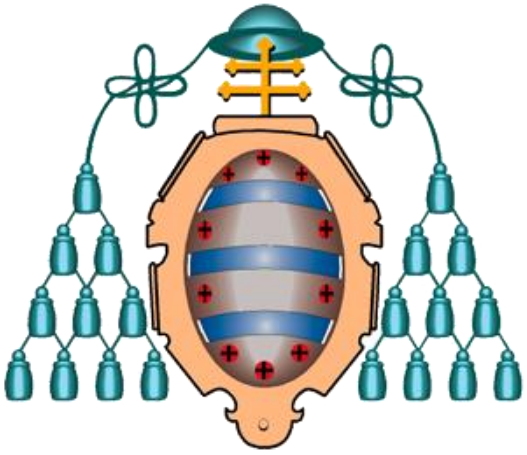


UNIVERSIDAD DE OVIEDO

Departamento de Informática

Sistemas y Servicios Informáticos para Internet

Testing Service Level Agreements in Service-based Applications

Pruebas de Acuerdos de Nivel de Servicio en Aplicaciones Basadas en Servicios

PhD Dissertation

Marcos Palacios Gutiérrez

Mayo, 2014

Don Pablo Javier Tuya González, profesor y catedrático del Área de Lenguajes y Sistemas Informáticos de la Universidad de Oviedo,

HACE CONSTAR QUE

Don Marcos Palacios Gutiérrez, Ingeniero en Informática por la Universidad de Oviedo, ha realizado bajo mi supervisión el trabajo de investigación titulado:

Testing Service Level Agreements in Service-based Applications

Una vez revisado, autorizo el comienzo de los trámites para su presentación como Tesis Doctoral al tribunal que ha de juzgarlo.

Fdo. Dr. Pablo Javier Tuya González
Área de Lenguajes y Sistemas Informáticos
Universidad de Oviedo
Gijón, Mayo 2014

Supervisor

Dr. Pablo Javier Tuya González, Universidad de Oviedo, Spain.

Internal Adviser

Dr. José García Fanjul, Universidad de Oviedo, Spain.

External Adviser

Dr. George Spanoudakis, City University London, UK.

External Examiners

Dr. Ernesto Damiani, University of Milan, Italy.

Dr. Massimiliano Di Penta, University of Sannio, Italy.

Acknowledgments

Here it comes, the last step of writing the thesis. Actually, I have expected this moment for a long time. The pleasant moment in which I really realize that the work is done and this period of my life is getting closer to the end. Now I could spend some minutes thinking and thanking all those people I care about, those that have somehow been part of this thesis and to whom I owe my sincerest gratitude. I hope not to forget anyone.

First and foremost I will ever be heartily thankful to my supervisor Javier Tuya. Thanks Javier for your guidance, your advices and for bringing light when I could only see darkness. Thanks for your patience and your essential reviews.

In the same way, thanks Fanjul for being adviser as well as friend. Thanks for having encouraged and supported me in the difficult situations we had to deal with. Thanks for enjoying (and always paying) the meals with me after each achieved goal and after each successful publication. I do not have to say that this thesis would not have been possible without you and Javier.

I would also thank George Spanoudakis for having warmly welcomed me in his office at City University London. Thanks George for giving me the opportunity to work with you and improve my research. Of course, I would like to express my gratitude to all the people I met in London. Thanks Luca for being my guardian angel there and enjoying that unforgettable stay in Hawaii. Thanks Rafa, Icamaan, Ricardo, María, Manoel, Kalsey and the rest of friends for making me feel like at home there. I will ever miss you!

During this research period I have been surrounded by great people that have definitely helped and supported me. My special thanks to my mates in the office (Dae, Rubén, María as well as those that, in any moment, worked in a nearby desk) and my colleagues of the Software Engineering Research Group at the University of Oviedo. Thanks Claudio for the hundreds of coffees you offered.

Finally and more importantly, I must thank my family and friends for being always wherever and whenever I needed them. Thanks mum, dad and brother for loving me as I am, with my virtues and defects. Thanks Andrea for always encouraging me and for all the time we have shared together. Thanks Mino, Ces, Isa and Noe for supporting me and always being willing to help me. My last thanks to all the people that asked me at some time *"How is your thesis going?"* even when they do not probably understand what I was working about.

Thank you so much!

Agradecimientos

Llega el último paso de la escritura de la tesis. He estado esperando este momento desde hace mucho tiempo. Ese agradable momento en el que de verdad soy consciente que el trabajo se ha acabado y que este período de mi vida toca a su fin. Ahora es cuando puedo dedicar unos minutos a pensar en todas aquellas personas que me importan, aquellas que de una forma u otra han sido parte de esta tesis y, por tanto, aquellas a las que debo mi más sincero agradecimiento. Espero no olvidarme de nadie.

En primer lugar y especialmente siempre estaré agradecido de corazón a mi director Javier Tuya. Gracias Javier por tu dirección, tus consejos y por enseñarme el camino hacia la luz cuando yo sólo podía ver oscuridad. Gracias por tu paciencia y tus innumerables revisiones.

De la misma forma, gracias Fanjul por ser mi tutor y mi consejero además de mi amigo. Gracias por haberme animado y apoyado en las situaciones difíciles que hemos tenido que afrontar. Gracias por haber compartido (y siempre pagado) las comidas tras cada reto cumplido y tras cada artículo brillantemente publicado. No tengo ni que decir que sin ti y sin Javier esta tesis no hubiese sido posible.

Me gustaría también dar las gracias a George Spanoudakis por su cálida acogida en su oficina en la City University London. Gracias George por darme la oportunidad de trabajar contigo y mejorar mi investigación. Por supuesto, también me gustaría dar las gracias a toda la gente que conocí en Londres. Gracias Luca por si mi ángel de la guarda y por compartir aquel inolvidable viaje a Hawaii. Gracias Rafa, Icamaan, Ricardo, María, Manoel, Kalsey y demás amigos por hacerme sentir como en casa. ¡Siempre os echaré de menos!

Durante todo este tiempo he estado rodeado de gente increíble que me han ayudado y apoyado. Mi especial agradecimiento a los compañeros de laboratorio (Dae, Rubén, María así como también a todo aquel que, en algún momento, estuvo trabajando en el ordenador de al lado) y a mis colegas del Grupo de Investigación en Ingeniería del

Software de la Universidad de Oviedo. Gracias Claudio por los cientos de café que siempre ofreciste.

Finalmente y con mayor importancia, tengo que agradecer a mi familia y amigos el haber estado siempre donde y cuando los necesité. Gracias a mi madre, mi padre y mi hermano por quererme como soy, con mis virtudes y mis defectos. Gracias Andrea por apoyarme siempre y por todo este tiempo juntos. Gracias Mino, Ces, Isa y Noe por apoyarme y estar siempre dispuestos a ayudarme. Mi último agradecimiento a toda aquella gente que en algún momento me preguntó *"¿Cómo va la tesis?"*, incluso cuando probablemente ni siquiera entendían en lo que yo estaba trabajando.

¡Muchas gracias a todos!

Abstract

In the scope of Service Based Applications (SBAs), a Service Level Agreement (SLA) is a contract or document that contains the conditions that must be fulfilled during the provision and consumption of the services. Our research hypothesis states that a cost-effective set of test cases can be systematically obtained by means of analyzing the information contained in the SLA, which is specified using the WS-Agreement standard language. The execution of such test cases would contribute to prevent, minimize or mitigate the negative consequences derived from the violations of the SLA.

In this PhD we address the testing of SLA-aware Service Based Applications by means of devising SLATF (SLA Testing Framework). SLATF allows identifying a set of test requirements from the information contained in the SLA and, later, deriving the test suites to exercise such requirements. In SLATF we have also proposed a four-valued logic in order to unequivocally evaluate the SLA and its guarantee terms.

The identification of test requirements and derivation of test cases in SLATF are performed within two different approaches. On the one hand, we consider the individual guarantee terms of the SLA in order to design the tests. In this case, we use combinatorial testing techniques to combine the test requirements and obtain a reduced number of test cases. On the other hand, we also take the logical relationships between the guarantee terms into account. In this second case, we devise a coverage based criterion in order to derive the test cases.

Both testing approaches presented in this dissertation have been automated in the SLACT (SLA Combinatorial Testing) tool and evaluated in an eHealth service-based scenario.

Resumen

En el ámbito de las Aplicaciones Basadas en Servicios (SBAs – Service Based Applications), un Acuerdo de Nivel de Servicio (SLA – Service Level Agreement) es un contrato o documento que contiene las condiciones a cumplir por los servicios constituyentes de la aplicación. Nuestra hipótesis de investigación indica que un conjunto de casos de prueba puede obtenerse de forma sistemática mediante el análisis de la información contenida en el SLA, el cual va a estar especificado utilizando el lenguaje estándar WS-Agreement. La ejecución de dichos casos de prueba contribuiría a evitar, minimizar o mitigar las consecuencias derivadas de las violaciones del SLA.

En esta tesis doctoral contribuimos a la prueba de las aplicaciones basadas en servicios que tienen asociadas un SLA mediante la elaboración de SLATF (SLA Test Framework). SLATF permite identificar un conjunto de requisitos de prueba a partir de la información contenida en el SLA para, posteriormente, derivar los conjuntos de casos de prueba que ejercitan dichos requisitos. En SLATF también se propone un lógica cuatri-valuada para evaluar de una forma inequívoca el SLA y sus términos de garantía.

La identificación de los requisitos de prueba y la derivación de los casos de prueba en SLATF se lleva a cabo usando dos enfoques diferentes. Por una parte se tiene en cuenta cada término de garantía del SLA para diseñar las pruebas. En este caso usamos técnicas estándar de pruebas combinatorias con el fin de combinar los requisitos de prueba y obtener un número reducido de casos de prueba. Por otra parte se tienen en cuenta las relaciones lógicas entre los términos del SLA. Para ello definimos un criterio de pruebas basado en cobertura para derivar los casos de prueba.

Ambas propuestas han sido automatizadas en la herramienta SLACT (SLA Combinatorial Testing) y evaluadas en un escenario de tele-asistencia médica basado en servicios.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

Claude Bernard, 1813-1878
French physiologist

This chapter introduces the scope of the research work and highlights the research hypothesis and objectives. It also presents the outcomes of this dissertation, including publications, research visits and developed tools. After that, it outlines the mandatory requirements for obtaining the qualification of International Doctor and their fulfillment. Finally, it summarizes the structure of the thesis.

1.1 Research context

The scope of this dissertation is the testing of software applications developed under the Service Oriented Architecture (SOA) paradigm. Over the last years SOA has been used to develop distributed applications by integrating available services over the web. Such services are autonomous and platform-independent entities that can be described, published, discovered and dynamically assembled for developing rapid, low-cost, interoperable and evolvable distributed applications [109].

In the field of Service Based Applications (SBAs), Service Level Agreements (SLAs) are contracts or technical documents that contain the conditions that must be fulfilled by the involved stakeholders during the provision and consumption of the services. These agreements act as a guarantee where the set of terms that govern the executions of the constituent services of the SBA are specified. They also state the penalties to be applied upon the violation of such terms. It is therefore important for both stakeholders to avoid or mitigate the consequences derived from SLA violations.

The SLAs are one of the cornerstones of this dissertation. The other one is software testing, which is the process of evaluating a program with the intent of finding faults [88]. Testing plays a key role in the development of software to evaluate whether the program meets its requirements, both functional and non-functional, and to gain confidence about the absence of bugs. Actually, it is acknowledged that software testing may consume up to a 50% of the total development time and budget of a software project [7][88]. Unfortunately, as stated by Edsger Wybe Dijkstra (Turing Award in 1972), "*Software testing can be used to show the presence of bugs, but never to show their absence*". This means that, due to the complexity of the software, it is impractical, often impossible, to detect all the faults of a program [7][88][32].

1.2 Research hypothesis

This dissertation is aligned with the following research hypothesis:

In the scope of Service Based Applications (SBAs), a Service Level Agreement (SLA) is a contract or technical document that contains the conditions that must be fulfilled during the provision and consumption of the services. We claim that a cost-

effective set of test cases can be systematically obtained by means of analyzing the information specified in the SLA. The execution of such test cases would contribute to prevent or mitigate the negative consequences derived from the violations of the SLA.

1.3 Research aims and objectives

The main objective of this PhD is the definition of a framework to test Service Based Applications where the execution conditions of the services are specified in a Service Level Agreement. Within this framework, different techniques and tools should be devised in order to identify the situations that are of interest to be tested as well as guidelines to suitably generate the test cases. The proposed framework shall provide the tester a set of systematic steps to define the tests.

The research objectives of this PhD are stated as follows:

1. To devise the main activities, inputs and outputs that the testing framework is composed of.

2. To define a concise way to evaluate the SLA and its internal elements.

3. Grounded in this evaluation, to identify a set of test requirements from the specification of the SLA.

4. To assure, as far as possible, that the identified test requirements and test cases are feasible concerning the SLA specification and the behaviour of the SBA.

5. To propose guidelines to derive a set of test cases by combining the previously identified test requirements.

6. To automate, as much as possible, the tasks involved in the generation of tests from the SLA.

7. To evaluate the proposed approach in realistic scenarios.

1.4 Research outcomes

In this section the contributions as well as the research outcomes of this PhD are outlined.

1.4.1 Contributions

The contributions of this dissertation are summarized as follows:

- Definition of a framework that allows testing SLA-aware service based applications. This framework focuses on proactive software testing techniques in the sense that a set of test cases can be derived in order to exercise different test requirements identified from the specification of the SLA. This framework is based in a four-valued logic we have also devised to evaluate an SLA and its internal elements. Specifically, our work will be focused on SLAs specified using the WS-Agreement standard language [3].

