to its structure: *input*, *output* and *data*. The input family includes Merge, Join and Exclusion dependencies. This family focuses on the cardinality and behaviour of the inputs activities. The output family includes the Sequence, Fork and Alternative dependencies. This family mainly focuses on the outputs of activities. Finally, the Data family includes the Write dependency since its classification tree indentifies peculiar situations not included in the previous families.

6.3.1. Input dependencies: Merge, Join, Exclusion

This section explains the process of designing classification tree for the input family. Since the structure of all their trees (Merge, Join, Exclusion), are similar, we detail the generation with the Merge dependency and then, the specific characteristics of the Join and Exclusion classification trees are defined.

Merge classification tree

Figure 6.2 depicts the classification tree for the Merge dependencies. The dotted lines represent the separation between the deepest level in the tree and level before. The criteria for defining the test coverage items depend on this division as is shown in the Section 6.4.

The Merge dependency relates a set of more than one activity with another one. 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 the behaviour of the whole dependency to see whether it is fulfilled or not. In other words, whether the execution flow (*Flow*) can continue or not. So at the top level we identified two exclusive classes. *Continue* class classifies the situations where at least one activity has completed, whereas *Stop* class classifies the situations where no single activity has completed successfully.

In the case of *Continue* class, we identify two orthogonal classes to represent the activities that have completed (*Finished*) and the ones that do not (*Not finished*). In the case of the *Finished* activities, we define two orthogonal classes in order to represent the *Cardinality* of the completed activities (*One* and *More than one*) and the way they finished (*Completed* or *Later compensated*). We define such cardinality in order to specify the condition that at least one activity must be completed. Also we identified those behaviours to specify the two possible scenarios for an activity that is in the completed state when flow achieves the Merge dependency.

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 *Not Finished* class. This class shows whether activities are still running (*Running*), have been compensated before the flow achieved the dependency (*Previously Compensated*) or are aborted (*Aborted*). If they are still running it means all of them are in the Active state. We identify three types of the

Aborted class: some have been withdrawn, some have been cancelled or some have failed.

The other branch of the tree (*Stop* class) means that the dependency was not fulfilled. In other words, no activity has completed. So the relevant feature to test is, again, the way in the activities have finished. We use again the orthogonal classes *Selected* and *Not finished Behaviour* to represent the behaviour of the activities. In the same way, we use the *Abort* class to specify the different ways an activity can abort.

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 not extra constraints about combining leaf nodes apart from the specification by the orthogonality of the some classes as was described above. The way the leaf nodes are used to define the test coverage items are explained in the Section 6.4.

Join classification tree

The Join dependency requires all input activities to complete. The difference between Join and Merge is that while Join requires all the activities completed, to the Merge dependency is necessary only one. Therefore, the Join classification tree (Figure 6.3) has a similar structure to the Merge classification tree but CT focuses on the cardinality (*Cardinality*) of the input activities, especially when the dependency requirements are not met (*Stop*).

Exclusive classification tree

The Exclusive dependency requires exactly only one input activity to complete. The difference between Merge CT and Exclusive CT (Figure 6.4) lies in that the later focus on both behaviour (*Behaviour*) and cardinality (*Cardinality*) aspects of the activities in the scenarios where the execution flow does not continue (*Stop*).