- Design, development and automation of a criterion that allows testing the SBA using the specification of the individual SLA Guarantee Terms as the test basis. From this information, a set of test requirements are identified using the aforementioned logic and standard combinatorial testing techniques are applied in order to combine such requirements and generate the test cases. In addition to this, specific rules are identified in order to avoid the obtaining of non-feasible combinations of test requirements. The whole approach has been automated in SLACT (SLA Combinatorial Testing) tool.

- Design, development and automation of a coverage-based criterion that allows generating a reduced set of tests from the logical relationships of the SLA guarantee terms. Our criterion is based on the MCDC criterion and defines specific rules in order to avoid the obtaining of non-feasible tests. We have also developed a prototype for the automatic generation of the new test requirements concerning the hierarchical structure of the SLA.

- Validation of both testing criteria. An eHealth scenario, proposed in the context of a FP7 European Project, has been used as case study in order to evaluate the feasibility of the work developed during this PhD.

1.4.2 Publications ordered by year, type and objective

A summary of the publications derived from this dissertation is presented in Figure 1.1. It classifies our contributions according to the year of publication (vertically) and the main objective of each contribution. A summary of the publications for year is presented at the bottom of the column. Furthermore, we have used different geometric shapes to represent the type of publication. A star represents a publication in a journal indexed in the Journal Citations Report® [117], a circle represents a publication in an international conference ranked in the ERA Conference Ranking Exercise (CORE) [41] and Microsoft Academic Research ranking [81]. A square represents a publication in a national conference or workshop. Finally, a hexagon represents a publication in other type of journals.

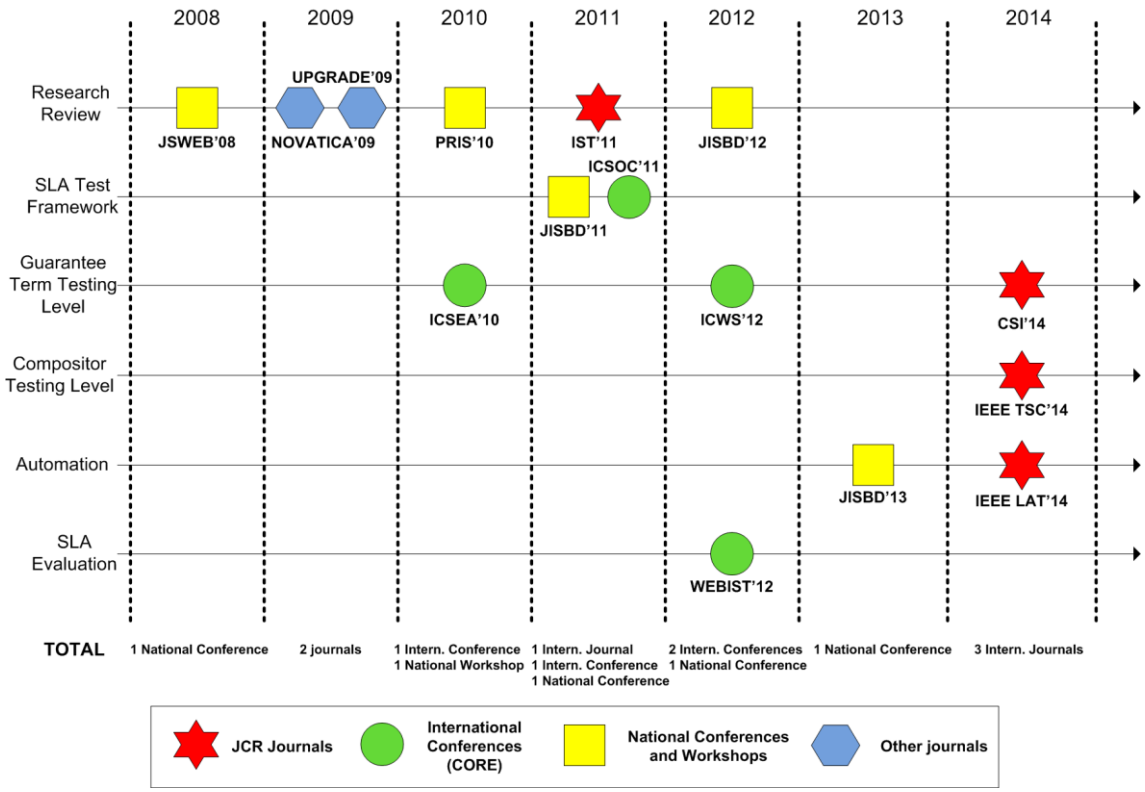


Figure 1.1: Summary of publications grouped by topic and year.

Table 1.1 presents another view of our publications classified according to the place of publication.

Category	Publications	Acronyms
JCR Journals	4	IST'11 IEEE TSC'14 IEEE LAT'14 CSI'13 (Submitted)
International Conferences (in CORE ranking)	4	ICSEA'09, ICSOC'11 ICWS'12, WEBIST'12
National Conferences	3	JSWEB'08, JISBD'11, JISBD'12
National Workshops	1	PRIS'10
Other journals	2	NOVATICA'09, UPGRADE'09
Tool Demonstrations	1	JISBD'13
Total	**15**	

Table 1.1: Summary of publications grouped by place of publication.

1.4.3 Publications in chronological order

A complete list of the publications derived from research work is presented below in chronological order.

[2008]. During the first year, we performed an initial study about the research aligned with the testing of web service compositions, focusing on those compositions specified using the BPEL standard language. The results derived from this study were published in a national conference in the field of web services and SOA [95].

JSWEB'08

M. Palacios, J. García-Fanjul, J. Tuya, C. de la Riva. **Estado del arte en la investigación de métodos y herramientas de pruebas para procesos de negocio BPEL**. *IV Jornadas Científico-Técnicas en Servicios Web y SOA (JSWEB-08)*, pág. 132–137, Sevilla (Spain), 2008.

[2009]. During this year, we completed the state of the art initiated the year [48][47] before and we started to focus our attention on a specific characteristic of the SOA paradigm such as the dynamic binding of the services.

NOVATICA'09

J. García-Fanjul, M. Palacios, J. Tuya, C. de la Riva. **Pruebas de composiciones de servicios web**. *Revista Novática*, nº 200, pp. 61-64, July-August 2009.

UPGRADE'09

J. García-Fanjul, M. Palacios, J. Tuya, C. de la Riva. **Methods for Testing Web Service Compositions. Methods for Testing Web Service Compositions**. *UPGRADE, The European Journal for the Informatics Professional*, 10 (5), pp. 62-66, 2009.

[2010]. In this year, we performed a mapping study following the guidelines proposed by Kitchenham et al. [67][68][69][20] about the works that address the testing of SOA where the services are discovered, selected and invoked at runtime. The research protocol we developed to perform the mapping study was published in a national workshop [97]. The initial review of the found studies gave us hints about the testing of Service Level Agreements as a promising research topic so we carried out a preliminary research that was published in an international conference [96]. In this work we use the Category Partition Method (CPM) [94], which has previously been used in other fields of software testing [11][34].

ICSEA'10

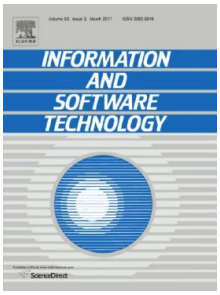
M. Palacios, J. García-Fanjul, J. Tuya, C. de la Riva. **A Proactive Approach to Test Service Level Agreements**. *5th International Conference on Software Engineering Advances (ICSEA 2010)*, pp. 453-458, Nice (France), 2010.

- Conference indexed in the CORE Ranking.

PRIS'10

M. Palacios, J. García-Fanjul, J. Tuya. **Protocolo para la revisión sistemática de estudios sobre pruebas en SOAs con enlace dinámico**. *5th Taller sobre Pruebas en Ingeniería del Software (PRIS 2010)*, Valencia (Spain), Septiembre 2010, Actas de los Talleres de las *Jornadas de Ingeniería del Software y Bases de Datos*, Vol. 4, No. 5, pp. 17-24, 2010.

[2011]. During this year we published the results derived from the mapping study performed the year before [98][103]. At that moment we realized that we would guide our research work to the SLA-based testing of Service Based Applications. Hence, we started to devise a test method that combines the advantages of both proactive and reactive approaches. The design of the method and the definition of a test strategy for SLAs were published in an international conference [99] and a national conference, respectively [100].



IST'11

M. Palacios, J. García-Fanjul, J. Tuya. **Testing in Service Oriented Architectures with Dynamic Binding: a Mapping Study**. *Information and Software Technology*, vol 53 (3), March 2011.

- Impact Factor: 1.829
- In Computer Science - Software Engineering category: position 19 of 93 (first quartile – Q1).

ICSOC'11

M. Palacios. **Defining an SLA-aware Method to Test Service Oriented Systems**. *9th International Conference on Service-Oriented Computing*, PhD Symposium, Paphos (Cyprus), 2011. In G. Pallis et al. (Eds.): ICSOC 2011, LNCS 7221, Springer 2012, pp. 164–170, 2012.

- Conference indexed in the first level (A) of CORE Ranking.

JISBD'11

M. Palacios, J. García-Fanjul, J. Tuya. **Definición de una Estrategia de Pruebas basada en Acuerdos de Nivel de Servicio**. *XVI Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2011)*, pp. 519-524, La Coruña (Spain), 2011.

[2012]. In this year, we focused on the definition of a test criterion that allows identifying a set of test requirements from the information contained in the SLA. The

first results of this work were published in one of the most important international conferences in the scope of web services [101]. Furthermore, we collaborated with other researchers to study the evaluation of SLAs in the context of the NDT (Navigational Development Techniques) methodology [42][90]. This study was also published in an international conference [102].

ICWS'12

M. Palacios, J. García-Fanjul, J. Tuya, G. Spanoudakis. **Identifying Test Requirements by Analyzing SLA Guarantee Terms**. *19th International Conference on Web Services (ICWS)*, pp. 351-358, Honolulu, Hawaii (USA), 2012.

- Conference indexed in the first level (A) of CORE Ranking.

WEBIST'12

M. Palacios, L. Moreno, M.J. Escalona, M. Ruiz. **Evaluating the service level agreements of NDT under WS-Agreement. An empirical analysis**. *8th International Conference on Web Information Systems and Technologies (WEBIST 2012)*, pp. 246-250, Porto (Portugal), 2012.

- Conference indexed in the CORE Ranking.

JISBD'12

M. Palacios, J. García-Fanjul, J. Tuya. **Testing in Service Oriented Architectures with Dynamic Binding: A Mapping Study**. *12th Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2012)*, pp. 383-384, Almería (Spain), 2012 (Recently Published Articles Track).

[2013]. During this year, we refined and concluded the work we had developed the years before. First of all, we elaborated an article that contains all the details about the criterion that allows deriving test cases from the information specified in the SLA by means of applying combinatorial testing techniques. This article was submitted to the Computer Standards & Interfaces journal [106]. After that, we elaborated another article

that presents a coverage-based criterion that allows identifying test requirements from the logical combination of SLA guarantee terms. This article was submitted and accepted in the IEEE Transactions on Service Computing journal [105] and it was published in 2014. Finally, we presented the details about the development of SLACT tool in an article that was published in the IEEE Latin America Transactions journal [107]. A demonstration about the aforementioned tool was also showed in a national conference [104].

IEEE TSC'13

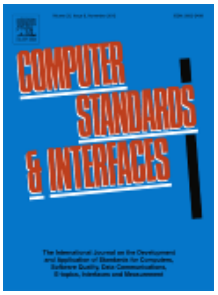
M. Palacios, J. García-Fanjul, J. Tuya, G. Spanoudakis. **Coverage-based Testing for Service Level Agreements**. *IEEE Transactions on Services Computing*. 28 Feb. 2014. IEEE computer Society Digital Library.

- Impact Factor: 2.460
- In Computer Science - Software Engineering category: position 5 of 105 (first quartile – Q1).
- In Computer Science – Information Systems category: position 13 of 132 (first quartile – Q1).

IEEE LAT'13

M. Palacios, J. García-Fanjul, J. Tuya. **Design and Implementation of a Tool to Test Service Level Agreements**. *IEEE Latin America Transactions*. Vol. 12, Issue 2, March 2014, pp. 256-261.

- Impact Factor: 0.218
- In Computer Science - Information Systems category: position 129 of 132 (fourth quartile – Q4).
- In Engineering – Electrical & Electronic category: position 227 of 242 (fourth quartile – Q4).



CSI'13

M. Palacios, J. García-Fanjul, J. Tuya, G. Spanoudakis. **Automatic test case generation for WS-Agreements using combinatorial testing**. *Computer Standards & Interfaces*. (Submitted in 2013).

- Impact Factor: 0.978
- In Computer Science - Software Engineering category: position 55 of 105 (third quartile – Q3).
- In Computer Science – Hardware and Architecture category: position 26 of 50 (third quartile – Q3).

JISBD'13

M. Palacios, P. Robles, J. García-Fanjul, J. Tuya. **SLACT: a Test Case Generation Tool for Service Level Agreements**. *13th Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2013)*, Madrid, 2013 (Tool Demonstration Track).

1.4.4 Research visits

During his academic stage, the PhD student has visited the Department of Computing of the City University London in the United Kingdom (5 months). In London he has collaborated with the members of the Software Engineering at City Research Group (SE@City), founded and leaded by Dr. George Spanoudakis. Since these visits he has closely worked with Dr. Spanoudakis, who is one of the most important international researchers in the service oriented computing research area, specially focused on topics related to Service Level Agreements. The information about his visits at City is represented in Table 1.2.