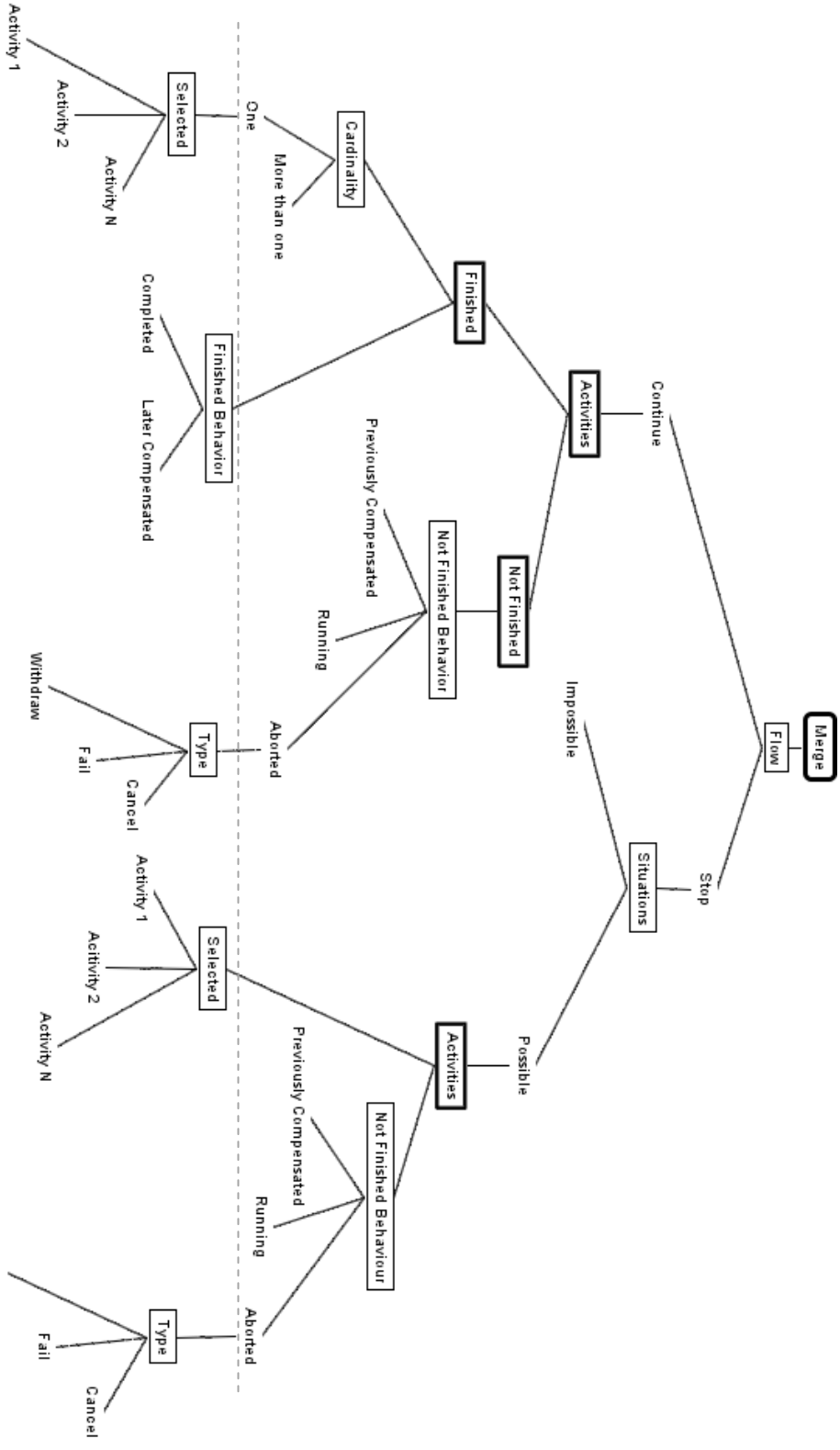


Figure 6.2. Merge classification tree

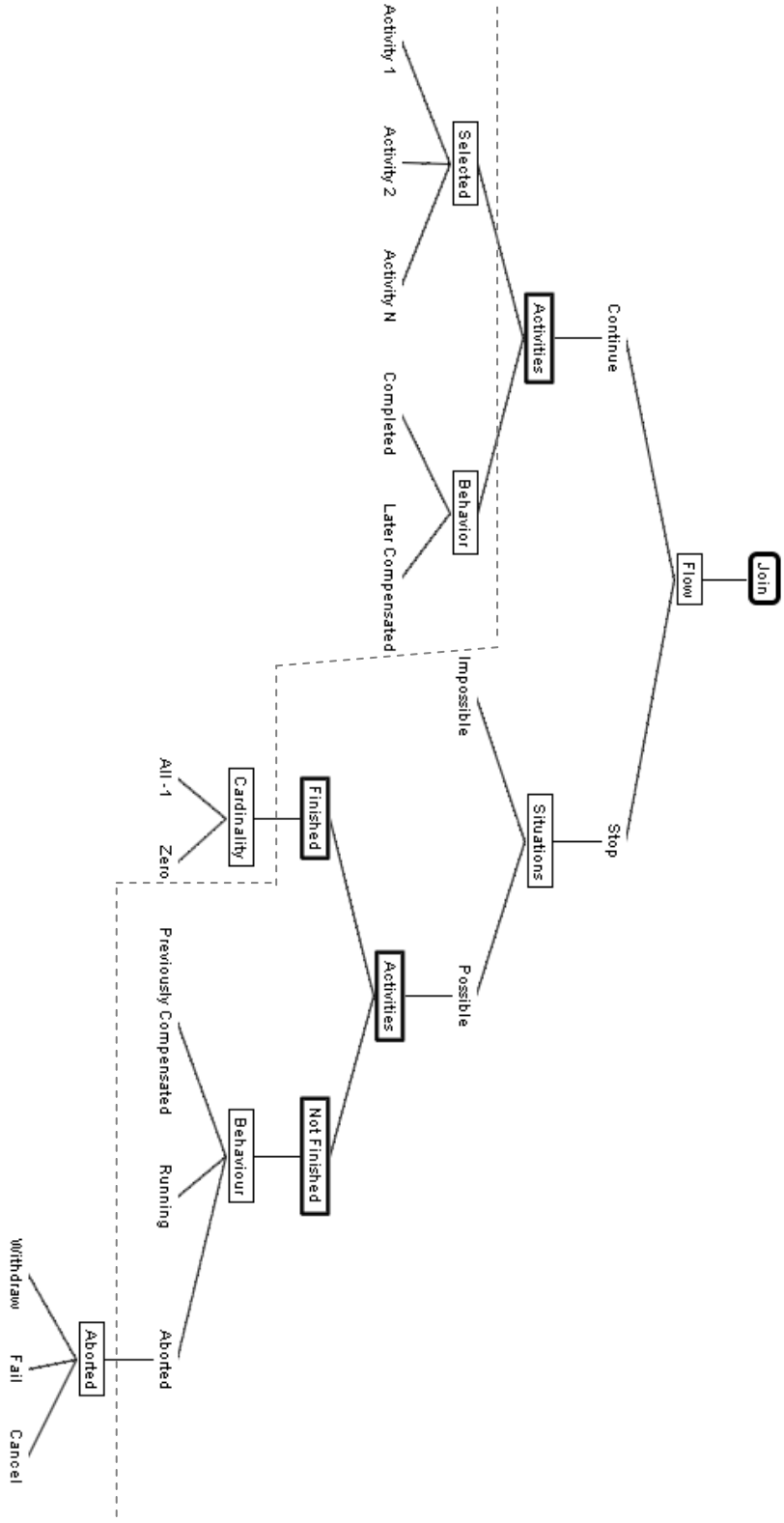


Figure 6.3. Join classification tree

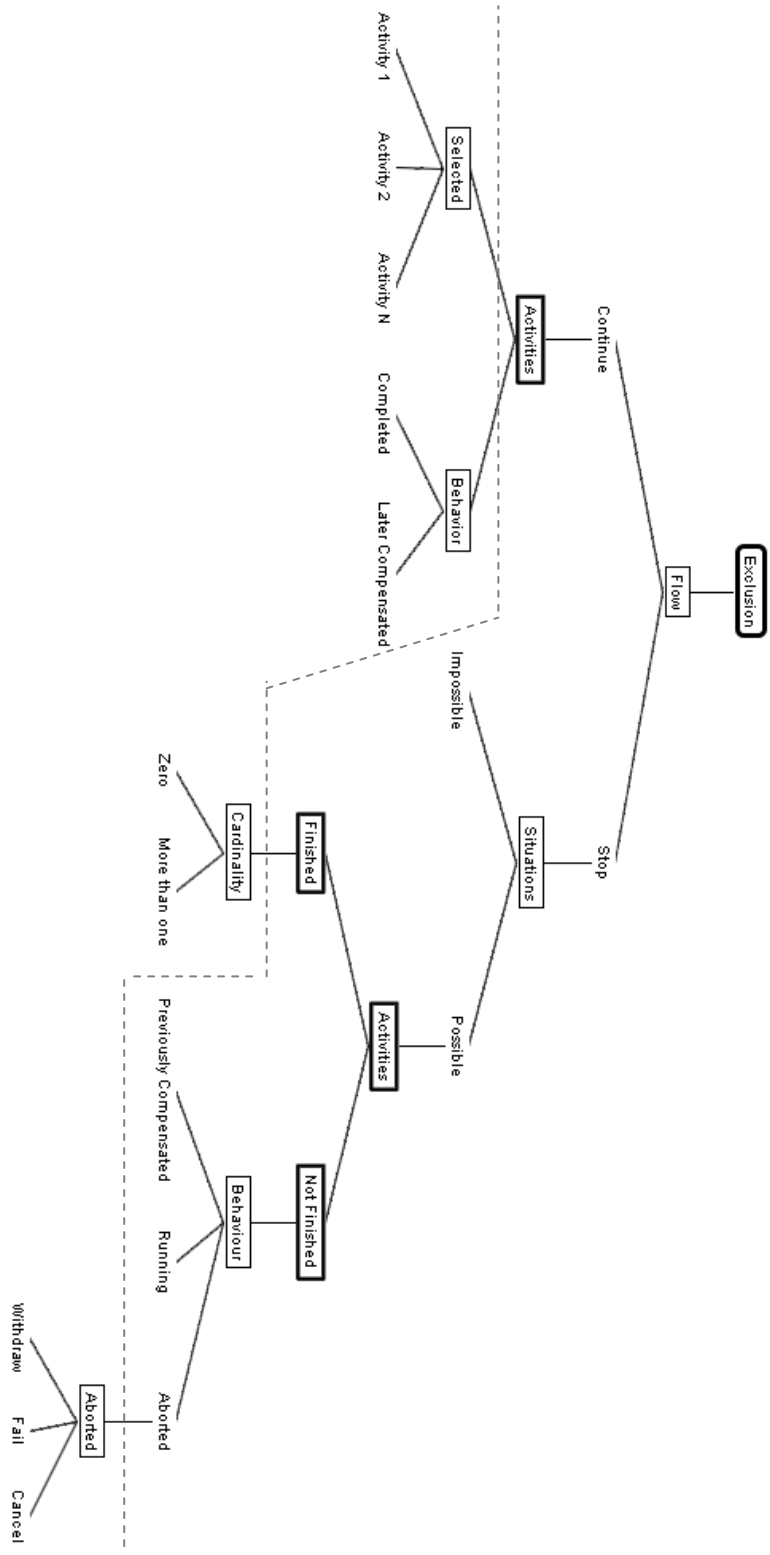


Figure 6.4. Exclusive classification tree

6.3.2. Output dependencies: Alternative, Fork, Sequence

This section explains the process of designing classification tree for the output family. In the same way that the input family, we illustrate the generation with one dependency, Alternative. The specific characteristics of the Fork and Sequence classification trees are later commented.

Alternative classification tree

The Alternative dependency relates an input activity with a set of output activities. It defines two requirements: (i) only one of the output activities can begin, and (ii) the input activity must complete before the output activity can begin. The classification tree for the alternative dependency is shown in Figure 6.5. The main feature we address in the tree is if the flow (*Flow*) can continue (*Continue*) or not (*Stop*).

In the case of *Continue* class, we identify two orthogonal classes to represent how the input activity has completed (*Input*) and the information related to the output activity (*Output*). For the *Input* class we identify two possible interesting values from a testing point of view. *Completed* refers that the activity was successfully completed while *Later Compensate* means that it was in the completed state when the flow achieved the dependency but later the activity was compensated. To represent the information related to the output activity, we define two orthogonal classes. *Selected* refers to the activity selected to begin and *Behaviour* represents the behaviour of such activity. *Behaviour* class is decomposed in the possible values: *completed*, *compensated*, *running*, and *abort.* Since there are three different ways of abort, the *Abort* class is decomposed in such values: *withdraw*, *fail* and *cancel*.

In the case of *Stop* class, the tree focuses on the possible situations (*Situations*) that not allow the flow to continue. If the flow cannot continue it is due to the input activity have not completed (*Possible*). So we focus on the different non-completed behaviour (*Not Finished Behaviour*) of that activity. Note that we define a value *Impossible* to define unexpected

behaviour of that dependency. In other words, scenarios where an output activity begins but the input activity have not completed. In the *Not Finished Behaviour* class, we identify the possible values: *Previously Completed*, *Running*, or *Aborted*. As in other cases, the class *Aborted* is decomposed in the different ways to abort.

Fork classification tree

The Fork dependency relates an input activity with a set of output activities. Fork defines that once the input activity has completed all the output activities can begin. So the difference between Fork and Alternative is that the latter defines only one output activity to begin while Fork defines all of them to begin. Therefore, Fork relaxes the Alternative requirements.

The structure of both trees (Figure 6.5) is the same because, from a testing point of view, we are still interested in the possible scenarios of the input and output activities. Note that Fork dependency is more important to be tested when it is the last dependency. It means that once the output activities finish, the flow of execution of the whole transaction finishes too. In this case, those output activities are not input activities of others dependencies so they have to be taken into account in the Fork dependency. In the case that Fork is not the last dependency, its output activities will be the input activities of others dependency and will be further taken into account on those dependencies.

Sequence classification Tree

Sequence dependency relates one input activity with one output activity. It defines that the input activity must complete before the output activity can begin. Sequence is, therefore, a simplification of the Alternative. While in Alternative there are a set of candidate output activities, in Sequence there is only one. The classification tree (Figure 6.6) still focus on the same concepts (behaviour of the input and output activities) but it only has to take in account one input dependency.