Year	Arrival Date	Departure Date	Goal
2011	11/11/2011	31/12/2011	Definition of a test method by means of analyzing the information contained in the Guarantee Terms of an SLA.
2012	01/05/2012	31/07/2012	Improvement of the identification of test requirements taking the logical relationship between the Guarantee Terms into account.

Table 1.2: Information about the research visits.

1.4.5 Developed tools

The results of this dissertation have been integrated into a tool named SLACT (SLA Combinatorial Testing) [123], developed in the context of our research group and registered in the Spanish Intellectual Property Office (Nº 05/2013/458). Basically, SLACT allows analyzing the specification of the SLA with the aim at identifying test requirements that will be combined in order to generate a set of test cases. A detailed description of SLACT is presented in Chapter 7 of this dissertation.

1.5 International thesis

With the elaboration, submission and defence of this dissertation we expect to obtain not only the degree of Doctor of Philosophy (PhD) but the International Mention of such degree. In Spain, the Royal Decree 99/2911 [16] is the official document that regulates the requirements to be fulfilled in order to achieve the international mention in the PhD degree, which are the following:

1. During the academic stage in the PhD programme, the student has completed a minimum stay of three months outside Spain in a Higher Education institution or a prestigious research centre in another foreign country, studying or doing research. This stay must be acknowledged by the thesis director and must also be certified by the PhD Programme.

2. Part of the PhD dissertation, at least the abstract and conclusions, has been written and presented in one of the official languages, excluding Spanish, commonly used for scientific communications in the field of knowledge.

3. The PhD dissertation has been informed by a minimum of two experts from a Higher Education institutions or research institution from a foreign country other than Spain.

4. At least one expert from a Higher Education institution or research institution from a foreign country, other than Spain, with a degree of Doctor and different from the supervisor of the stay mentioned in the first requirement has been part of the PhD Dissertation Committee.

All of the above requirements have been met during the academic stage of this PhD. The first requirement is satisfied thanks to the two certified stays as a visiting researcher at City University London, supervised by Dr. George Spanoudakis (see Section 1.4.4: Research Visits). This dissertation has been completely written in English, which is the most widely accepted language in the field of Computer Science, so the second requirement is also satisfied. Furthermore, both the Abstract and Conclusions sections are written in Spanish as well. With the aim at satisfying the third requirement, this dissertation has been sent to Dr. Ernesto Damiani and Dr. Massimiliano Di Penta in order to get their evaluation reports. Finally, Dr. Christos Kloukinas has accepted our invitation to be part of this PhD Dissertation Committee so the last requirement will also be met.

1.6 Structure of this dissertation

The reminder of this document is structured as follows:

Chapter 1 comprises this introduction section.

Chapter 2 highlights the basic concepts used within this dissertation and summarizes the state of the art in the addressed research topic.

Chapter 3 presents the SLA Testing Framework (SLATF), including its main activities, inputs and outputs.

Chapter 4 proposes a concise way to unequivocally evaluate Service Level Agreements, including their guarantee terms and compositors.

Chapter 5 addresses the generation of tests taking the information contained in the individual SLA guarantee terms into account. To do this, we devise a criterion that makes use of standard combinatorial testing techniques.

Chapter 6 also addresses the generation of tests but, in this case, we consider the information contained in the logical combinations of guarantee terms. To do this, we devise a criterion that is based on coverage-based testing.

Chapter 7 describes the details about the level of automation of the two aforementioned criteria.

Chapter 8 presents the evaluation of the proposed approach in an e-Health based case study provided in the context of a European Project.

Chapter 9 and Chapter 10 states the conclusions of the research performed during this PhD and outlines potential research lines for our future work (in English and Spanish respectively).

1.7 The picture of this dissertation

The last section of the chapter summarizes the main concepts of this dissertation in form of a word cloud (Figure 1.2), being agreed with the classical adage that says "A picture is worth a thousand words".

Figure 1.2: Word Cloud of this dissertation.

Chapter 2

Background

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger Wybe Dijskstra, 1930-2002
Dutch computer scientist, 1972 Turing Award

This chapter outlines the main concepts of this dissertation. Firstly, it provides important definitions about software testing, one of the cornerstones of this thesis. After that, it presents the Service Oriented Architecture (SOA) software development paradigm and the important role of Service Level Agreements (SLAs) to regulate the distribution of the services. Finally, it describes the related work in the field of SOA testing and SLA-based testing.

2.1 Introduction

In this chapter we present the main concepts that represent the cornerstones of this dissertation. Firstly, we introduce the software testing by means of providing standard definitions that will help to understand the main core of our approach. After that, we explain the importance of the Service Oriented Architectures paradigm within the development of a great number of applications nowadays. Furthermore, we outline the most relevant properties of Service Level Agreements and, to be more specific, the WS-Agreement standard language. Finally, we analyze other works that have previously been proposed in the scope of the topic addressed in this dissertation.

2.2 Software Testing

Several definitions have been proposed in the literature when referring to software testing. According to the ISO/IEC 24765 (Software and Systems Engineering Vocabulary):

"Software Testing can be defined as an activity in which a system is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system" [59].

This definition requires the Software Under Test (SUT) to be executed. Other authors distinguish between static and dynamic testing [9]. In this case, the static testing is carried out without executing the SUT whereas the dynamic testing always involves the execution of the SUT. In the context of this dissertation, the SUT is any service-based application, typically a web service composition, in which the execution conditions of the constituent services are specified in an SLA.

In the recently published ISO/IEC/IEEE 29119 Software and Systems Engineering - Software Testing:

"Software Testing is defined as a set of activities conducted to facilitate the discovery and / or evaluation of properties of one or more test items" [60].

Another typical definition of testing says that:

"Software Testing involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results" [54].

In addition to this, one of the classical and simpler definitions outlines the main objective of testing as:

"Software testing is the process of evaluating a program with the intent of finding faults" [88].

The detection of problems in the SUT is usually addressed with two main different approaches. On the one hand, the execution of the SUT is typically performed through the design and execution of test cases. According to the IEEE Standard Glossary of Software Engineering Technology, a test case is "*a set of inputs, execution conditions, and expected results developed for a particular objective*" [58]. Thus, executing the software and comparing the obtained outputs with the expected results allows determining whether the behaviour of the software is correct or not. The generation and execution of test cases is considered a *proactive* or *ex ante* approach in the sense that it is able to detect problems in the SUT before such problems occur in an operational environment and lead to undesired consequences for the stakeholders.

On the other hand, monitoring is also a widely used testing technique that allows passively observing real time executions with the aim of detecting any deviation from the expected behavior of the software during its operation [35]. Monitoring based approaches are considered *reactive* because problems are detected *ex post*, after they have occurred and, thus, potential further consequences cannot always be avoided. The concept of monitoring is named as *on-line testing* by Bertolino in [14].

Concerning these two main testing approaches, *a test requirement represents a specific feature or situation of the SUT that must be satisfied or covered during testing* [92]. Test requirements are typically identified following a specific test strategy, which might be based on different factors such as risks, models of the system, expert advice or heuristics. In this context, the test basis represents all documents from which the

requirements of a system can be inferred or the documentation on which test cases are based [61].

In this dissertation, we tailor different standard testing techniques in order to identify the test requirements and, later, combine such test requirements with the aim at deriving the test cases. Below, we briefly describe the main characteristics of these testing techniques.

The *Classification Tree Method* [53] provides a systematic way to hierarchically partition the inputs of a SUT into classifications and classes via the construction of an appropriate classification tree. Each classification is a disjoint partition related to the SUT and each class is a disjoint partition of the values of the corresponding classifications. From the constructed tree, test coverage items shall be derived by combining leaf nodes using combinatorial techniques. In this context, a test coverage item represents an attribute or combination of attributes regarding the SUT that will be exercised by a test case.

Combinatorial testing techniques [52][91] are used to generate test cases that achieve different levels of coverage. The combinations are defined in terms of parameters and the values that they can take. To align this with the constructed classification tree, classifications represent parameters and classes represent parameter values. There are different combinatorial testing techniques such as *Pair-wise* (broadly use in software testing [78][89]), *All combinations* or *Each choice* that will be later used in this thesis.

Modified Condition Decision Coverage (MCDC), defined in the RTCA/DO-178B standard [116], is a broadly studied structural coverage criterion. MCDC is applied to a specification of the SUT, which may be the code of the program itself. MCDC allows identifying test requirements taking the specification of the SUT into account and, besides, it provides a linear increase in the number of test requirements [38].

2.3 Service Oriented Architectures

In this section we present the basic concepts about the paradigm of Service Oriented Architectures (SOAs) and we highlight the main characteristics of Service Level Agreements (SLAs), which are one of the key issues of this thesis.

Service Oriented Architectures have become a successful paradigm to develop distributed applications by integrating available services over the web. Such services are autonomous and platform-independent entities that can be described, published, discovered and dynamically assembled for developing rapid, low-cost, interoperable and evolvable distributed applications [109]. Web services are the most used SOA based technology and they are supported with a set of W3C XML based standards: Simple Object Access Protocol (SOAP) [124], Web Service Description Language (WSDL) [139] and Universal Description, Discovery and Integration (UDDI) [132].

Service Oriented Architectures allow the interaction between service providers and clients. In a Service-Based Application, the provider publishes the description of the services, generally specified in WSDL, in a registry. This registry can be implemented using the UDDI standard and it is in charge of storing service descriptions and acts as an intermediary between providers and clients. After the services are published, a client sends a query to the registry to find the desired service. The registry matches the client's request with the available information and returns to the client a set of service interface descriptions that satisfy its requirements. The client has to select the most suitable service and bind with its provider performing the invocation of the service and receiving the corresponding response.

In Figure 2.1 the typical SOA triangle, adapted from [110], with the roles that each stakeholder plays and the operations that can be performed is depicted.

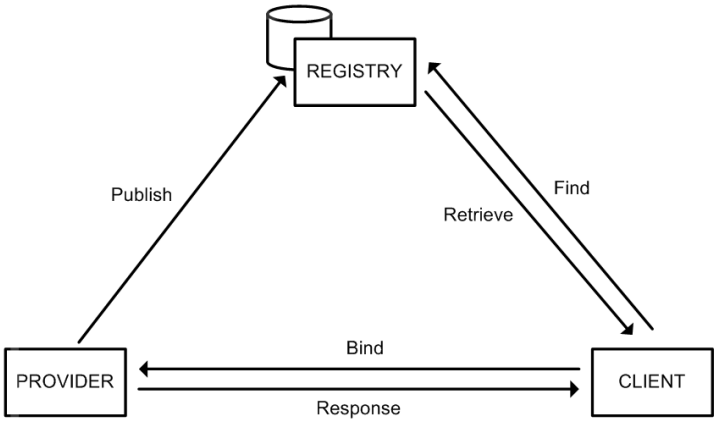


Figure 2.1: SOA Architecture: roles and operations.

A client can decide at design time which service is going to be executed so the binding is considered as static. However, a challenging feature of SOA is the possibility to select and invoke a service at runtime. There are two scenarios where this binding can be considered as dynamic. In the first scenario, a set of potential candidate services is available at design time although the client does not know exactly which one is going to be invoked until the moment of the binding. In the second, the discovery, selection and invocation can be performed at runtime using a registry. In this case, until the discovery, the client has no knowledge about the potential services that can be invoked.

2.4 Service Level Agreements

Service Level Agreements (SLAs) are contracts that specify the rules for the trading between the consumers and the Service Based Applications (SBAs) providers. Typically, these rules specify which the constituent services of the SBA that will be regulated by the agreement are, and how these services should be offered.

The management of SLAs [115] is an integral part of the applications developed under the rules of a standard SOA Governance framework [12] Bertolino, SOA Test Framework] and has recently received considerable attention both in industry and academia (see, for example, the SLA@SOI FP7 European Project [122]). Many large companies, including Amazon [1], Microsoft [83], Google [51], AT&T [4] and HP [57], that provide XaaS (Everything as a Service) use SLAs as a mechanism for specifying the functionalities and QoS levels that they are capable of providing in their XaaS

offerings. Although the existing SLAs in the industrial domain seems to be quite simple nowadays, they could become very complex by means of establishing relationships between the terms or including information regarding the functional and non-functional features of the services as well as the penalties derived from the violations of the agreed guarantees.

In addition to typical tasks involved within the management of the SLAs, including negotiation [36][112][142], evaluation [26], optimization [113][135], monitoring [77][126][136] or testing [37], the specification of the SLAs has been widely studied over the last few years [130]. In many occasions the SLAs are specified in documents without any kind of format or even using natural language. Unfortunately, this lack of methodology when creating an SLA hinders the automatic management of the agreement. In our case, the testing of the SLAs requires using such documents as the test basis so we need to have the specification of the SLA somehow formalized in order to automate as much as possible the obtaining of tests.

Over the last decade, different languages have been proposed [108] with the aim to support and standardize the specification of SLAs (e.g., WSLA [66], WS-Agreement [3], WSLO [129], SLANG [71][121], WS-QoS [128] or WS-Policy [134]). Among them, WS-Agreement is the one that has received more attention regarding the SLA-based testing, at least from the academic scope. WS-Agreement presents a generic syntax that allows extrapolating its derived outcomes to any other existing SLA specification language. In fact, WS-Policy, which is gaining attraction from the industrial space, shares the same notation as WS-Agreement to represent the relationships between the guarantees. Thus, in this work, we focus on the syntax and semantics of WS-Agreement because it is a well-accepted standard in the SOA protocol stack for the management of the SLAs (Figure 2.2) and has been used in different approaches regarding the testing of SBAs [15][77][85].