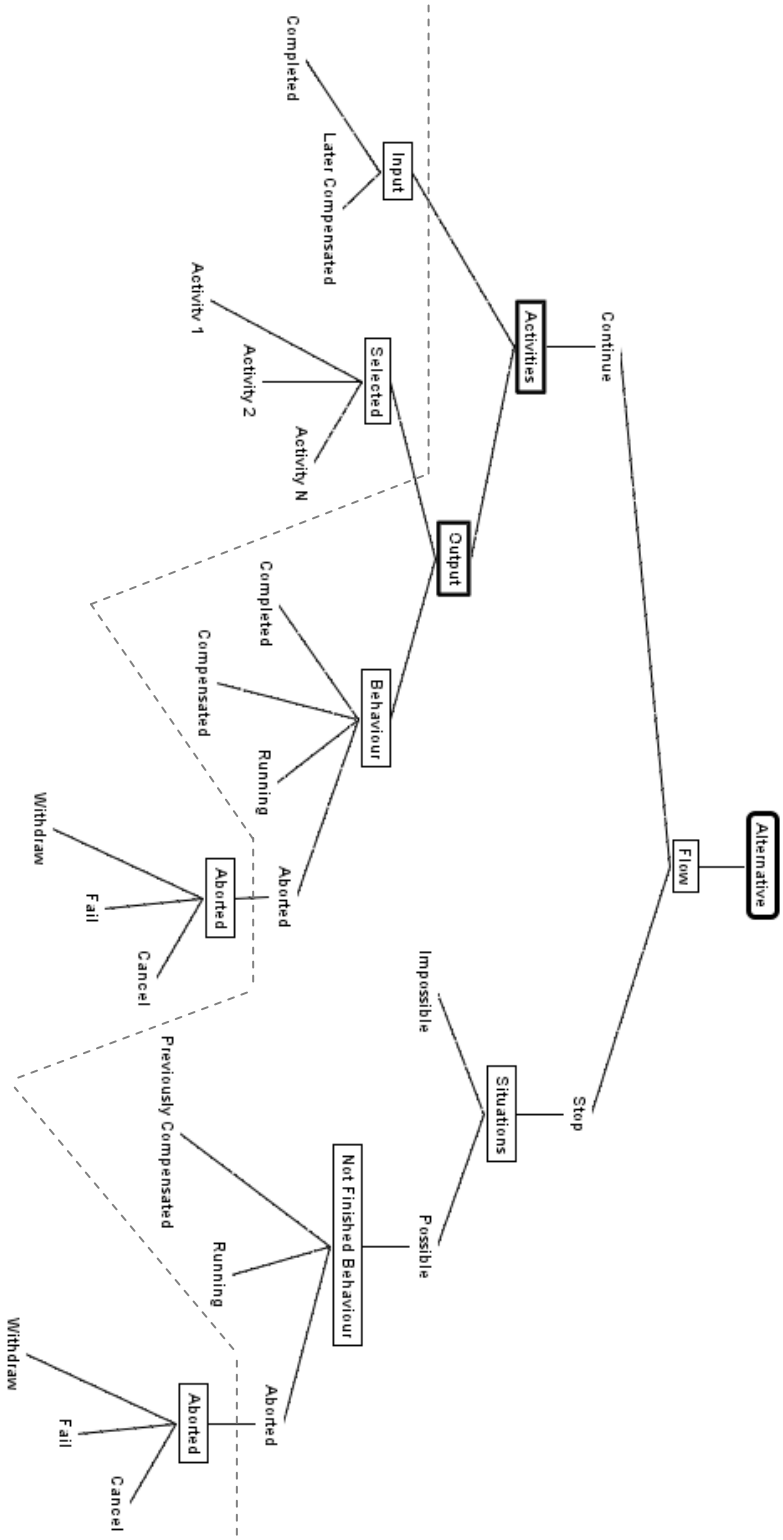


Figure 6.5. Alternative classification tree

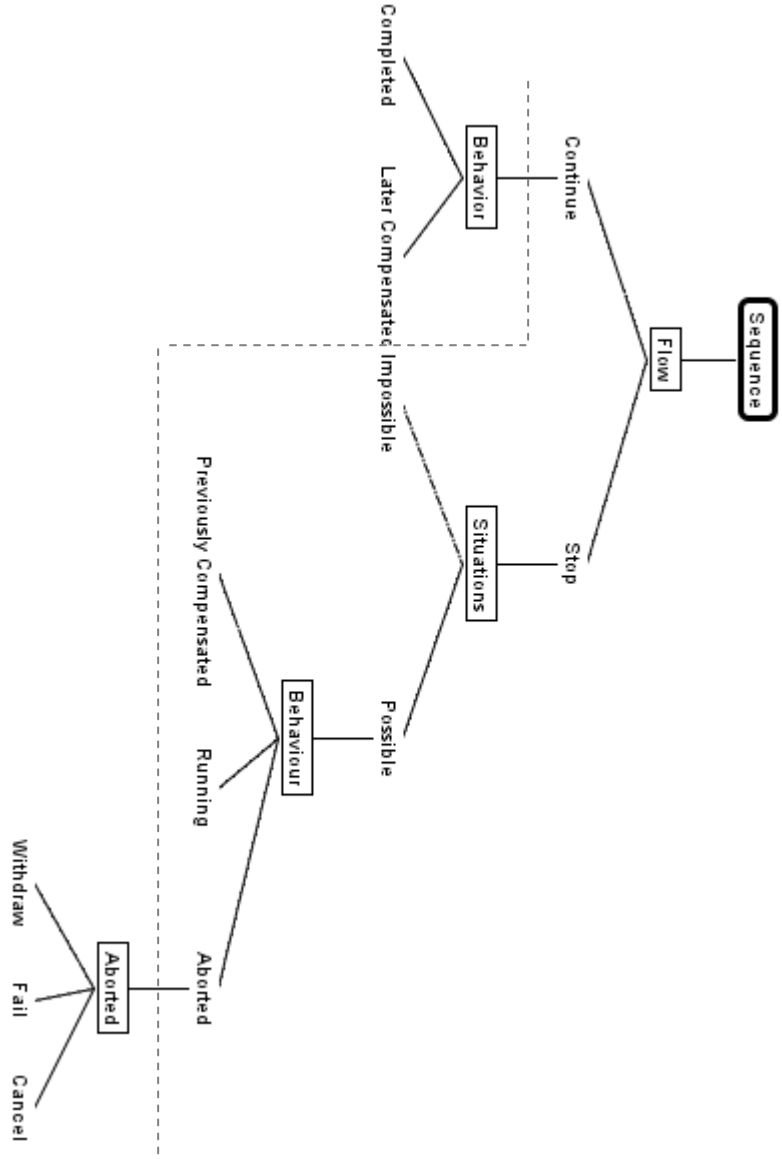


Figure 6.6. Sequence classification tree

6.3.3. Data dependency: Write

We classify the write dependency in a different family because it is not focus on the control of flow execution but in the data used by the activities. Write dependency relates the activities that modify a data element (*Definer*) with the activities that use the data element (*User*). The classification tree is shown in Figure 6.7.

In both *Definer* and *User* classes, we are interested in which alternative define/use (*Selected*) the data element and its behaviour (*Behaviour*).

In the *Definer* class we identify two orthogonal classes regarding the behaviour. *State* defines the way the activity finished (*Completed*, *Aborted*, *Compensated*). Note that for testing purposes it is not considered neither when the definer activity was compensated (*previously* or *later*) nor the way of abort (*withdraw*, *fail*, *cancel*). But it does is important to identify if the data element was modified or not (*Modification*). It could be, for example, that due to the business logic of the activity, the data element is not modified even if the activity was completed. On the other hand, it could be that the data element is modified even if the activity was cancelled. All those scenarios have to be taken into account for testing purposes.

In the *User* classes, we focus on the finished behaviours of the user activities. It is not relevant from a testing point of view if an activity that does not modify a data element does not achieve the completed state. So we identify the possible values for the *Finished Behaviour*: *Completed* and *Compensate*. Now it is important when the data element was modified for compensation (*Compensation Order*). The user activity can read the data element before the definer activity is compensated (*Previously*) or, in the other case, it can read the data element once the definer has been compensated (*Later*). Note that the data element could have different value because the definer activity can modify the data element when execute the compensation. Also note that the *Compensation Order* class only has sense if the *Compensated* value for the definer state is selected.

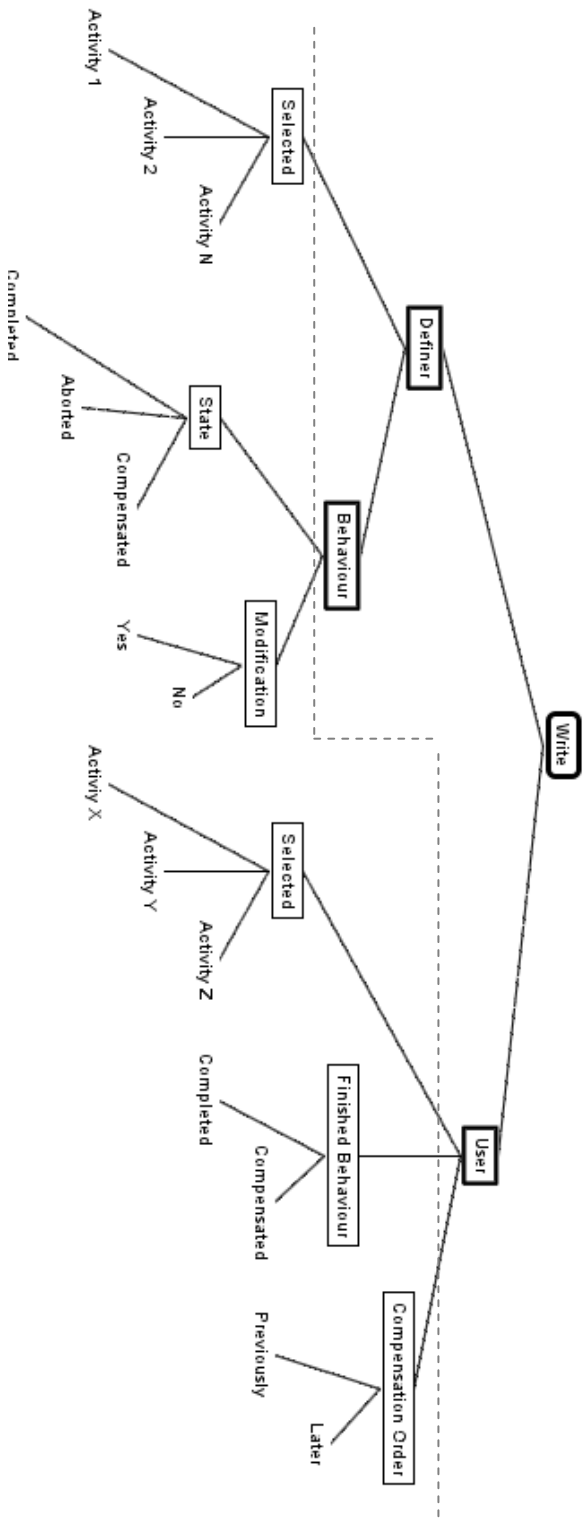


Figure 6.7. Write classification tree

6.4. Test case design

The goal of the test case design process is to achieve a suitable test suite. As was explained in Section 3.2.1, 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 previously identified by some test technique. And all the test cases (test suite) must cover (exercise) all the identified test coverage items. In this chapter, 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, 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 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 behaviour that all involved activities in the dependency must follow. It is generated by combining its Primitive TCIs using different combination criteria according to the *depth dimension*.

Figure 6.8 illustrates an example of Combined TCI (states are specified in brackets and transitions between dashes) derived from combining two Primitive Test Coverage Items.

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. Section 6.4.1 presents different criteria for the use and combination of the Primitive TCIs. The strategy used to organize and to combine the Combined TCIs in scenarios that define a test case is presented in Section 6.4.2.

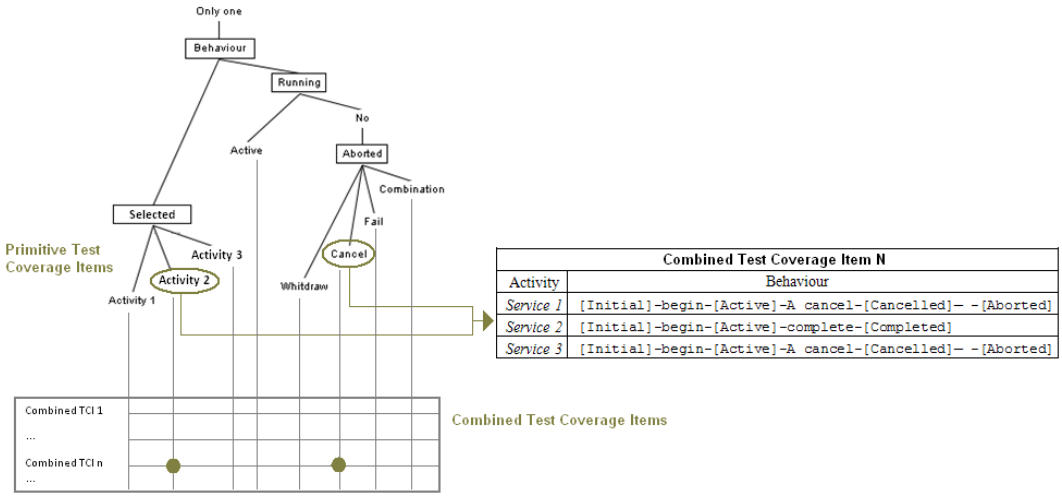


Figure 6.8. Combination of Primitive TCIs

6.4.1. 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 adjusting the testing thoroughness.

Primitive TCI selection (level)

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 6.2), then this criterion requires using the 19 Primitive TCIs which are found in the tree.
- *Weak level*: This level, called *N-1 level*, requires covering all non-elementary classes but only one value of each elementary class. In this criterion, a random leaf node (from children nodes) is selected. For example in the Merge CT, this criterion requires using 10 Primitive TCIs: one value for each class *Selected*, *Finished Behaviour* and *Aborted,* and *Valid*, plus the 9 rest values of the non-elementary classes. The number of Primitive TCI may be 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. As example, the dotted line in the Figure 6.2 separates the N level to N-1 level in the *Merge* dependency.

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, i.e., *each-used* coverage, does not enforce any requirement on how the Primitive TCIs are combined. The more complex coverage criteria, such as *pair-wise* or *N-wise* coverage, is concerned with (sub-) combinations of interesting values of different parameters [129]. As example, we define the boundaries and an intermediate value of such range of combinations criteria: the *each-used*

(simplest), *pair-wise* (intermediate) and *N-wise* (strongest). Note that other intermediate criteria can be used such as 3-wise, 4-wise, etc. [129].

- *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.

- *Pair-wise*: it requires that all possible pair combinations between the selected Primitive TCIs to be included in the set of Combined TCIs derived from this dependency. This criterion subsumes the *each-used* 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* and *pair-wise* criteria.

Handling composite dependencies

The Primitive TCIs (leaf nodes) 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.

Criteria comparison

Figure 6.9 depicts the four set of Combined TCIs generated by combining the orthogonal criteria of level (*strong* and *weak*) and combination (*each-used* and *N-wise*) for the Merge CT. 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. Then each row defines a Combined TCI achieved by applying the specified combination criterion.

With the *weak level* similar Combined TCIs are generated for both *N-wise* and *each-used* criterion. The reason is that *weak level* prunes the leaf nodes in the elementary classes *Selected*, *Aborted*, *Rest of Activities* and *All Activities.* This pruning added to the merge´s logic reduces the possibility of combination. The strong level generates more Combined TCIs than *weak level* since the former always selects more Primitive TCIs. This allows for exercising more exhaustively the dependency but the test effort (number of Combined TCI generated) can considerably grow if strong combination criteria are used.

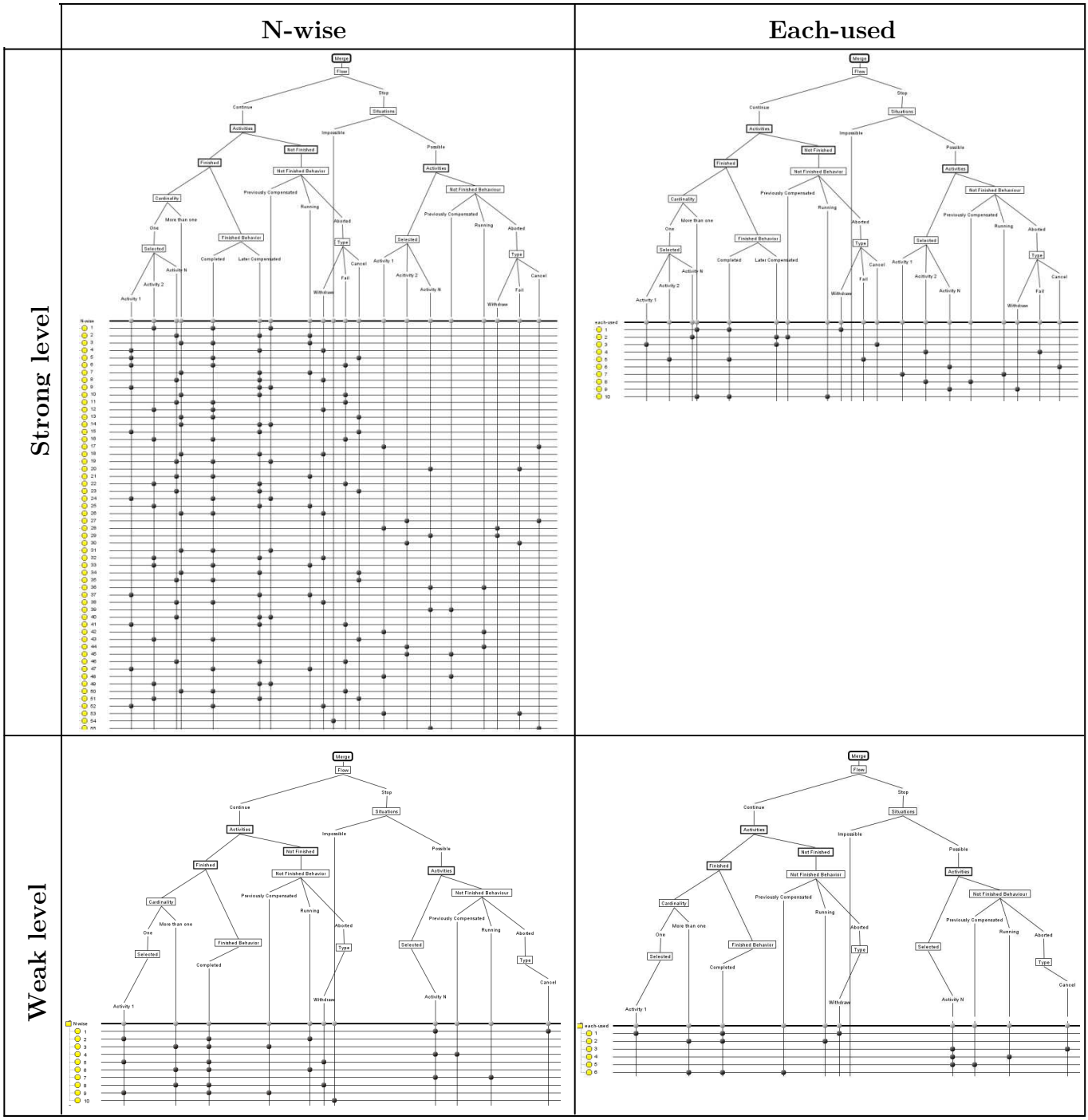


Figure 6.9. Combined TCIs generation

6.4.2. 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 [129]. BC is a determinist iterative strategy, which means, given a base test case the same test suite is produced every time. The first 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, smallest, or first. 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 fixed.

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 3.4). 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 6.10. 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 6.10. 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 6.11. The strategy gets the test coverage items which are not covered by 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 6.11. 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 6.12). This provides a set of generated test cases that cover all the Combined TCI.

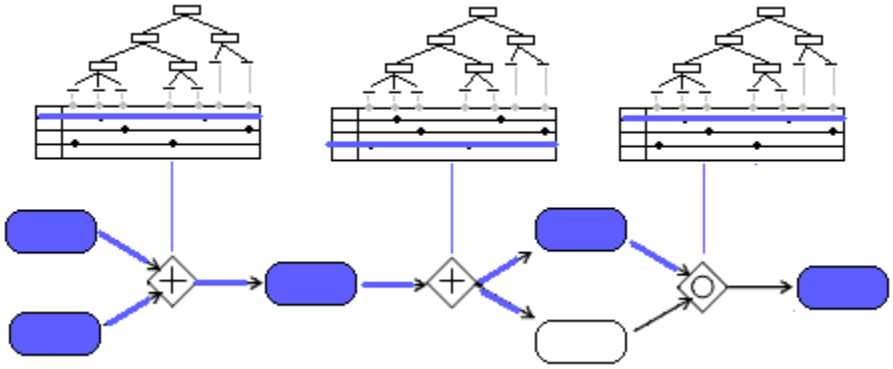


Figure 6.12. WS transaction test case

6.5. Case study: Web Travel Agency

In this section we evaluate the proposed criteria. We use the Travel Agent case study and model it according to the transaction model used by our method. The valuation is carried out in order to show that the proposed multi-dimensional criteria meeting 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

We have implemented a Travel Agency case study which is widely discussed in the literature [1, 130-132]. *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, the process determines which reservations to make, sending simultaneous requests to the transport, hotel and car rental providers. The transport can be by flight or by train. 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 sends to the customer the reservation confirmation and final itinerary details. Finally the Agency contacts the payment services to charge in the customer´s credit card of the total amount. The payment services also charges an extra 1% fare for using the 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). The average number of methods per class is 6.583 with an average of 11.83 lines per method. 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 serviceLogic classes implement the business logic of the activity, for example, checking the availability in a hotel and booking a room. The serviceWS wraps the logic class and other classes required by the service. Auxiliary classes are *Customer* and *Itinerary* that are used by all the services. There are also two classes regarding the transaction processing. *TAcontext* represents the data elements shared by the activities (itinerary, amount and customer data) while *TAflow* manages the execution of the all services.

6.5.1. Transactional modelling of the case study

We model the travel agency case study according to the transaction model presented in Section 3.1. Figure 6.13 depicts the modelling of the travel agency and shows some of the important activities, data elements and dependencies.

The services (and their activities), data elements and dependencies involved in the 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})

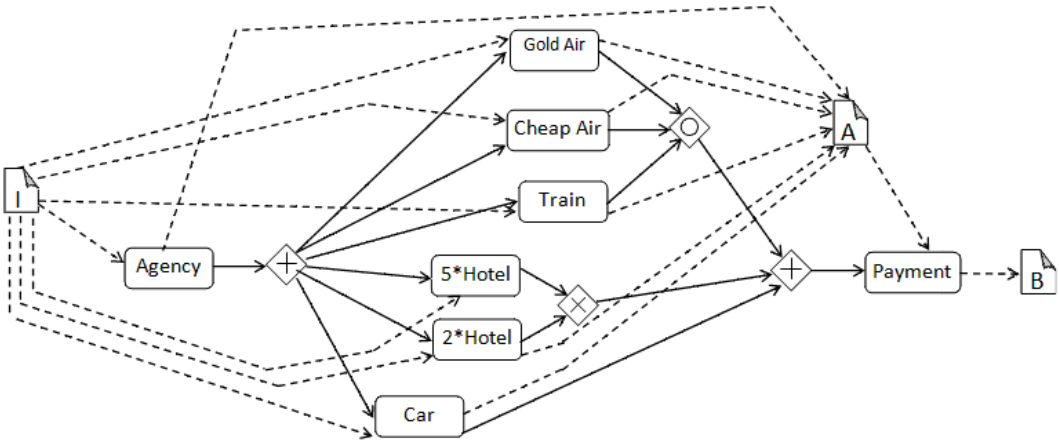


Figure 6.13. Web Travel Agency case study

6.5.2. Experimental parameters

Existing works on testing web services focus on the unit testing of the flow management, commonly a WS-BPEL process [65, 69, 133-135]. But they do not address the evaluation of fault detection in a particular implementation of services. We measure the effectiveness of the testing method as the degree on which the generated test cases are able to reveal defects injected in a concrete implementation (described above) of the whole 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 web service 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, MC=KM/TM * 100. Mutation score is, therefore, an objective measure to evaluate the effectiveness of a test suite.