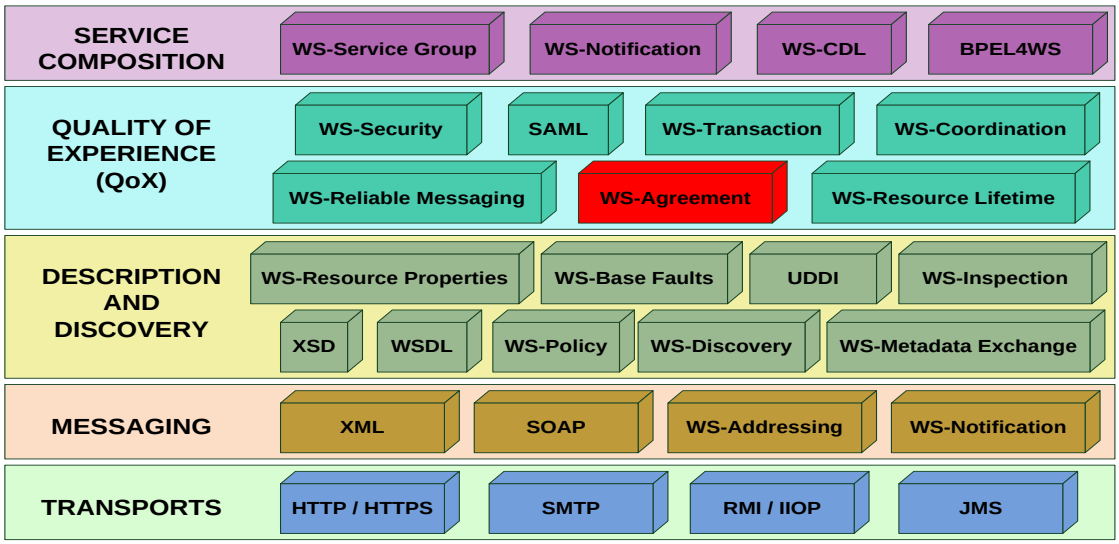


Figure 2.2: Web Service Protocol Stack (adapted from IBM Software Group [118]).

WS-Agreement

WS-Agreement (WSAG) [3] is an XML based language proposed by the Open Grid Forum (OGF) that specifies a protocol for the establishment of agreements between two parties. This standard defines a hierarchical structure for the specification of an SLA. The specification of an SLA using the WS-Agreement standard language is composed of three main parts (Figure 2.3). These are:

- **Name**: This part represents an optional name that can be given to the agreement.
- **Context**: This part describes the involved parties and their role as initiator or responder. Additionally, it may specify any other information of the agreement that is not related with the obligations of these parties, such as the "Expiration Date.
- **Terms**: This part expresses the negotiated and agreed obligations of each party. Obligations are specified using different types of terms:
    - Service Description Terms (SDT): describe information about the functional aspects of the services.

- Service Properties (SP): provide measurable aspects that are used to express the requirements (guarantees) of the services.
- Guarantee Terms (GT): describe the obligations that must be satisfied by a specific obligated party.

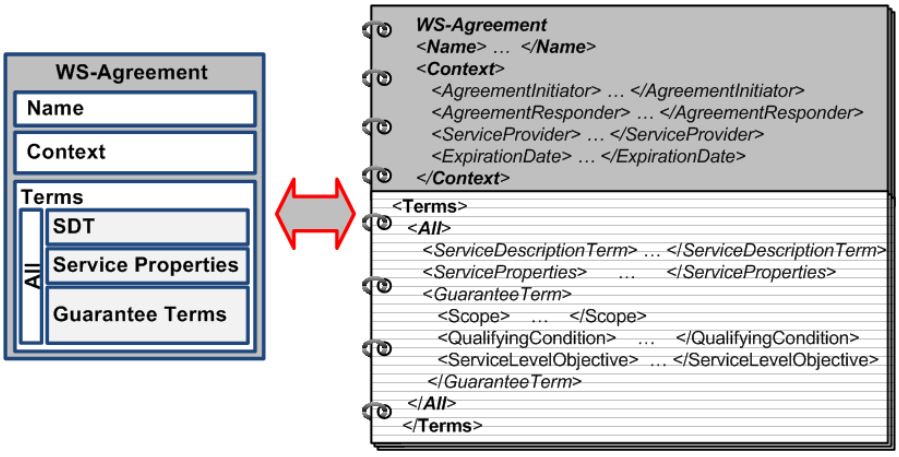


Figure 2.3: WS-Agreement structure.

The most important information of the SLA is represented by means of the Guarantee Terms, which describe the obligations that must be satisfied by a specific obligated party. A *Guarantee Term* (GT) contains the following internal elements:

1) The *Scope* specifies the list of services the term applies to.
2) The *Qualifying Condition* (QC) represents a precondition or assertion that determines whether the term is relevant and must be considered during the evaluation process.
3) The *Service Level Objective* (SLO) specifies the guarantee that must be met.
4) Optionally a *Business Value List* (BVL) for such term may also be specified containing some information as the penalties for not having satisfied the associated guarantee.

The specification of domain specific languages or extensions to express the conditions of the Guarantee Terms is out of the scope of WS-Agreement.

It is worth noting that WS-Agreement allows the logical combination of these terms by means of elements named *Compositors*. More specifically, there are three different compositors: *All*, *OneOrMore* and *ExactlyOne*, which are equivalent to the logical AND, OR and XOR operators respectively.

2.5 Related Works

In this section we briefly outline the works that have been previously proposed to address the testing of applications developed under the Service Oriented Architecture paradigm and those attempts that take the SLA as the test basis.

2.5.1 SOA Testing

As stated by Canfora and Di Penta [21], the dynamic and adaptive nature of SOA-based systems makes that most testing techniques cannot be directly applied when testing services and service-oriented systems. Some key issues that affect the testability of service-based applications include:

- *Lack of observability of the service code and structure*. The services become just interfaces for consumers and system integrators, and this hinders the application of white-box testing approaches that require having some knowledge of the structure of the code and data flow.

- *Dynamicity and Adaptiveness*. In traditional systems it is almost possible to determine the component that will be invoked or, at least, the set of possible targets [80]. This is not true anymore for SOA, where a system may be described by means of a workflow of abstract services that are automatically discovered, selected and invoked at runtime, which is called *dynamic binding*.

- *Lack of control*. While component or libraries are physically integrated in a classical software system, this is not the case for services, which are deployed in an independent infrastructure and are maintained and evolved under the control of the service provider. This implies that the system

integrators cannot decide the strategy to migrate a new version and, consequently, to perform regression testing to the system [23].

- *Cost of testing*. The invocation of the services that are deployed in the provider's infrastructure also affects the cost of testing, when services are charged on a per-use basis. Furthermore, repeated invocations of a service for testing may not be allowed if the service produces side effects other than a simple response, for example in web service compositions that implements the business process of a hotel booking [24].

These and other more issues imply that traditional testing techniques need to be adapted when testing service based systems. In the literature, many efforts have been made in the area of testing of services and service-based applications. A survey of SOA testing is presented in [22] and another survey about web service testing is presented in [17]. Furthermore, a systematic review about formal approaches to test service based software is provided by Endo and Simao in [40] and another systematic review is presented by Zakaria in [141], where they study the state of the art about unit testing web service compositions specified using the standard language WS-BPEL [138]. Finally, Palacios et al. [98] present a mapping study about testing SOA-based systems with dynamic binding.

In addition to this, many works have been proposed to address the testing of web service composition [39][45][46][55][56][74][127]. Those works focus on different aspects of the compositions such as internal behaviour, coordination, control flow execution, robustness or non-functional testing. The testing of web service composition in the integration level has also been investigated by Bucchiarone et al. in [19].

2.5.2 SLA-based Testing

In the context of service-based applications, the SLA-based testing has been identified as a challenge [23][6]. During recent years, many works have been proposed with the final objective of detecting SLA violations. Typically, related strands of work may be categorized in two main groups:

- *Testing*: the set of works which are aimed at anticipating problems and/or prevent them before deployment, when such problems would lead to undesired consequences for the stakeholders who have signed the agreement.

- *Monitoring*: the set of works that are aimed at detecting SLA violations at runtime when the SUT is already deployed in the operational environment.

These two main approaches have been discussed in the literature when testing SOA, identifying their advantages as well as their drawbacks. In Figure 2.4, Canfora and Di Penta [21] show the role of testing and monitoring, aiming at exploding the synergies between both testing techniques.

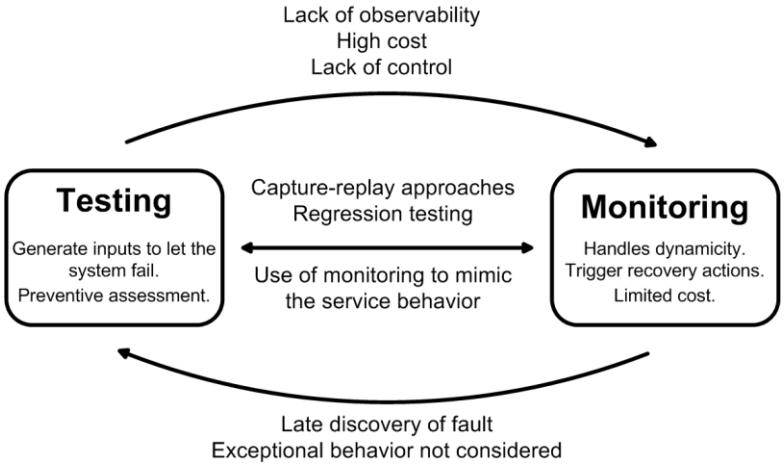


Figure 2.4: The role of testing, of monitoring and their interaction (Adapted from [21]).

In this figure, there are many weaknesses of monitoring that suggest to perform testing (simple arrows), and vice versa. Also, there many cases where testing and monitoring can be applied in order to complement each other (double arrow).

2.5.2.1 Testing

Regarding the testing of SLAs, few approaches have focused on the identification of tests from the specification of the SLAs in order to anticipate problems.

Di Penta et al. [37] perform black-box and white-box testing by means of using Genetic Algorithms with the aim of detecting SLA violations in atomic and composite

services. This approach generates combinations of inputs and bindings of the constituent services that may cause violations of the SLA.

Bertolino et al. [10] propose PUPPET framework, which generates test beds from the WSDL and BPEL specification of service compositions, considering the information contained in a WS-Agreement. Although they do not specify the tests for the SBA, they provide the necessary infrastructure to deploy and execute such tests.

Kotsokalis et al. [70] have proposed to use Binary Decision Diagrams in order to model the content of SLAs for testing purposes. However they do not focus on a specific standard language although they attempt to obtain the diagrams from WS-Agreement. In their approach, they use two different values to evaluate the terms of the SLA.

Finally, Muller et al. [86] propose static testing by automatically detecting and explaining inconsistencies between the terms of WS-Agreements using a Constraint Satisfaction Problem based approach.

2.5.2.2 Monitoring

Regarding the second group, there are more works that use monitoring techniques rather than testing to detect SLA violations.

Mahbub and Spanoudakis [77] focus on WS-Agreement to propose modelling and monitoring the conditions contained in the SLA using an Event Calculus (EC) based approach. Raimondi et al. [114] proposed a system that automatically monitors SLAs, translating timeliness constraints into timed automata, which is used to verify traces of services executions. Comuzzi et al. [31] tackles the relation between the establishment and monitoring of SLAs in the scope of SLA@SOI European Project [122]. Sahai et al. [119] propose an automated and distributed SLA monitoring engine in order to check the compliance of the SLA and, if necessary, take control actions to enable such compliance. Michlmayr et al. [87] present a framework that combines the advantages of both client- and server-side SLA monitoring. This framework builds on event processing to inform about QoS measurements and possible SLA violations. Fakhfakh et al. [43] elaborate a complete, generic and semantically rich model for an SLA based

on ontologies. They use the Semantic Web Rule Language (SWRL) to specify the SLA obligation within the model. Such model facilitates the monitoring of the SLA as well as triggering eventual corrective actions in case of SLA violations.

Beyond these works, there are other systems that have been developed to monitor whether service based applications violate SLAs including, for example, SALMon [2][93], SALMonADA [85], SLAMonitor [50], HA-SLA [84] and CLAM [18].

2.5.2.3 Other Approaches

In the borderline between these testing and monitoring, there is a set of works that make use of information gathered from monitoring techniques in order to prevent SLA violations.

Lorenzoli and Spanoudakis [75][76] present the EVEREST+ framework, which supports the monitoring and prediction of potential violations of the QoS metrics specified in an SLA. Leitner et al. [73] propose a framework that allows monitoring and predicting SLA violations before they have occurred using machine learning techniques and they have also addressed the prevention of SLA violations using self-adaption [72]. Ivanovic et al. [62] propose a constraint-based approach to monitor and analyze the QoS metrics included in the SLA for the purpose of anticipating the detection of potential SLA violations. Finally, Schmieders et al. [120] combined monitoring and prediction techniques in order to prevent SLA violations.

2.6 Summary

In this chapter we have presented the basic concepts that are necessary to understand the remaining of this dissertation.

We have introduced some notions about software testing, including general definitions of testing, test cases, test requirements as well as the description of some standard testing techniques, which are used in this thesis.

After that, we have presented the main characteristics of Service Oriented Architectures including the stakeholders that usually appear in the life cycle of a service based applications as well as the typical operations that are performed in order to

discovery, select and invoke the services. Furthermore, we have also highlighted the importance of Service Level Agreements within the provisioning and consumption of the services, focusing on the WS-Agreement standard for being the language we have used in the current work.

Finally, we have briefly described the state of the art about the existing works that address the testing in the context of SOA-based systems and we have presented those works that put their effort in detecting, preventing or predicting SLA violations.