To apply mutation to our *Travel Agency* application, we have used the MuJava tool [136]. MuJava 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 our *Travel Agency* application. The number of mutants generated by each Java class is shown in Table 6.2.

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
Auxiliary	Customer	78	59	19
Auxiliary	Itinerary	7	0	7
Auxiliary	PaymentTransfer	10	8	2
Transaction	TAcontext	59	14	45
Transaction	TAflow	217	211	6

Table 6.2. Generated mutants

To evaluate the requirement RQ2, we assess the cost-benefit relation of using the different criteria for generating the Combined TCIs, and thus, 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), say 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 CBN.

6.5.3. Results

We obtained the results 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 it generates the set of Combined TCIs. The number of Combined TCIs generated for each dependency and each combination of criteria are shown in Table 6.3. As was expected, the *strong level* noticeable increase 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 increment is only remarkable when this strategy is combined with the *strong level*. Also we see that the number of Combined TCI is very similar with the *weak level* criterion independently of the way of combination.

ID	Dependency	Strong level N-wise	Weak level N-wise	Strong level each-use	Weak level each use
D1	Fork	42	7	12	7
D2	Exclusion	17	5	9	5
D3	Merge	41	10	10	6
D4	Join	17	5	9	5
D5	Sequence	8	5	8	5
D6	Sequence	8	5	8	5
D7	Write	126	1	3	1
	Total	259	38	59	34

Table 6.3. Combined TCIs generated by the 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

transaction and then executes the algorithms shown in Section 5.2 to generate the test cases. Each test case specifies the behaviour that the activities have to follow during the execution. Such behaviours are described in a text file for each test case.

The third step is to automatically execute the test cases in the 2.507 mutated versions. 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 was obtained. The mutation score and cost-benefit results are summarized in Table 6.4. Mutation scores grouped by type of class are shown in Table 6.5. 'S' and 'W' means *strong level* and *weak level* respectively, while 'N' and 'E' respectively means *N-wise* and *each-used*. Information about number of killed mutants, alive mutants, traditional mutants score and class mutation score for each class is displayed in Annex D.

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 6.4. 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 6.5. Results by type of class

6.5.4. Discussion

Regarding RQ1, the results presented in Table 4 show the effectiveness of the method measured in terms of the mutation score. All test suites achieve a mutation score greater than 65% and even, three of the four achieve a 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 6.5 shows that the *weak level* criterion achieve notably inferior mutation scores, especially in reservation and auxiliary classes. These two types of classes are less complex than the other, so 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 the number of test cases generated (TC). The benefit (killed mutants, KM) also differs in the 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-used* criteria, this combination achieves the lowest mutation score. At the other end of the spectrum, *strong level / N-wise* have the worst CBN value but

achieve the best mutation score. In the middle we found the *weak level / N-wise* and *weak level / each- used* combinations that, with a reasonable value of CBN, achieve a high mutation score. So these results give a metric to the tester in order to decide the test effort to use depending on what factor (cost or benefit) want give priority to.

In relation to the RQ3, we consider the two types of injected faults: traditional mutants and class mutants. According to the results summarized in Table 6.5, we see that the mutation score achieved in the traditional mutants is mostly greater than the one achieved in the class mutation. The average mutation scores of all classes (see Appendix D) show the same tendency. So the results seem to show that the type of fault influences the effectiveness of the method.

We identify two main limitations and threads to validity of this evaluation. Firstly the mutation technique simulates the faults that could appear during the development of WS Transaction based application. As far we do not have information about actual faults, we do not how representative these injected faults are. But 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 [137]. This contributes to mitigate this threat. Secondly, as is usual in software engineering experiments, there is also the question of how representative the case study is. This work tried to mitigate that thread by using a case study widely accepted in the literature.

6.6. Summary

In this chapter we have presented 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 method elaborates a classification-tree analysis for each kind of dependency in order to identify the relevant test conditions, and subsequently, to define the test coverage items 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: *Travel Agency*. Evaluation results showed that the proposed criteria have the potential to design effective test cases for WS transaction and to allow the tester to adjust the method in terms of 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 adjusting the intensity of the test process by taking into account the time and effort limitations. It allows the tester to prioritize the tests by firstly using low-effort criteria and subsequently complement them when additional test effort can be applied.

Chapter 7

Conclusions

Life has given me strong blows. I could have become vulnerable and shot myself or I could look up to the sky and carry on. I preferred the second option

Manuel Preciado

This chapter presents the conclusions of this thesis. It outlines the main contributions of this work and also gives a critical analysis of the proposed methods. It also sets the directions for future research work.

7.1. Synthesis and results

This thesis has investigated into the issue of testing the WS transactions — a key issue that has not been given attention to by the current research.

Due to the existence of different approaches to manage WS transactions, we firstly developed and evaluated the Abstract Transaction Model (AbTM). AbTM defines a transaction as a set of activities and a set of dependencies between those activities. AbTM also identifies four roles that are commonly present in the transaction life-cycle: initiator, executor, coordinator and terminator. AbTM has the potential to capture the behaviour of a WS transaction independently of the underlying standard or model. It therefore, serves as a template for existing transactions model and standards and provides an easy and uniform way for testing different WS transactions.

The second main contribution of this thesis is the design and development of the Framework for Testing Transactions (F2F). F2T, inspired by the risk-based testing methodologies, has been devised to organize all the concepts involved in the process of test case design of WS transaction. It encompasses the concepts from the transaction definition (using the AbTM) to the test case generation. F2T identifies a set of hazards and develop techniques that systematically address those hazards.

The framework identifies three orthogonal dimensions of testing WS transactions (level, feature, depth) according to the basic test concepts (test level, test conditions, test coverage items). These have been used in testing the *participant level* (Chapter 4) as well as *transaction level* (Chapter 5 and Chapter 6).

At the *participant level*, we proposed a model-based testing method that focuses on the executor role to automatically generate test cases for testing the failures and reliability of WS transaction standards. The proposed test approach was implemented as a prototype system in which various test cases were automatically generated and mapped to WS

transaction standards. The evaluation was performed using the case study of *Nigh Out*, which is an open source WS-BA-based application provided by Jboss. The experiments showed that our approach can be used to define different test cases and test the reliability and failures of different WS transaction standards.

At the *transaction level*, we proposed two different approaches. Firstly (Chapter 5) we presented a set of test criteria to guide the test case generation. The criteria are based in the logical conditions defined by the dependencies that manage the execution of the activities primitive tasks. The proposed method was used in an industrial case study, the *Cajastur Insurance Application (CIA)*. The obtained feedback showed the viability of the method.

A new approach was defined (Chapter 6) that dealt with the *transaction level* by taking into account the limitations of the previous method (in Chapter 5). This approach also generates test cases according to the dependencies between the activities involved in a WS transaction. In this case, 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 (from *each-used* to *N-wise*). 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 when additional test efforts are required.

7.2. Critical analysis and future work

This section provides a critical analysis of the methods proposed in the thesis and also defines the perspectives for future work.

Testing WS transactions is a non-trivial research issue given the distributed, dynamic and loosely coupled nature of the process. The Framework for Testing Transactions (F2T) identified a set of seven properties (*Composition*, *Dependency*, *Recovery*, *Consistency*, *Visibility*, *Durability*, and *Controllability*) that should be tested in order to ensure the correct behaviour of the whole transaction. The hazards that imperil such properties are organized according to the test dimensions (*level*, *feature* and *depth*). In this thesis we have addressed *Composition*, *Dependency* and *Controllability* properties. The remaining properties motivate interesting research topic which can be addressed in future research work.

F2T relies on the capability of the Abstract Transaction Model (AbTM) to capture the behaviour of existing transaction models and standards. AbTM has been designed after an in-depth study of the existing solutions for managing WS transactions. Currently BTP, WS-BA and WS-COOR transaction standards have been modelled through the AbTM. In future, we intend to study the capability of AbTM to model transaction-based applications running under non-transaction standards such as WS-BPEL [89].

In relation to testing the *participant level*, the evaluation of the test case execution is based on the *system outcome*. Future works should also take into account the *user outcome* to deliver the verdict. Furthermore, the proposed method applies transition test criterion that ensures the coverage of all transitions and states specified in the AbTM. The method however does not guarantee the code coverage. As a part of the future research work we plan to enhance the prototype system in order to monitor the execution

of the code. Finally, the current method is focus on the executors. A future work is to deal with the rest of the roles involved in a WS transaction.

Finally, in relation to the methods proposed to test the *transaction level*, it would useful to have a tool that provides support for automating the test case generation. In the Classification-Tree based approach, future work should evaluate different strategies to compose the test cases. In addition, although the data is taking into account in the Write dependency, more specific analysis of the data patterns is clearly a still open issue to be addressed.

Capítulo 8

Conclusiones

La vida me ha golpeado fuerte. Podía haberme hecho vulnerable y acabar pegándome un tiro, o podía mirar al cielo y crecer. Elegí la segunda opción

Manuel Preciado

Este capítulo presenta las conclusiones de la tesis. Se resumen las principales contribuciones de este trabajo de investigación y se discuten las limitaciones del los métodos propuestos. También se definen las líneas para futuros trabajos de investigación.

8.1. Resumen y resultados

Esta tesis ha centrado sus esfuerzos en la prueba de transacciones en servicios web, un elemento clave que no ha recibido atención en la investigación actual.

Dada la variedad de estándares y protocolos existentes para manejar transacciones en servicios web, esta tesis primeramente desarrolló y evaluó un Modelo Abstracto de Transacciones (AbtM). AbTM define una transacción como un conjunto de actividades y una serie de dependencias entre esas actividades. El modelo propuesto diferencia cuatro roles que están siempre presentes durante el ciclo de vida de una transacción: iniciador, ejecutor, coordinador y terminador. AbTM puede capturar el comportamiento de una transacción independientemente del protocolo que utilice. Por tanto, sirve como una plantilla genérica para modelar los actuales modelos de transacciones permitiendo un mecanismo sencillo y uniforme para realizar pruebas de diferentes transacciones en servicios web.

La segunda contribución de esta tesis es el diseño y desarrollo del Marco para Pruebas de Transacciones (F2T). La realización de este marco de trabajo estuvo inspirada por las metodologías de pruebas basadas en riesgo. F2T organiza todos los conceptos involucrados en el diseño de casos de prueba para transacciones en servicios web. Comprende los conceptos desde la definición de la transacción (usando el AbTM) hasta la generación de las pruebas. F2T identifica un conjunto elementos de riesgo y desarrolla técnicas de prueba sistemáticas para mitigarlos.

El marco de pruebas identifica tres dimensiones ortogonales en la prueba de transacciones (nivel, característica, profundidad) de acuerdo a los conceptos generales de las pruebas de software (nivel de prueba, condiciones de prueba, cobertura de elementos de prueba). Esta organización se ha utilizado para definir métodos de prueba en el *nivel participante* (Capítulo 4) así como el *nivel transacción* (Capítulos 5 y 6).

En el *nivel participante*, hemos propuesto una técnica de prueba basada en modelos que se centra en el rol ejecutor. El método permite la

generación automática de casos de prueba para detectar fallos y comprobar la fiabilidad de los estándares para transacciones en servicios web. Se implementó el método mediante un prototipo y fue evaluado utilizando el caso de estudio *Night Out*, una aplicación *open source* de Jboss que utiliza el estándar WS-BA. Los experimentos mostraron que se puede utilizar nuestro método para generar automáticamente casos de prueba adecuados para diferentes estándares.

Para el *nivel transacción* propusimos dos enfoques diferentes. Primero (Capítulo 5), presentemos un conjunto de criterios de prueba para guiar la generación de los casos de prueba. Estos criterios están basados en condiciones lógicas que se derivan de las dependencias (relaciones) existentes entre las diferentes actividades que componen la transacción. Este método se utilizó en un caso de estudio industrial, la *Aplicación para Seguros de Cajastur*. El *feedback* obtenido mostró la viabilidad del método propuesto.

En el Capítulo 6 propusimos un nuevo enfoque para el *nivel transacción* teniendo en cuenta las limitaciones del método anterior (Capítulo 5). El método también genera casos de prueba focalizados en las dependencias entre actividades, pero en este caso, se utiliza la técnica de análisis de árboles de clasificación (classification-tree analysis). Por cada dependencia, se genera un árbol con el objetivo de identificar las condiciones de prueba relevantes así como los elementos específicos de prueba. El objetivo es derivar casos de pruebas para ejercitar todas las dependencias. Propusimos dos familias de criterios de prueba ortogonales, una para la selección de los elementos de prueba, y otra para la combinación de dichos elementos. Para evaluar el método propuesto utilicemos un caso de estudio ampliamente presente en la literatura: la Agencia de Viaje Web. La evaluación mostró que los criterios propuestos tienen el potencial para diseñar buenos casos de prueba y permiten al ingeniero de pruebas ajustar el método en términos de efectividad, esfuerzo de las pruebas y relación coste-beneficio. Además, el método permite el diseño automático de los casos de prueba lo que reduce el tiempo y coste de diseño y mejora su efectividad. Los diferentes criterios permiten al ingeniero de pruebas ajustar la intensidad del proceso de pruebas teniendo en cuenta el tiempo y el esfuerzo. De esta

manera, puede priorizar las prueba usando primero criterios de bajo coste y posteriormente ir complementándolos con criterios más costosos si fuese necesario.

8.2. Análisis crítico y trabajo futuro

Esta sección analiza métodos propuestos en esta tesis y define las perspectivas de trabajo futuro.

Realizar pruebas para transacciones en servicios web es una ardua tarea debido a la naturaleza distribuida, desacoplada y dinámica del proceso. El Marco para Pruebas de Transacciones (F2T) identificó un conjunto de siente propiedas (*Composición*, *Dependencia*, *Recuperación*, *Consistencia*, *Visibilidad*, *Durabilidad* y *Control*) que deberían ser comprobadas para poder asegurar el correcto comportamiento de la transacción. Los riesgos que afectan a dichas propiedades han sido organizados en tres dimensiones de prueba (*nivel*, *característica* y *profundidad*). En esta tesis nos hemos centrado en las propiedades *Composición*, *Dependencias* y *Control*, por lo que una motivante línea de trabajo futuro sería explorar el resto de las propiedades propuestas.

F2T se basa en la habilidad del Modelo Abstracto de Transacciones (AbTM) para capturar el comportamiento de los actuales estándares de transacciones en servicios web. AbTM se diseñó tras un profundo estudio de las soluciones existentes para el manejo de este tipo de transacciones. Actualmente se han modelado los estándares BTP, WS-BA y WS-COOR. En el futuro tenemos planteado estudiar la capacidad del AbTM para modelar aplicaciones basadas en transacciones que no ejecutan un estándar específico de transacciones, como puede ser el WS-BPEL[89].

En relación con el método propuesto para probar el *nivel participante*, la evaluación de los casos de prueba se basó en la salida del sistema. Un interesante trabajo futuro sería tener en cuenta también la salida del usuario para definir el veredicto del caso de prueba. Además, el método aplica cobertura de transiciones lo que asegura una cobertura total de los estados y transiciones especificados en el AbTM. Sin embargo, el método no puede

asegurar ninguna cobertura de código ya que no hay un enlace formal entre el AbTM y el código de la aplicación. Como parte de nuestro trabajo futuro, tenemos planteado mejorar el prototipo con el objetivo de poder monitorizar la ejecución del código y así obtener información sobre su cobertura. Finalmente, el método actualmente se centra en el rol del ejecutor. Un futuro trabajo sería abordar el resto de los roles involucrados en la transacción.

Con respecto a los métodos propuestos para probar el nivel transacción, sería útil el desarrollo de herramientas que permiten la generación automática de los casos de prueba. En el enfoque basado en árboles de clasificación, trabajos futuros deberían evaluar otras posibles estrategias para componer los casos de prueba. Además, aunque el uso de la información se tiene en cuenta en la dependencia Write, un análisis más específico de los patrones de uso de la información es claramente todavía un objetivo a tratar.

Appendices

A. Algorithm ABC-DC

```
Algorithm ABC-DC (input wT: web_transaction; output ts: test_suite)
{
     s_stack: stack of activities
     a: activity
     tc: test case
     ts: test suite

     a_stack = A(wT)
     while (a_stack is not empty)
     {
          a = a_stack.pop

          if (there is not tc in ts where begin(a) = true)
          {
            tc= empty;
            tc+= (begin(a)=true);
            tc+= BC_true (a);
            ts+=tc;
          }

          if (there is not tc in ts where begin(a) = false)
          {
            tc= empty;
            tc+= (begin(a)=false);
            tc+= BC_false (a);
            ts+=tc;
          }
     }
     return tc;
}
```

```
auxiliary procedure BC_true (input a: activity; output tc:
test_case)
{
     tc: test_case
     a2= activity
     tk= task
     tc=empty;
     if (BeginCond(a) = true)
     {
            return tc
     }
     else
     {
            for each condition c in BeginCond(a)
            {
                  a2=activity involved in c
                  tk = task involved in c
                  if (tk== begin)
                        tc+= Begin(a2)=true
                  else if (tk== commit)
                        tc+= Commit(a2)=true
                  else
                        tc+=Abort(a2)=true
                  tc+=BC_true(a2)
                  if (BeginCond(a) is true when c is true)
                        return tc;
            }
     }
}
```

```
auxiliary procedure BC_false (input a: activity; output tc:
test_case)
{
      tc: test_case
      a2: activity
      tk: task
      tc=empty;

      if (BeginCond(s) = false or BeginCond(s) is empty )
      {
            return tc
      }
      else
      {
            for each condition c in BeginCond(s)
            {
                  a2=activity involved in c
                  tk = task involved in c
                  if (tk== Begin)
                        tc+= Begin(a2)=false
                  else if (tk== commit)
                        tc+= Commit(a2)=false
                  else
                        tc+=Abort(a2)=false
                  tc+=BC_false(a2)
                  if (BeginCond(a) is false when c is false)
                  return tc;
            }
      }
}
```

B. OPC Test cases

	CRS	**OI**	**cOI**	**PCC**	**cPCC**	**CA**	**DF**	**DT**
TC1.1	Begin, Commit	Begin, Commit	-	Begin, Commit	-	Begin, Commit	Begin	Begin
TC1.2	Begin, Commit	-	-	Begin, Commit	-	-	-	-
TC1.3	Begin, Commit	Begin, Commit	-	-	-	-	-	-
TC1.4	Begin	Begin, Commit, Abort	-	Begin, Commit	Begin	-	-	-
TC1.5	Begin	-	Begin	Begin, Commit, Abort	Begin, Commit	-	-	-
TC1.6	-	Begin	-	Begin	-	-	-	-

Test conditions for OPC application using ABC-DC criterion

	CRS	**OI**	**cOI**	**PCC**	**cPCC**	**CA**	**DF**	**DT**
TC 2.1	Begin, Commit	Begin, Commit	-	Begin, Commit	-	Begin, Commit	-	Begin, Commit
TC 2.2	Begin, Commit	Begin, Commit	-	Begin, Commit	-	Begin, Commit	Begin, Commit	Begin, Commit Abort
TC 2.3	Begin, Commit	Begin, Commit	-	Begin, Commit	-	Begin, Commit	Begin, Commit Abort	Begin, Commit
TC 2.4	Begin, Commit	Begin, Commit	-	Begin, Commit	-	Begin, Commit Abort	-	-
TC 2.5	Begin, Commit	Begin, Commit Abort	-	Begin, Commit	Begin, Commit	-	-	-
TC 2.6	Begin, Commit	Begin, Commit Abort	-	Begin, Commit	Begin, Commit Abort	-	-	-
TC 2.7	Begin, Commit	Begin, Commit	Begin, Commit	Begin, Commit Abort	-	-	-	-
TC 2.8	Begin, Commit	Begin, Commit	Begin, Commit, Abort	Begin, Commit Abort	-	-	-	-
TC 2.9	Begin, Commit, Abort	-	-	-	-	-	-	-

Test conditions for OPC application using ACAC-DC criterion

	CRS	**OI**	**cOI**	**PCC**	**cPCC**	**CA**	**DF**	**DT**
TC 3.1	Begin, Commit	Begin, Commit	-	Begin, Commit	-	Begin, Commit	-	Begin, Commit
TC 3.2	Begin, Commit	Begin, Commit	-	Begin, Commit	-	Begin, Commit	Begin, Commit	-
TC 3.3	Begin, Commit	-	-	Begin, Commit	-	-	-	-
TC 3.4	Begin, Commit	Begin, Commit	-	-	-	-	-	-
TC 3.5	Begin, Commit	Begin, Commit Abort	-	Begin, Commit	Begin, Commit	-	-	-
TC 3.6	Begin, Commit	Begin, Commit, Abort	-	Begin	-	-	-	-
TC 3.7	Begin, Commit	Begin, Commit	Begin Commit	Begin, Commit, Abort	-	-	-	-
TC 3.8	Begin, Commit	Begin	-	Begin, Commit, Abort	-	-	-	-
TC 3.9	Begin	-	-	-	-	-	-	-

Test conditions for OPC application using ACC-MCDC criterion

C. OPC mutations

MUT1	$BeginCond(s_i)$	$CommitCond(s_i)$	$AbortCond(s_i)$
CRS	*	*	*
OI	$C(CRS)$	*	*
cOI	$A(PCC) \wedge C(OI)$	$A(PCC)$	*
PCC	$C(CRS)$	*	*
$cPCC$	$B(OI) \wedge C(PCC)$	$A(OI)$	*
CA	$C(OI) \wedge C(PCC)$	*	*
DF	$C(CA)$	*	$C(DT)$
DT	$C(CA)$	*	$C(DF)$

Examples of specification mutation using ARO

MUT2	$BeginCond(s_i)$	$CommitCond(s_i)$	$AbortCond(s_i)$
CRS	*	*	*
OI	$C(CRS)$	*	*
cOI	$A(PCC)$	$A(PCC)$	*
PCC	$C(CRS)$	*	*
$cPCC$	$A(OI) \wedge C(PCC)$	$A(OI)$	*
CA	$C(OI) \wedge C(PCC)$	*	*
DF	$C(CA)$	*	$C(DT)$
DT	$C(CA)$	*	$C(DF)$

Examples of specification mutation using MAO

D. Travel Agency 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

Bibliography

[1] 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.