In the following chapters, we will present our contribution within the context of this dissertation. We will start presenting SLATF (SLA Testing Framework) in Chapter 3, which allows testing SLA-aware service-based applications.

Chapter 3

SLA Testing Framework

Computers are incredibly fast, accurate, and stupid:
humans are incredibly slow, inaccurate and brilliant;
together they are powerful beyond imagination.

Albert Einstein, 1879-1955
German physicist, 1921 Nobel Prize in Physics

This chapter presents SLATF (SLA Testing Framework), a framework that allows testing SLA-aware Service-based Applications (SBAs). It introduces the main activities involved in SLATF, which will be later described in detail in the rest of chapters of this dissertation.

3.1 Introduction

As we have outlined in Chapter 2, the process of developing software under the Service-Oriented Architecture (SOA) paradigm presents specific characteristics that need to be taken into account. Basically, different providers develop the services and define the interfaces of such services, which are made public using registries, web pages or any other adequate source. The services are therefore deployed in an operational environment which may be, for example, in the own providers' infrastructures or hired to a third-party hosting service. Once the services are ready to be used, it is necessary to establish the set of conditions that will govern the provision and consumption of such services. These conditions are often agreed between the provider and the consumer and specified in a Service Level Agreement (SLA), which acts as a guarantee for the stakeholders involved in the service trading.

During the execution of the services, different problems may happen in the service based application. Sometimes these problems are related to the agreed conditions specified in the SLA, whose violations lead to the application of the corresponding penalties also stated in the SLA. Hence, a necessity arises in the sense that we need to detect such problems in the SBA as soon as possible so as the consequences derived from an SLA violation can be avoided. Typically, the detection of such problems is usually carried out by means of software testing approaches, which may be reactive or proactive as we have previously described in Section 2.2.

In the scope of this PhD we focus on the definition of a proactive approach to test SLA-aware Service Based Applications (SBAs). In this chapter we start presenting a step-wise test framework that addresses the detection of problems related to guarantee terms of the SLA by using software testing techniques.

3.2 SLA Testing Framework (SLATF)

The objective of SLATF (SLA Testing Framework) is to test a service-based application where an SLA specifies the conditions that must be checked to decide whether the execution of the services is successful. This framework involves a test process that contains four activities to deal with: (1) SLA evaluation, (2) identification

of test requirements, (3) generation of test cases, and (4) execution of test cases. The general architecture of SLATF is depicted in Figure 3.1 and the information flow of the framework, with its activities and their corresponding inputs and outputs, is represented in Figure 3.2.

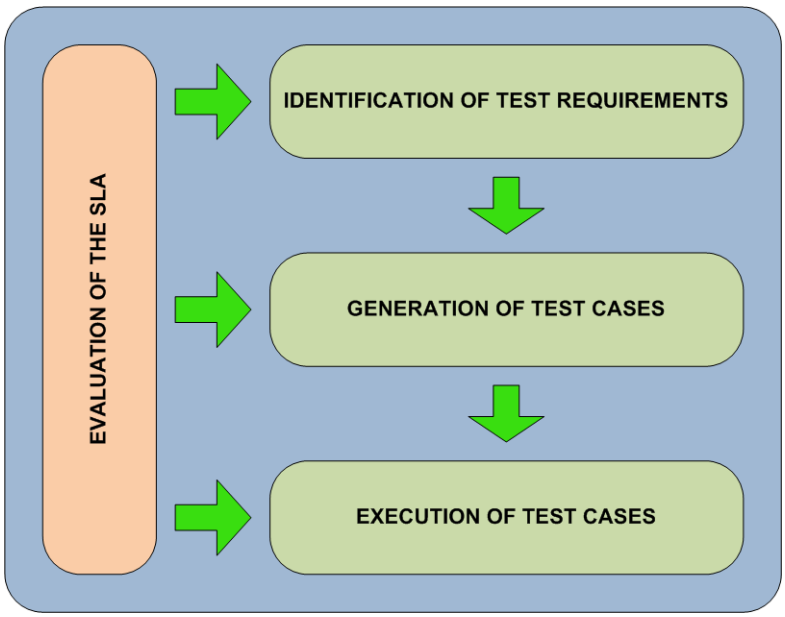


Figure 3.1: SLATF architecture.

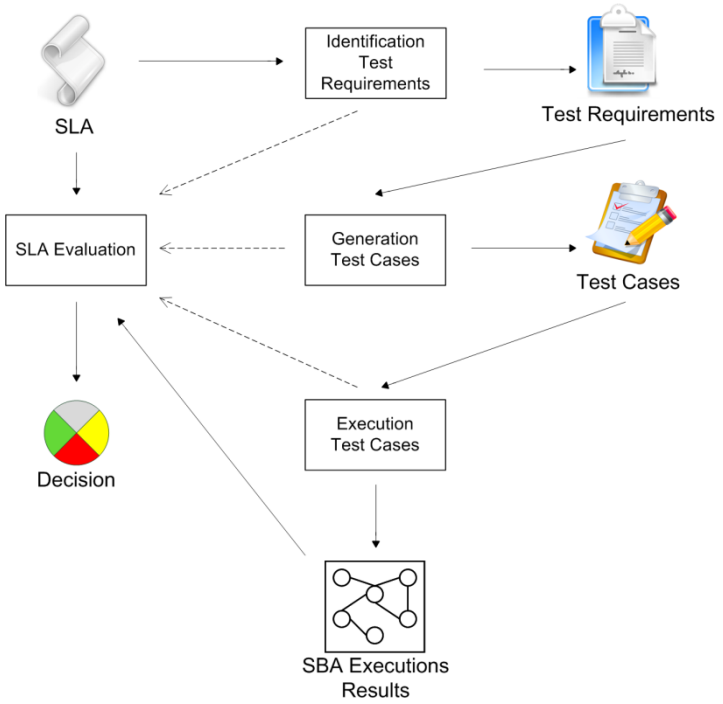


Figure 3.2: SLATF Testing Framework.

In Figure 3.2 the activities of SLATF are represented using rectangles. These activities receive different artifacts as inputs and provide their own outputs. The relation between the activities and their inputs and outputs is represented with directed arrows in the figure.

First of all, SLATF requires the definition of a concise way to determine whether the SLA has been fulfilled or violated. This evaluation of the SLA is an orthogonal activity that closely affects the rest of the testing process. This means that there is a dependency between the SLA evaluation and the other three activities involved in the process in the sense that such activities use the evaluation in order to carry out their tasks. This dependency is represented in Figure 3.2 using dashed arrows pointing from the dependent activities to the SLA evaluation, which is the independent one.

The main input of SLATF is an SLA specified in the WS-Agreement standard language [3], which may represent both functional and non-functional characteristics of the service based application. Taking the aforementioned evaluation process into account, such SLA is used as the test basis in order to identify test requirements (situations that should be covered during testing) regarding the SBA.

The next activity of SLATF is the generation of a set of test cases, which takes the previously identified test requirements as input. The set of test cases will be designed with the aim at covering all the test requirements. Each test case will cover as many test requirements as possible in order to reduce the size of the final set of test cases.

The last step of SLATF involves the execution of the test cases in the service based application. The results of the tests allow detecting problems in the SBA and determining whether the behaviour satisfies the agreed guarantees by means of evaluating the SLA. This decision is represented in Figure 3.2 using a solid line that departs from the SLA evaluation activity.

In the rest of this chapter we describe the general characteristics of the four different activities to be carried out in SLATF, aligned with the research objectives stated in Chapter 1. Both the SLA evaluation and the identification of test requirements will be fully addressed in the following chapters of this dissertation. We will also

outline how to derive test cases from the identified test requirements. Further automation of the test cases execution is expected to be addressed in our future work and, thus, is out of the scope of this dissertation.

3.2.1 SLA Evaluation in a nutshell

The evaluation of an SLA requires making a decision about whether the agreed guarantees are being respected or not during a specific execution of the SLA-aware Service Based Application. By means of observing the service executions, the evaluation allows detecting whether a deviation from the expected behaviour of the SBA according to the SLA has arisen. Disregarding the testing approach to be applied (for example, proactive or reactive as stated in Chapter 2), the SLA evaluation is a process that always needs to be performed when an SLA is associated to the SBA. This decision is typically carried out using a binary logic so the SLA can be evaluated as Fulfilled or Violated. However, the use of these two classical values makes difficult to represent all the potential situations that can arise during the evaluation of the SLA. Thus, in Chapter 4 we will introduce two more evaluation values (Not Determined and Inapplicable), which lead to a four-valued logic in order to evaluate SLAs.

Furthermore, it is worth mentioning that the evaluation of an SLA depends on the evaluation of its guarantee terms. Hence, it is necessary to define a specific way to determine the evaluation value in two different levels:

1. The evaluation of each individual guarantee term.

2. The evaluation of the logical relationships between the guarantee terms and, consequently, the complete SLA.

The complete description of the activity that determines the evaluation of the SLA is fully addressed in Chapter 4 of this dissertation.

3.2.2 Identification of test requirements

The identification of test requirements is performed by means of analyzing the SLA and taking the aforementioned four-valued logic into account. Typically, a test requirement represents a specific feature of the software that must be satisfied or

covered during testing [92]. At this stage, it is necessary to clarify the concept of test requirement within the scope of this dissertation. On the one hand, the test requirements may be identified taking the information contained in the individual guarantee terms into account. These test requirements are called Primitive Test Requirements in this dissertation with the following meaning:

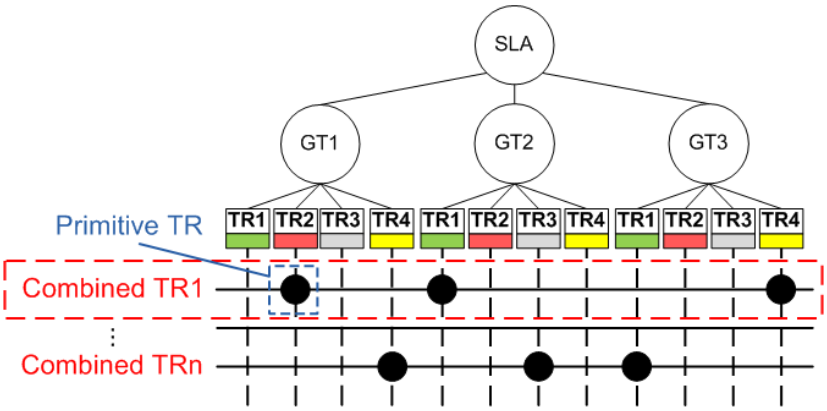
- *A Primitive Test Requirement (Primitive TR)* involves exercising a situation associated to the evaluation of a single guarantee term.

On the other hand, we have previously mentioned that the guarantee terms can be grouped according to a specific relationship, for example, using the compositor elements. Likewise, new test requirements can also be identified by means of combining the Primitive Test Requirements:

- *A Combined Test Requirement (Combined TR) involves exercising a particular combination of Primitive Test Requirements that require the evaluation of multiple guarantee terms.*

The terminology of these two types of test requirements has been previously used in other works related to software testing in SBAs [25]. Figure 3.3 illustrates an example of the specification of Primitive Test Requirements for an SLA with three guarantee terms as well as the specification of Combined Test Requirements.

For each guarantee term represented in the figure a set of Primitive Test Requirements are identified. As we have introduced in Section 3.2.1, we will use four different values to evaluate each guarantee term. These evaluation values are represented in the figure with coloured squares (green for Fulfilled, red for Violated, grey for Not Determined and yellow for Inapplicable). The Primitive TRs are represented with solid circles under such evaluation values. For example, the Primitive TR remarked in the figure represents a situation in which the guarantee term GT1 is evaluated as Violated. It is worth mentioning that it is possible to identify one Primitive TR for each of the four aforementioned evaluation values.



	GT1	*GT2*	*GT3*
Combined TR1	Violated	Fulfilled	Inapplicable
...		...	
Combined TRn	Inapplicable	Not Determined	Fulfilled

Figure 3.3: Primitive and Combined Test Requirements.

Based on these Primitive TRs, different Combined Test Requirements are identified by combining one Primitive TR from each guarantee term. Hence, a Combined TR represents a scenario in which the test situations associated to a particular combination of Primitive TRs are exercised. For example, the first Combined TR (remarked in the figure) involves testing the situation in which GT1 is forced to be Violated, GT2 is evaluated as Fulfilled and GT3 is Inapplicable during the evaluation.

Bearing this classification in mind, the concept of test requirement can be defined as follows in the context of this dissertation:

***Definition 1**: A test requirement represents a situation related to the SUT in which one or more Guarantee Terms has to take a predetermined evaluation value and specific conditions must be satisfied.*

In addition to this, we will define different testing techniques that determine how the test requirements are obtained (from the individual guarantee terms or the relationships between such terms). Here again, we distinguish between two different testing levels (as we did during the SLA evaluation), depending on the test basis considered to identify the test requirements:

1. Guarantee Term Testing Level

2. Compositor (Logical relationships between Guarantee Terms) Testing Level.

The first of these levels will be addressed in Chapter 5 of this dissertation whereas the second level will be addressed in Chapter 6 of this dissertation.

3.2.2.1 Guarantee Term Testing Level

The identification of the test requirements in the first of these testing levels is represented in Figure 3.4.