[2] M. P. Machulak, J. J. Halliday, and M. C. Little. "Metadata Support for Transactional Web Services". In *EDOC Conference Workshop, 2007. EDOC '07. Eleventh International IEEE*, pp. 53-56, 2007.

[3] T. Reuters. (2012, 25/09/2012). *Journal Citation Reports*. Available: http://thomsonreuters.com/products_services/science/science_products/a-z/journal_citation_reports/

[4] ERA. (2012, 29/09/2012). *The Computing Research and Education Association of Australasia, CORE*. Available: http://core.edu.au/

[5] Microsoft. (2012). *Microsoft Academic Research*.

[6] BOE. (2011, 01/11/2012). *Real Decreto 99/2011, de 28 de enero, por el que se regulan las enseñanzas oficiales de doctorado*. Available: http://www.boe.es/buscar/doc.php?id=BOE-A-2011-2541

[7] I. Gartner. (2012). *IT Glossary*. Available: http://www.gartner.com/it-glossary/service-oriented-architecture-soa/

[8] L. Boris. "Defining SOA as an architectural style", *IBM developerWorks,* 2007.

[9] OASIS. (2011, 24/09/2012). *SOA Reference Model*.

[10] 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.

[11] W3C. (24/08/2012). *World Wide Web Consortium(W3C)*. Available: http://www.w3.org/

[12] W3C. (2007, 24/08/2012). *Simple Object Access Protocol (SOAP)*. Available: http://www.w3.org/TR/soap/

[13] W3C. (2001, 24/08/2012). *Web Services Description Language (WSDL)*. Available: http://www.w3.org/TR/wsdl

[14] W3C. (2004, 24/08/2012). *Web Services Glossary*. Available: http://www.w3.org/TR/ws-gloss/

[15] OASIS. (24/08/2012). *Organization for the Advancement of Structured Information Standards*. Available: https://www.oasis-open.org/

[16] T. Wang, J. Vonk, B. Kratz, and P. Grefen. "A survey on the history of transaction management: from flat to grid transactions", *Distrib. Parallel Databases,* vol. 23, pp. 235-270, 2008.

[17] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

[18] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*, Morgan Kaufmann Publishers, 2009.

[19] A. K. Elmagarmid. *Database transaction models for advanced applications*: Morgan Kaufmann Publishers, 1992.

[20] C. Mohan and B. Lindsay. "Efficient commit protocols for the tree of processes model of distributed transactions", *SIGOPS Oper. Syst. Rev.,* vol. 19, pp. 40-52, 1985.

[21] B. W. Lampson and D. B. Lomet. "A New Presumed Commit Optimization for Two Phase Commit". In *Proceedings of the 19th International Conference on Very Large Data Bases*, pp. 630-640, 1993.

[22] X. Yao and A. J. Glenstrup, "Distributed Transaction Management in SOA-based System Integration," IT University of Kopenhagen, 2007.

[23] L. Gao, S. D. Urban, and J. Ramachandran. "A survey of transactional issues for Web Service composition and recovery", *Int. J. Web Grid Serv.,* vol. 7, pp. 331-356, 2011.

[24] E. B. Moss. "Nested Transactions: An Approach to Reliable Distributed Computing", *Massachusetts Institute of Technology,* 1981.

[25] H. Garcia-Molina and K. Salem. "Sagas". In *SIGMOD 87*, pp. 249-259, 1987.

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

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

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

[29] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. "A Multidatabase Transaction Model for InterBase". In *Proceedings of*

the 16th International Conference on Very Large Data Bases, pp. 507-518, 1990.

[30] M. Younas, B. Eaglestone, and R. Holton. "A Formal Treatment of the SACReD Protocol for Multidatabase Web Transactions". In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pp. 899-908, 2000.

[31] J. Warne. "An Extensible Transaction Framework: Technical Overview". ANSA Archi-tecture for Open Distributed Systems Project1993.

[32] B. Limthanmaphon and Y. Zhang. "Web service composition transaction management". In *Proceedings of the 15th Australasian database conference - Volume 27*, Dunedin, New Zealand, pp. 171-179, 2004.

[33] M. Little. "Transactions and Web services", *Commun. ACM,* vol. 46, pp. 49-54, 2003.

[34] M. Little and T. J. Freund. "Introducing WS-CAF—more than just transactions", *Web Services Journal,* vol. 3, pp. 52-55, 2003.

[35] B. Kratz. "Protocols for long running business transactions". Infolab, Tilburg University, 17, 2004.

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

[37] M. Little. "Web services transactions: Past, present and future". In *XML Conference and Exposition*, Philadelphia, USA, 2003.

[38] OASIS. (2006). *Web Services Composite Application Framework.* Available: https://www.oasis-open.org/committees/ws-caf

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

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

[41] OASIS. (2009). *Web Services Business Activity.* Available: http://docs.oasis-open.org/ws-tx/wsba/2006/06

[42] M. Little, J. Maron, and G. Pavlik. *Java transaction processing: design and implementation*, Prentice Hall PTR, 2004.

[43] C. Anis, S. Benjamin, H. Andreas, and M. Mira. "Reliable, Secure, and Transacted Web Service Compositions with AO4BPEL". In *Web Services, 2006. ECOWS '06. 4th European Conference on*, pp. 23-34, 2006.

[44] S. Chang-ai, E. el Khoury, and M. Aiello. "Transaction Management in Service-Oriented Systems: Requirements and a Proposal", *Services Computing, IEEE Transactions on,* vol. 4, pp. 167-180, 2011.

[45] M. Younas and K.-M. Chao. "A tentative commit protocol for composite web services", *Journal of computer and system sciences,* vol. 72, pp. 1226-1237, 2006.

[46] M. Schäfer, P. Dolog, and W. Nejdl. "An environment for flexible advanced compensations of Web service transactions", *ACM Trans. Web,* vol. 2, pp. 1-36, 2008.

[47] Z. Wenbing, L. E. Moser, and P. M. Melliar-Smith. "A Reservation-Based Extended Transaction Protocol", *Parallel and Distributed Systems, IEEE Transactions on,* vol. 19, pp. 188-203, 2008.

[48] J. E. Ferreira, K. R. Braghetto, O. K. Takai, and C. Pu. "Transactional Recovery Support for Robust Exception Handling in Business Process Services". In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pp. 303-310, 2012.

[49] C. Jiuxin, L. Junzhou, Z. Song, Z. Xiao, L. Bo, Z. Gongrui, and Z. Biao. "A Context-Aware Recovery Mechanism for Web Services Business Transaction". In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pp. 352-359, 2012.

[50] S. Choi, H. Kim, H. Jang, J. Kim, S. M. Kim, J. Song, and Y.-J. Lee. "A framework for ensuring consistency of Web Services Transactions", *Information and Software Technology,* vol. 50, pp. 684-696, 2008.

[51] M. Alrifai, P. Dolog, W. T. Balke, and W. Nejdl. "Distributed Management of Concurrent Web Service Transactions", *Services Computing, IEEE Transactions on,* vol. 2, pp. 289-302, 2009.

[52] F. Montagut, R. Molva, and S. Tecumseh Golega. "The Pervasive Workflow: A Decentralized Workflow System Supporting Long-Running Transactions", *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on,* vol. 38, pp. 319-333, 2008.

[53] M. von Riegen, M. Husemann, S. Fink, and N. Ritter. "Rule-Based Coordination of Distributed Web Service Transactions", *Services Computing, IEEE Transactions on,* vol. 3, pp. 60-72, 2010.

[54] J. Cao, B. Zhang, B. Mao, and B. Liu. "Constraint Rules-based Recovery for Business Transaction". In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pp. 282-289, 2010.

[55] Z. Honglei, C. Hua, Z. Wenbing, P. M. Melliar-Smith, and L. E. Moser. "Trustworthy Coordination of Web Services Atomic Transactions", *Parallel and Distributed Systems, IEEE Transactions on,* vol. 23, pp. 1551-1565, 2012.

[56] R. T. Khachana, A. James, and R. Iqbal. "Relaxation of ACID properties in AuTrA, The adaptive user-defined transaction relaxing approach", *Future Generation Computer Systems,* vol. 27, pp. 58-66, 2011.

[57] M. Bozkurt, M. Harman, and Y. Hassoun. "Testing Web Services: A survey". Department of Computer Science, King's College London, Technical Report TR-10-012010.

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

[59] A. T. Endo and A. d. S. Simão. "A Systematic Review on Formal Testing Approaches for Web Services". In *IV Brazilian Workshop on Systemati and Automated Software Testing*, Natal, Brasil, pp. 89-98, 2010.

[60] M. Palacios, J. Garcia-Fanjul, and J. Tuya. "Testing in Service Oriented Architectures with dynamic binding: A mapping study", *Inf. Softw. Technol.,* vol. 53, pp. 171-189, 2011.

[61] S. Bhiri, C. Godart, and O. Perrin. "Transactional patterns for reliable web services compositions". In *6th International Conference on Web Engineering*, Palo Alto, California, USA, pp. 137-144, 2006.

[62] ISO/IEC 9126-1 Software engineering, product quality 2001-06-15.

[63] J. El Hadad, M. Manouvrier, and M. Rukoz. "TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition", *Services Computing, IEEE Transactions on,* vol. 3, pp. 73-85, 2010.