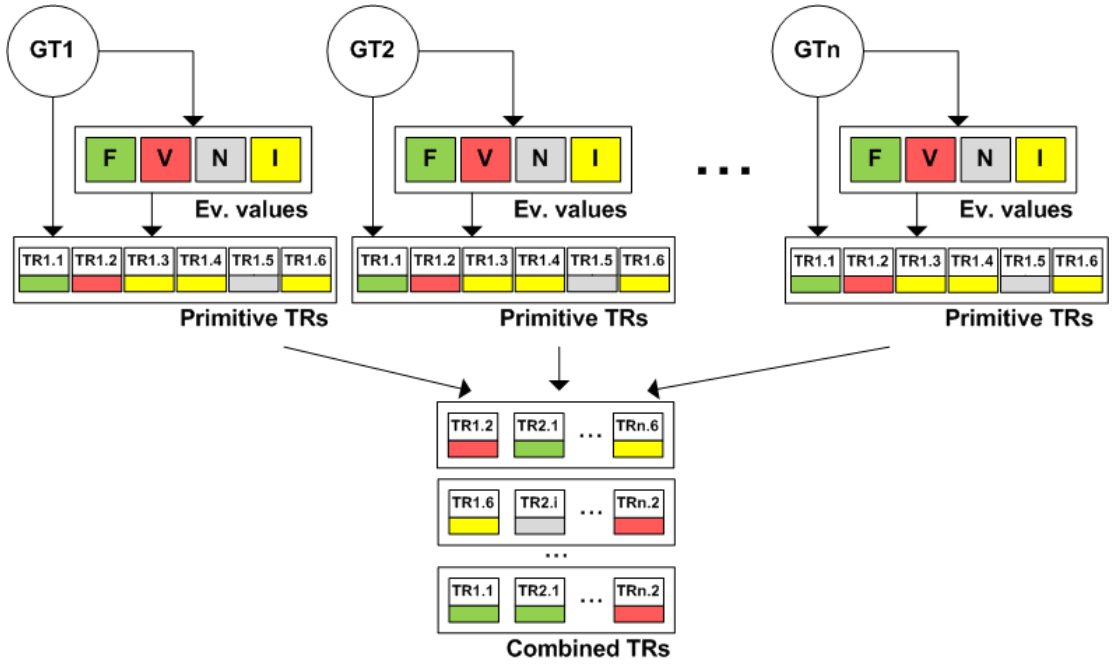


Figure 3.4: Guarantee Term Testing Level Test Requirements.

In the Guarantee Term testing level the Primitive TRs are obtained from the specification of the individual SLA guarantee terms. Each Primitive TR considers the conditions specified in the internal elements of the guarantee term (Scope, QC and SLO) as well as the evaluation value of such term. To be more specific, up to six Primitive TRs can be identified for each SLA guarantee term (top part of Figure 3.4) as we will describe in Chapter 5. These Primitive TRs are represented in the figure with squares. Each square contains the identifier of the test requirement in the top part and a coloured rectangle in the bottom part. The colour represents the evaluation value of the

guarantee term for such Primitive TR (green for Fulfilled, red for Violated, yellow for Inapplicable and grey for Not Determined).

After the identification of the Primitive TRs, combinatorial testing techniques [52] are applied in order to obtain the Combined Test Requirements. A Combined TR will contain one Primitive TR for each of the guarantee terms specified in the SLA (bottom part of Figure 3.4).

3.2.2.2 Compositor Testing Level

The identification of the test requirements in the second of the testing levels is represented in Figure 3.5.

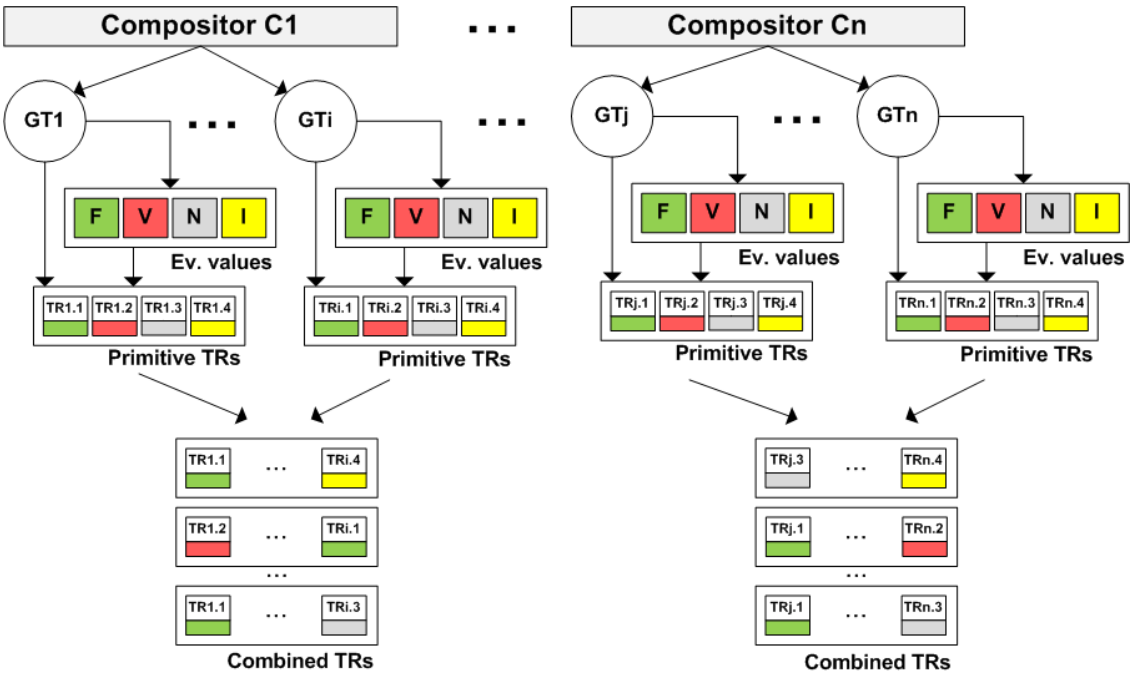


Figure 3.5: Compositor Testing Level Test Requirements.

In the second of the testing levels, we will define SLACDC (SLA Condition Decision Coverage), a coverage-based testing criterion that allows identifying test requirements taking the logical relationships of the SLA guarantee terms into account. Firstly, the Primitive TRs are identified from the individual guarantee terms. In this case, we obtain one Primitive TR for each of the evaluation values that the guarantee term can take. This means that we will identify four Primitive TRs for each guarantee

term (top part of Figure 3.5). After that, we will obtain the Combined TRs by means of combining the Primitive TRs applying SLACDC. A Combined TR will usually cover one Primitive TR for each of the guarantee terms contained in a compositor (bottom part of the Figure 3.5).

3.2.2.3 Additional issues

With the identification and combination of the test requirements, the problem of the combinatorial explosion may arise [52]. The number of situations related to the SLA that need to be tested could become unmanageable when the complexity of the SLA grows. The identification of the test requirements tries to maximize the trade-off among different criteria such as risks, models of the systems, likely failures, compliance requirements, expert advice or heuristics [61]. In some cases, it may be adequate to design an in-depth and exhaustive set of test requirements even if it involves a high cost in terms of money or effort. In other cases, however, there might be constraints hindering the definition the tests. When this happens, the tester is forced to select a less exhaustive testing technique. Hence, in each of the two aforementioned testing levels we will make design decisions in order to obtain a reasonable cost-effective set of test requirements.

Finally, we must also bear in mind that some of the identified test requirements may represent non-feasible situations regarding the SLA-aware service based application. In this case and based on the syntax and semantics of the SLA as well as the business logic of the SBA, we will define specific constraints with the aim at avoiding the obtaining of non-feasible test requirements. These constrains will be presented in Section 5.3.3 of this dissertation.

3.2.3 Generation of test cases

The objective of this activity is the derivation of a set of test cases that cover all the previously identified test requirements. A test case represents a scenario of the SBA that exercises one or more of the previously identified test requirements. The goal of generating the test cases is achieved by means of suitably selecting a set of test requirements to be covered, considering that the more test requirements that can be covered in a single test case, the fewer test cases will be needed in order to cover all the

requirements. Hence, we need to get a good balance between covering too many and too few test requirements in each test case.

The generation of the test cases depends on the way we have previously identified the set of Primitive and Combined Test Requirements. In the Guarantee Term Testing Level described in Section 3.2.2.1, a Combined TR involves the evaluation of all the guarantee terms specified in the SLA. Hence, such Combined TR represents a complete specific scenario of the SBA. This implies that a test case will generally cover only one Combined TR although it is also possible to design test cases that cover more than one Combined TR. In Figure 3.6 we have extended the diagram of Figure 3.4 in order to represent the relation between the Combined TRs and the test cases in the first of the testing levels.

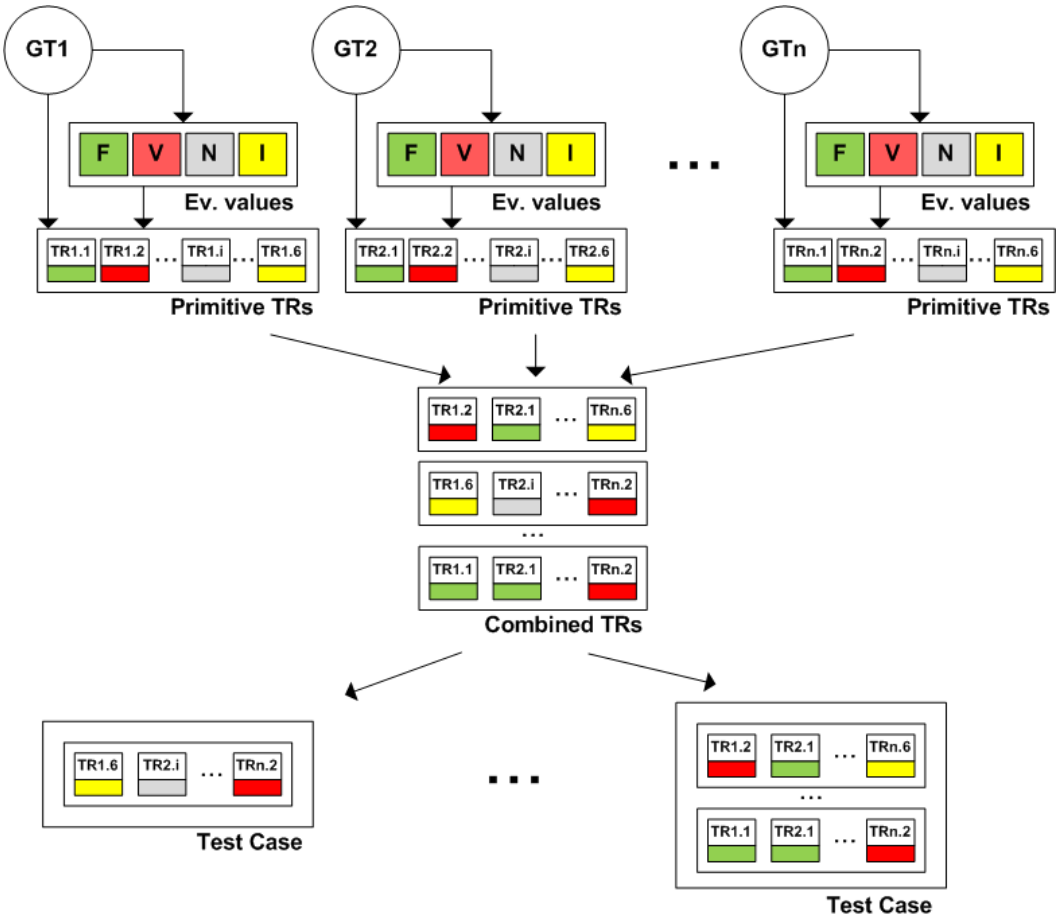


Figure 3.6: Relation Combined TR – Test Cases in the Guarantee Term Testing Level.

In the Compositor Testing Level described in Section 3.2.2.2, a Combined TR represents a scenario that only affects a specific part of the SLA (the guarantee terms involved in a compositor). In this case, a test case will usually cover one Combined TR for each of the compositors to represent a complete scenario related to the SBA. Here again, it is also possible to generate a test case in which different scenarios are sequentially exercised. In Figure 3.7 we have extended the diagram of Figure 3.5 to represent the relation between the Combined TRs and the test cases in this second testing level.

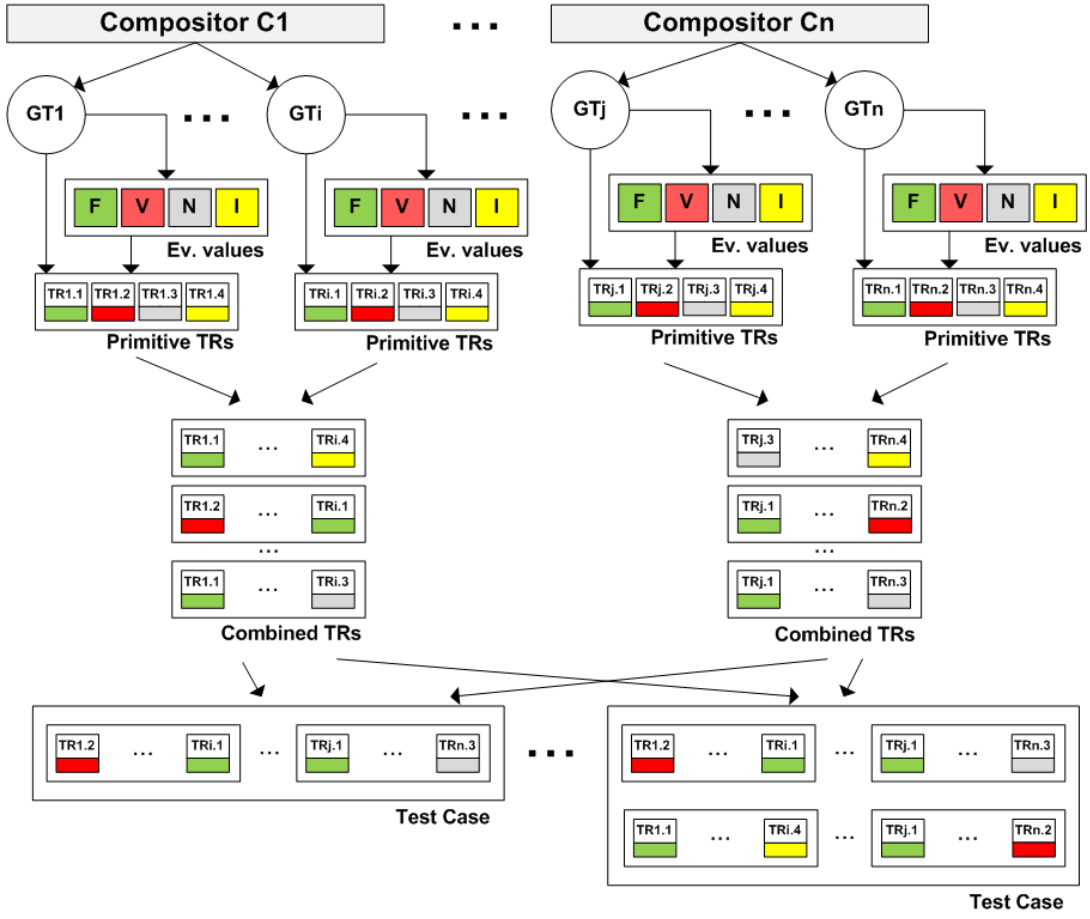


Figure 3.7: Relation Combined TR – Test Cases in the Compositor Testing Level.

The problem of the non-feasible situations arises again. Despite of having previously identified only feasible test requirements to be exercised, it does not mean that the combination of such test requirements constitute a scenario that makes sense. Typically, there will be Combined TRs that are incompatible to be combined within the

same test case due to the specification of the SLA or even the SBA so this task of generating the test cases by combining the Combined TRs is not definitely trivial and must be carefully performed. To address this issue, we have considered representing constraints that allow determining which Combined Test Requirements are mutually incompatible.

It is worth mentioning that both the identification of test requirements and the generation of test cases are often a tedious task so it is desirable to automate them as much as possible. In Chapter 7 of this dissertation we will describe SLACT (SLA Combinatorial Testing), a tool we have implemented to automate the identification of test requirements as well as the generation of test cases.

3.2.4 Execution of test cases

The execution of the derived test cases allows exercising the set of test requirements. Such executions are analyzed bearing in mind the SLA evaluation in order to determine whether the SBA behaves as expected. In spite of the fact that the execution of test cases is out of the scope of this PhD, below we outline some issues that need to be considered in order to complete the testing process described in SLATF.

First of all, it may be necessary to define a strategy in order to decide which test cases will be executed as well as the order of such executions. This is due to some factors that may hinder the testing process such as cost, excessive deadline pressure and so on. In these situations the test cases that are more likely to find a problem in the SBA should be executed first. Likewise, the test cases that aim at detecting more critical problems in the SBA should have more priority than the others. Aligned with this, a specific characteristic of SOA based systems needs to be considered in the sense that the tester does not probably have unlimited and full access to the services but executions usually imply an economic cost. Hence, the necessity to define a priority about the execution of the test cases and their priority becomes more critical.

A significant issue that also needs to be considered is whether all the test cases are executable or not. This is really important in software testing but even more in SOA systems because many times the tester does not have full control of the services or the testing environment so some test cases may become non-testable. In case he does not

have the capacity to configure the behaviour of the services properly regarding the specification of the test cases, he could need to implement mocks in order to mimic such behaviour.

3.3 Summary

In this chapter we have presented SLATF, a framework that allows testing SLA-aware service based applications. This framework involves different activities that need to be carried out, taking the specification of an SLA using the WS-Agreement standard language as the test basis.

The evaluation of the SLA is an orthogonal activity that affects all the testing process involved in the SLATF framework. All the details about such evaluation are fully addressed in Chapter 4 of this dissertation. This evaluation is a key issue in this research in the sense that it allows deciding whether the conditions of the SLA are being fulfilled or not and, furthermore, it contributes to the design of the tests.

The rest of the activities involved in SLATF are related to the design and execution of the tests. As the SLA contains both guarantee terms and compositors, two different testing levels are considered regarding the information of the SLA that is taken as the test basis. In each of these testing levels we will use specific testing techniques in order to identify and combine the test requirements, which can be of two different types: Primitive and Combined. The development of the testing process in each of the two testing levels is described in Chapter 5 and Chapter 6 of this dissertation respectively.

Finally, the automation of the aforementioned activities is addressed by implementing SLACT tool, which is fully described in Chapter 7 of this dissertation.

Chapter 4

SLA Evaluation

When code and comments disagree,
both are probably wrong.

Norm Schryer
Computer scientist

This chapter outlines the importance of the evaluation of SLAs in order to design tests as well as monitor the behavior of the service-based applications and detect derived problems. It introduces a four-valued logic that allows unequivocally evaluating the SLA and its internal elements.

4.1 Introduction

In the SLATF framework we have presented in Chapter 3 to test SLA-aware service based applications, the evaluation of an SLA is a key issue that requires determining whether the agreed conditions specified in the guarantee terms are being fulfilled or violated and contributes to design the tests. The decision about the evaluation is made by means of observing the behaviour of the constituent services of the SBA during their executions, collecting relevant data concerning the SLA and checking the specification of such SLA by analyzing the collected data.

Typically, the evaluation of an SLA as well as the evaluation of each guarantee term is performed in a dichotomic way. This means that it could be depicted with a two-way traffic light indicator with two colours: green if the SLA is fulfilled and red if the SLA is violated. For example, the dashboard represented in Figure 4.1 shows the results of the evaluation of an SLA that is associated to a SBA that implements the business of a Travel Agency.

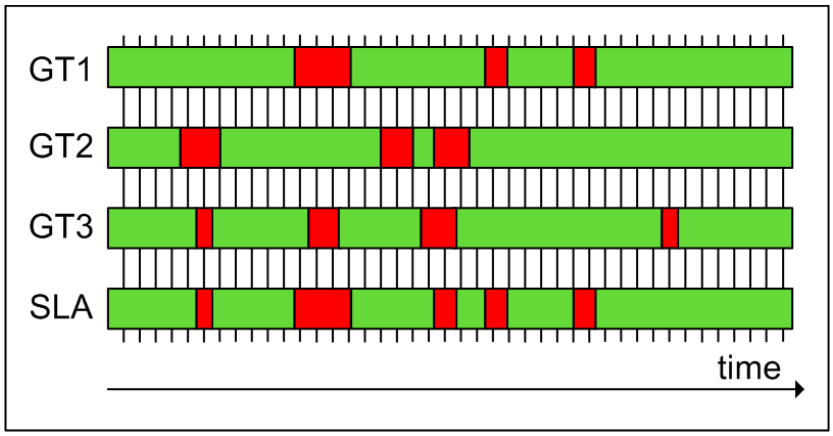


Figure 4.1: Dashboard with a two-way evaluation.

The Travel Agency uses a web service composition that invokes different services depending on the client's requests. The agency allows booking different transports such as flights or private cars as well as accommodation in hotels. Furthermore, the agency has different types of clients, which lead to specific offers or discounts.

The SLA represented in the figure contains three guarantee terms that are specified in the WS-Agreement standard language. As we introduced in Section 2.4, a

guarantee term is specified by means of the internal elements: Scope, Qualifying Condition and Service Level Objective. These guarantee terms are composed in this SLA as follows:

$$ALL\ (ev(GT1), OneOrMore(ev(GT2), ev(GT3)))$$

The evaluation of this SLA is equivalent to the following logical formula:

$$ev(SLA) = (\ ev(GT1)\ \wedge (\ ev(GT2)\ \vee\ ev(GT3)\ )\ )$$

This evaluation is performed by analyzing the information gathered from the monitors during the services executions. The green lights become red when a problem concerning the SLA has been detected. The evaluation of each guarantee term is depicted in the first three rows of the figure whereas the evaluation of the SLA as a whole is depicted in the last row. As can be seen, the aforementioned logical formula shows the direct relation between the evaluation of the guarantee terms and the evaluation of the SLA. The violation of GT1 leads to the violation of the SLA. Furthermore, this SLA is also violated when both guarantee terms GT2 and GT3 are being violated at the same time.

At this stage, the two typical values (Fulfilled / Violated) used in the dashboard of Figure 4.1 may not be enough to represent all the potential situations derived from the evaluation of an SLA and its guarantee terms that need to be tested. For example, there is a guarantee term that regulates the conditions that must be satisfied when the method *getDiscount* of the SBA is executed. This method provides the discount to be applied to the clients of the agency when booking its products. The method has been successfully executed during the monitoring period so the green led would be therefore permanently lighted in the dashboard. However and from a testing point of view, we do not have knowledge about how the SBA will behave when the method *getDiscount* is not executed. This means that despite the non-execution of *getDiscount*, the SBA has to properly manage the situation and continue arranging the client's order. Hence, it is necessary to design tests that aim at detecting potential problems in the SBA when *getDiscount* is not executed. In this context, a new evaluation value would be necessary to represent the aforementioned scenario in the dashboard.

Likewise, we have to consider the situation that arises when evaluating the guarantee term GT2. This term regulates the conditions that must be satisfied when the service *bookFlight* is invoked by a *VIP* client. This means that the Qualifying Condition of the term (*clientType = VIP*) determines whether such term is relevant during the evaluation of the SLA or not. As can be seen in the figure, the service *bookFlight* has been successfully executed by VIP clients except three gaps in which the term has been violated. However, we have no information about how the SBA will behave when a *non-VIP* client tries to book a flight. Hence, we need to exercise such situation with specific tests in order to detect any potential problem regarding the booking of a flight by a non-VIP client. Here again, this scenario cannot be represented in the dashboard with only two different colours and a new evaluation value would be necessary.

In addition to the potential values the SLA can take, the evaluation of such SLA depends on the evaluation of its guarantee terms and, consequently, on the logical relationships of such terms, as we introduced in Chapter 3. Hence, we identify two different levels regarding the evaluation of the SLAs:

- Level I: Individual Guarantee Terms.
- Level II: Compositor (logical relationships between Guarantee Terms) elements.

The first level involves making a decision about the fulfilment of each individual guarantee term represented in the SLA. The second level involves considering sets of guarantee terms logically grouped by compositor elements and determining whether these compositors are being fulfilled or not. Likewise, the evaluation of the guarantee terms allows evaluating the compositor elements and, recursively, the evaluation of all the compositors allows determining the final evaluation of the whole SLA. For example, the evaluation of GT2 and GT3 represented in Figure 4.1 allows evaluating the *OneOrMore* compositor. Likewise, the evaluation of such compositor and GT1 allows evaluating the *All* compositor and, consequently, the SLA.

In this section we propose a logic that allows evaluating both individual guarantee terms and compositors, from a testing point of view, including the potential situations

derived from the decision about whether the SLA is being fulfilled or not. The use of this logic in the SLATF framework described in Section 3.2 tackles two different but important objectives in the sense that it allows: on the one hand, the obtaining of the expected evaluation value of the SLA and its internal elements (including both Guarantee Terms and Compositors). On the other hand, it also allows guiding the identification of test requirements through the application of coverage criteria to the guarantee terms and compositor elements specified in the SLA.

4.2 Evaluation of Guarantee Terms

In this section we focus on each individual guarantee term in order to address the evaluation of the SLA. To deal with this issue, we firstly introduce the concept of evaluation value and we anticipate the potential evaluation values that can be taken:

***Definition 2**: An evaluation value is the output provided by the mechanism in charge of making a decision about the fulfilment of a guarantee term, a compositor or an SLA. There are four different evaluation values: (F) Fulfilled, (V) Violated, (N) Not Determined and (I) Inapplicable.*

This definition means that an element of the SLA denoted by t can be evaluated with those four evaluation values using a function *ev(t)*:

$$ev(t) = \{Fullfilled, Violated, Not\ Determined, Inapplicable\}$$

As we have outlined in Section 2.4, a guarantee term in WS-Agreement is composed of the internal elements Scope, Qualifying Condition (QC) and Service Level Objective (SLO). These elements and their evaluation are represented in Figure 4.2.

According to the semantics of such elements, it is the Qualifying Condition element who determines whether the guarantee term is relevant and requires to be evaluated. This means that if the QC is not satisfied, the guarantee term does not have to be taken into account during the evaluation process. If the QC is satisfied, then we check whether the service specified in the Scope is executed and, consequently, the fulfillment of the Service Level Objective (SLO).

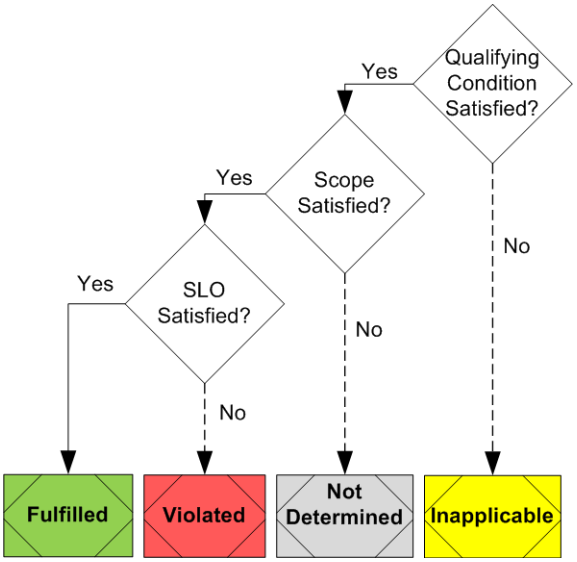


Figure 4.2: SLA evaluation values.