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

[65] J. Garcia-Fanjul, C. de la Riva, and J. Tuya. "Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking". In *Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, pp. 127-130, 2006.

[66] H. M. Rusli, S. Ibrahim, and M. Puteh. "Testing Web Services Composition: A Mapping Study", *Communications of the IBIMA,* vol. 2011, 2011.

[67] A. T. Endo, A. da Simao, S. Souza, and P. Souza. "Web Services Composition Testing: A Strategy Based on Structural Testing of Parallel Programs", *TAIC PART '08. Testing: Academic & Industrial Conference,* 2008.

[68] A. Cavalli, T.-D. Cao, W. Mallouli, E. Martins, A. Sadovykh, S. Salva, and F. Zaïdi. "WebMov: A Dedicated Framework for the Modelling and Testing of Web Services Composition". In *IEEE International Conference on Web Services*, Florida, USA, 2010.

[69] 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, pp. 135-141, 2008.

[70] I. Rabhi. "Robustness Testing of Web Services Composition". In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 631-638, 2012.

[71] C.-a. Sun, Y. Shang, Y. Zhao, and T. Y. Chen. "Scenario-Oriented Testing for Web Service Compositions Using BPEL". In *Quality Software (QSIC), 2012 12th International Conference on*, pp. 171-174, 2012.

[72] Z. Hong and Z. Yufeng. "Collaborative Testing of Web Services", *Services Computing, IEEE Transactions on,* vol. 5, pp. 116-130, 2012.

[73] Y. Gwyduk, Y. Taewoong, and M. Dugki. "A QoS model and testing mechanism for quality-driven Web services selection". In *Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on*, p. 6 pp., 2006.

[74] V. Pretre, F. Bouquet, and C. Lang. "Using Common Criteria to Assess Quality of Web Services". In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pp. 295-302, 2009.

[75] D. Yuetang, P. Frankl, and C. Zhongqiang. "Testing database transaction concurrency". In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pp. 184-193, 2003.

[76] 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, pp. 78-87, 2005.

[77] V. Guarnieri, N. Bombieri, G. Pravadelli, F. Fummi, H. Hantson, J. Raik, M. Jenihhin, and R. Ubar. "Mutation analysis for SystemC designs at TLM". In *Test Workshop (LATW), 2011 12th Latin American*, pp. 1-6, 2011.

[78] C. Chin-Yao, H. Chih-Yuan, L. Kuen-Jong, and A. P. Su. "Transaction Level Modeling and Design Space Exploration for SOC Test Architectures". In *Asian Test Symposium, 2009. ATS '09.*, pp. 200-205, 2009.

[79] 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". In *Mechatronics and Automation, 2009. ICMA 2009. International Conference on*, pp. 2952-2956, 2009.

[80] 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.

[81] 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, pp. 76-94, 2008.

[82] N. Kokash and F. Arbab. "Formal Design and Verification of Long-Running Transactions with Eclipse Coordination Tools", *Services Computing, IEEE Transactions on,* vol. PP, pp. 1-1, 2011.

[83] M. Emmi and R. Majumdar. "Verifying Compensating Transactions". In *International Conference Verification, Model Checking, and Abstract Interpretation*, pp. 29-43 2007.

[84] 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, pp. 353-370, 2007.

[85] W. Gaaloul, S. Bhiri, and M. Rouached. "Event-Based Design and Runtime Verification of Composite Service Transactional Behavior", *Services Computing, IEEE Transactions on,* vol. 3, pp. 32-45, 2010.

[86] I. Saleh, G. Kulczycki, and M. B. Blake. "Formal Specification and Verification of Transactional Service Composition". In *Services (SERVICES), 2011 IEEE World Congress on*, pp. 474-481, 2011.

[87] J. Li, H. Zhu, and J. He. "Specifying and Verifying Web Transactions". In *International Conference on Formal Techniques for Networked and Distributed Systems*, pp. 149-168 2008.

[88] 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, p. 29, 2011.

[89] OASIS. (2007). *Web Services Business Process Execution Language v2.0*. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[90] I. Object Management Group. "OMG Unified Modeling Language Specification", ed, 2001.

[91] ISTQB. (2012, 03-03-2012). *Glossary of Terms*. Available: www.istqb.org

[92] R. Black. *Advanced Software Testing - Vol. 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager*, Rocky Nook, 2012.

[93] H. Zhu, P. A. V. Hall, and J. H. R. May. "Software unit test coverage and adequacy", *ACM Comput. Surv.,* vol. 29, pp. 366-427, 1997.

[94] S. Amland. "Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study", *The Journal of Systems and Software,* pp. 287-295, 2000.

[95] T. Kletz. "Hazop and Hazun: Identifying and Assessing Process Industry Hiizards", *Institution of Chemical Engineers,* 1992.

[96] A. Claesson. "A Risk Based Testing Process". In *QualityWeek Europe*, Brussels, Belgium, 2002.

[97] F. Redmill. "Exploring risk-based testing and its implications: Research Articles", *Softw. Test. Verif. Reliab.,* vol. 14, pp. 3-15, 2004.

[98] A. Vorster and L. Labuschagne. "A framework for comparing different information security risk analysis methodologies". In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, White River, South Africa, pp. 95-103, 2005.

[99] J. C. Bennet, G. A. Bohoris, E. M. Aspinwall, and R. C. Jall. "Risk analysis techniques and their application to software development", *European Journal of Operational Research,* pp. 467-475, 1995.

[100] J. V. Earthy. "Hazard and operability study as an approach to software safety assessment". In *Hazard Analysis, IEE Colloquium on*, pp. 5/1-5/3, 1992.

[101] H. Stallbaum, A. Metzger, and K. Pohl. "An automated technique for risk-based test case generation and prioritization", *International Conference on Software Engineering,* pp. 67-70, 2008.

[102] J. A. McDermid and D. J. Pumfrey. "A development of HAZARD analysis to aid software design", *COMPASS '94 'Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security'. Proceedings of the Ninth Annual Conference on,* 1994.

[103] B. Tekinerdogan, H. Sozer, and M. Aksit. "Software architecture reliability analysis using failure scenarios", *Journal of Systems and Software,* vol. 81, pp. 558-575, 2008.

[104] F. Crawley, M. Preston, and B. Tyler. *HAZOP : Guide to Best Practice: Guidelines to Best Practice for the Process and Chemical Industries*, Institution of Chemical Engineers, 2008.

[105] D. H. Stamatis. *Failure Mode and Effect Analysis: Fmea from Theory to Execution*, ASQ Quality Press, 2003.

[106] R. G. Dromey. "A model for software product quality", *Software Engineering, IEEE Transactions on,* vol. 21, pp. 146-162, 1995.

[107] M. Younas, B. Eaglestone, and R. Holton. "A Review of Multidatabase Transactions on The Web: From the ACID to the SACReD". In *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases*, pp. 140-152, 2000.

[108] 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, pp. 1-56, 2008.

[109] C. Guidi, R. Lucchi, and M. Mazzara. "A Formal Framework for Web Services Coordination", *Electronic Notes in Theoretical Computer Science,* vol. 180, pp. 55-70, 2007.

[110] S. A. Ehikioya and K. Barker. "A formal specification strategy for electronic commerce". In *Database Engineering and Applications Symposium, 1997. IDEAS '97. Proceedings., International*, pp. 201-210, 1997.

[111] S. Bhiri, O. Perrin, and C. Godart. "Ensuring required failure atomicity of composite Web services". In *Proceedings of the 14th international conference on World Wide Web*, Chiba, Japan, pp. 138-147, 2005.

[112] M. Pol, R. Teunissen, and E. Van Veenendaal. *Software Testing: A Guide to the TMap Approach*, Addison-Wesley, 2002.

[113] E. Lehmann and J. Wegener. "Test Case Design by Means of the CTE XL". In *8th European International Conference on Software Testing, Analysis & Review*, Kopenhagen, Denmark, 2000.

[114] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. "Generating Test Data From State-based Specifications", *Journal of Software Testing, Verification and Reliability,* vol. 13, pp. 25-53, 2003.

[115] Jboss. (2006, 29 Nov 2011). *Jboss Transactions.* Available: http://www.jboss.org/jbosstm

[116] GlassFish. (2005). *JAX-WS.* Available: http://jax-ws.java.net/

[117] RCTA/DO-178B. "Software Considerations in Airborne Systems and Equipment Certification". RTCA, Washington, USA1992.

[118] M. Grochtmann and K. Grimm. "Classification trees for partition testing", *Software Testing, Verification and Reliability,* vol. 3, pp. 63-82, 1993.

[119] V. Atluri, W.-k. Huang, and E. Bertino. "An Execution Model for Multilevel Secure Work-flows". In *11th IFIP Working Conference on Database Security*, 1997.

[120] P. K. Chrysanthis and K. Ramamritham. "Synthesis of extended transaction models using ACTA", *ACM Trans. Database Syst.,* vol. 19, pp. 450-491, 1994.

[121] G. J. Myers. *The art of software testing*, Wiley, New York :, 1979.

[122] P. E. Ammann and P. E. Black. "A Specification-Based Coverage Metric to Evaluate Test Sets". In *4th IEEE International Symposium on High-Assurance Systems Engineering*, Washington, DC., pp. 239-248, 1999.

[123] P. E. Black, V. Okun, and Y. Yesha. "Mutation Operators for Specifications". In *The Fifteenth IEEE International Conference on Automated Software Engineering* Grenoble, pp. 81-81, 2000.

[124] *Cajastur*. Available: https://www.cajastur.es/

[125] *Liberbank*. Available: http://www.liberbank.es/

[126] *Business Process Model and Notation*. Available: http://www.bpmn.org/

[127] T. Y. Chen and P. L. Poon. "On the effectiveness of classification trees for test case construction", *Information and Software Technology,* vol. 40, pp. 765-775, 1998.

[128] 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*, p. 81, 1997.

[129] M. Grindal, J. Offutt, and S. F. Andler. "Combination testing strategies: a survey", *Software Testing, Verification and Reliability,* vol. 15, pp. 167-199, 2005.

[130] 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, pp. 329-335, 2002.

[131] 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, pp. 12-17, 2008.

[132] 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, pp. 58-66, 2011.

[133] J. Yan, L. Zhongjie, Y. Yuan, S. Wei, and Z. Jian. "BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach". In *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pp. 75-84, 2006.

[134] 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, pp. 659-664, 2011.

[135] 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, pp. 107-112, 2008.

[136] Y.-S. Ma, J. Offutt, and Y. R. Kwon. "MuJava: an automated class mutation system: Research Articles", *Softw. Test. Verif. Reliab.,* vol. 15, pp. 97-133, 2005.

[137] 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, pp. 402-411, 2005.