Bearing Figure 4.2 in mind, a guarantee term specified in WS-Agreement is evaluated with the two classical evaluation values as follows:

- **FULFILLED** - if and only if the methods of the services specified in the *Scope* have been executed, the *Qualifying Condition* has been met and the *Service Level Objective* has been satisfied.

- **VIOLATED** - if and only if the methods of the services specified in the *Scope* have been executed, the *Qualifying Condition* has been met and the *Service Level Objective* has not been satisfied.

In addition to these two evaluation values, a third potential value named *Not Determined* that a term can take after its evaluation can be identified and arises when the service associated to the term has not been invoked. The interpretation of this value according to WS-Agreement is that, at the moment of the evaluation, no activity regarding the term has happened yet or no activity is currently happening that allows evaluating whether the term is fulfilled or violated. Actually, WS-Agreement identifies these three situations as the potential *runtime states* of an SLA. Hence, a guarantee term is evaluated as:

- **NOT DETERMINED** - if and only if the methods of the services specified in the *Scope* have not been executed and the *Qualifying Condition* has been met.

However, apart from these three values outlined in WS-Agreement, we have identified another specific situation where the term can be found after its evaluation and which has not been explicitly identified in WS-Agreement. This new situations arises when the Qualifying Condition of the term is not met during the execution of services. In this case, the Guarantee Term becomes invalid and it must not be taken into account for the purpose of the evaluation of the SLA so we say that a Guarantee Term is evaluated as:

- **INAPPLICABLE** - if and only if the Qualifying Condition has not been satisfied.

In other fields within the software engineering, it has been necessary to extend the typical binary logic (true / false) to deal with similar situations. For example, in the context of Database Management Systems (DBMS) the interpretation of the missing information is considered by means of a third value (null), which has also been broadly used in the scope of database applications testing [8][29][49][131]. In our case, the use of these two additional evaluation values (Not Determined and Inapplicable) could represent an analogous interpretation of the treatment of the null value in DBMS and leads to a four-valued logic to evaluate SLAs.

Example of guarantee term in WS-Agreement and its evaluation

Figure 4.3 shows an example of a guarantee term specified in WS-Agreement and its evaluation according to the diagram depicted in Figure 4.2.

This guarantee term affects the method *getDiscount* of the service *WSBookFlight*, which is part of the web service composition implemented by the Travel Agency, as we introduced in Section 4.1 of this chapter. In this case, the client contacts the agency in order to book a flight. To be more specific, the aforementioned method must provide the discount to be applied in the transaction. Regarding the evaluation of the term, if the method *getDiscount* is invoked in order to provide a discount for a *VIP client*, the response time will determine whether the term is evaluated as Fulfilled or Violated (left

part of Figure 4.3). In addition to this, if the method *getDiscount* is not executed during the web service composition when a *VIP client* tries to book a flight, we do not have information yet to say that the guarantee term has been satisfied or violated so, according to the four-valued logic, this term is evaluated as Not Determined (central part of Figure 4.3). Finally, in case the operation is performed by a *non-VIP client*, then the Qualifying Condition is not satisfied so the guarantee term becomes irrelevant for the SLA evaluation and, according to this logic, the term is evaluated as Inapplicable (right part of Figure 4.3).

```
<GuaranteeTerm>
    Name = "GT_VIP_Client" Obligated = "ServiceProvider"
    <Scope>
        serviceName = "WSBookFlight" method = "getDiscount"
   </Scope>
   <QualifyingCondition>
        clientType = "VIP"
   </QualifyingCondition>
   <ServiceLevelObjective>
        response time < 3
   </ServiceLevelObjective>
</GuaranteeTerm>
```

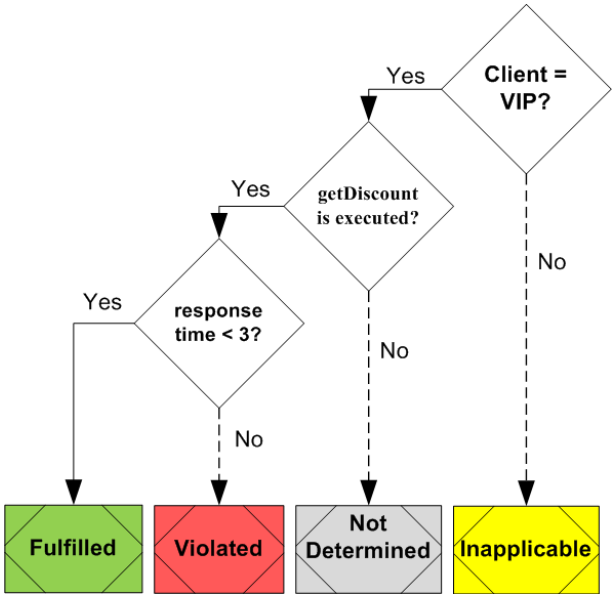


Figure 4.3: Evaluation of a Guarantee Term in WS-Agreement.

4.3 Evaluation of Compositor elements

After having described a systematic way to evaluate each individual SLA Guarantee Term, in this section we focus on the logical combinations of such terms. We have previously outlined in Section 2.4 that an SLA specified in WS-Agreement represents a hierarchical structure of guarantee terms, logically combined using the specific Compositor Elements *All*, *OneOrMore* and *ExactlyOne*, which are equivalent to the AND, OR and XOR logical operators, respectively. Thus, we complete the logic that allows evaluating the individual Guarantee Terms in order to unequivocally determine the evaluation value of these compositors.

According to the semantics of each compositor, in the following subsections we define how the compositor elements will be evaluated.

4.3.1 All Compositor

The *All* compositor in WS-Agreement is the equivalent to the classical AND logical operator in the sense that it requires that all its guarantee terms must be fulfilled. Hence, an *All* compositor element with multiple Guarantee Terms is evaluated as follows:

$$ev(All_{i=1}^{n}(t_i)) =$$

- Fulfilled if

$$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{F}) \wedge$$
$$(\forall j \in [1,n], j \neq i : (ev(t_j) = \boldsymbol{F} \vee ev(t_j) = \boldsymbol{I}))$$

- Violated if

$$\exists i \in [1,n] : ev(t_i) = \boldsymbol{V}$$

- Not Determined if

$$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{N}) \wedge (\nexists j \in [1,n], j \neq i : ev(t_j) = \boldsymbol{V})$$

- Inapplicable if

$$\forall i \in [1,n] : ev(t_i) = \boldsymbol{I}$$

The interpretation of this evaluation outcome is that an *All* compositor element with n guarantee terms is evaluated as Fulfilled if at least one of its guarantee terms has been evaluated as Fulfilled and the rest of such terms have been evaluated as Fulfilled or Inapplicable. The same compositor is evaluated as Violated when there is at least one guarantee term that has been evaluated as Violated. The *All* compositor is evaluated as Not Determined if there is at least one guarantee term evaluated as Not Determined and none of the rest of the guarantee terms has been evaluated as Violated. Finally, the *All* compositor is evaluated as Inapplicable if all its guarantee terms have been evaluated as Inapplicable.

In Table 4.1 we represent the evaluation value of an *All* compositor with three guarantee terms. In the first three columns we represent the multiple combinations of the evaluation values of such guarantee terms. In the last column we represent the evaluation value of the compositor (for example, if GT1 is evaluated as Fulfilled, GT2 is evaluated as Inapplicable and GT3 is evaluated as Violated, the *All* compositor is evaluated as Violated). The number of potential combinations is 64 (4^n, being n the number of guarantee terms in the compositor).

GT1	GT2	GT3	All
F	F	F	F
F	F	V	V
F	F	N	N
F	F	I	F
F	V	F	V
F	V	V	V
F	V	N	V
F	V	I	V
F	N	F	N
F	N	V	V
F	N	N	N
F	N	I	N
F	I	F	F
F	I	V	V
F	I	N	N
F	I	I	F
V	F	F	V
V	F	V	V
V	F	N	V
V	F	I	V
V	V	F	V
V	V	V	V
V	V	N	V
V	V	I	V
V	N	F	V
V	N	V	V
V	N	N	V
V	N	I	V
V	I	F	V
V	I	V	V
V	I	N	V
V	I	I	V

GT1	GT2	GT3	All
N	F	F	N
N	F	V	V
N	F	N	N
N	F	I	N
N	V	F	V
N	V	V	V
N	V	N	V
N	V	I	V
N	N	F	N
N	N	V	V
N	N	N	N
N	N	I	N
N	I	F	N
N	I	V	V
N	I	N	N
N	I	I	N
I	F	F	F
I	F	V	V
I	F	N	N
I	F	I	F
I	V	F	V
I	V	V	V
I	V	N	V
I	V	I	V
I	N	F	N
I	N	V	V
I	N	N	N
I	N	I	N
I	I	F	F
I	I	V	V
I	I	N	N
I	I	I	I

Table 4.1: Truth table of an All compositor with three guarantee terms.

4.3.2 OneOrMore Compositor

Likewise, an *OneOrMore* is the equivalent to the classical OR logical operator in the sense that it requires that at least one guarantee term must be fulfilled. Hence, an *OneOrMore* compositor element with multiple Guarantee Terms is evaluated as follows:

$$ev(OneOrMore_{i=1}^{n}(t_i)) =$$

- Fulfilled if

$$\exists i \in [1,n] : ev(t_i) = \boldsymbol{F}$$

- Violated if

$$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{V}) \wedge$$
$$(\forall j \in [1,n], j \neq i : (ev(t_j) = \boldsymbol{V} \ \vee \ ev(t_j) = \boldsymbol{I}))$$

- Not Determined if

$$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{N}) \wedge (\nexists j \in [1,n], j \neq i : ev(t_j) = \boldsymbol{F})$$

- Inapplicable if

$$\forall i \in [1,n] : ev(t_i) = \boldsymbol{I}$$

In Table 4.2 we represent the evaluation value of an *OneOrMore* compositor with three guarantee terms. In the first three columns we represent the multiple combinations of the evaluation values of such guarantee terms. In the last column we represent the evaluation value of the compositor (for example, if GT1 is evaluated as Inapplicable, GT2 is evaluated as Fulfilled and GT3 is evaluated as Not Determined, the *OneOrMore* compositor is evaluated as Fulfilled).

GT1	GT2	GT3	OneOrMore
F	F	F	F
F	F	V	F
F	F	N	F
F	F	I	F
F	V	F	F
F	V	V	F
F	V	N	F
F	V	I	F
F	N	F	F
F	N	V	F
F	N	N	F
F	N	I	F
F	I	F	F
F	I	V	F
F	I	N	F
F	I	I	F
V	F	F	F
V	F	V	F
V	F	N	F
V	F	I	F
V	V	F	F
V	V	V	V
V	V	N	N
V	V	I	V
V	N	F	F
V	N	V	N
V	N	N	N
V	N	I	N
V	I	F	F
V	I	V	V
V	I	N	N
V	I	I	V

GT1	GT2	GT3	OneOrMore
N	F	F	F
N	F	V	F
N	F	N	F
N	F	I	F
N	V	F	F
N	V	V	N
N	V	N	N
N	V	I	N
N	N	F	F
N	N	V	V
N	N	N	N
N	N	I	N
N	I	F	F
N	I	V	N
N	I	N	N
N	I	I	N
I	F	F	F
I	F	V	F
I	F	N	F
I	F	I	F
I	V	F	F
I	V	V	V
I	V	N	N
I	V	I	V
I	N	F	F
I	N	V	N
I	N	N	N
I	N	I	N
I	I	F	F
I	I	V	V
I	I	N	N
I	I	I	I

Table 4.2: Truth table of an OneOrMore compositor with three guarantee terms

4.3.3 ExactlyOne Compositor

Finally, an *ExactlyOne* compositor element is the equivalent to the XOR logical operator. This means that only one guarantee term must be fulfilled. Hence, an *ExactlyOne* compositor with multiple Guarantee Terms is evaluated as follows:

$$ev(ExactlyOne_{i=1}^{n}(t_i)) =$$

- Fulfilled if

$$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{F}) \wedge$$
$$(\nexists j \in [1,n], j \neq i : (ev(t_j) = \boldsymbol{F}) \vee (ev(t_j) = \boldsymbol{N}))$$

- Violated if

$$(\forall i \in [1,n] : ev(t_i) = \boldsymbol{V}) \vee$$
$$(\exists j,k \in [1,n], j \neq k : (ev(t_j) = ev(t_k) = \boldsymbol{F}))$$

- Not Determined if

$$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{N}) \wedge$$
$$(\nexists j,k \in [1,n], j \neq k : (ev(t_j) = ev(t_k) = \boldsymbol{F}))$$

- Inapplicable if

$$\forall i \in [1,n] : ev(t_i) = \boldsymbol{I}$$

In Table 4.3 we represent the evaluation value of an *ExactlyOne* compositor with three guarantee terms. In the first three columns we represent the multiple combinations of the evaluation values of such guarantee terms. In the last column we represent the evaluation value of the compositor (for example, if GT1 is evaluated as Fulfilled, GT2 is evaluated as Inapplicable and GT3 is evaluated as Fulfilled, the *ExactlyOne* compositor is evaluated as Violated).

GT1	GT2	GT3	ExactlyOne
F	F	F	V
F	F	V	V
F	F	N	V
F	F	I	V
F	V	F	V
F	V	V	F
F	V	N	N
F	V	I	F
F	N	F	V
F	N	V	N
F	N	N	N
F	N	I	N
F	I	F	V
F	I	V	F
F	I	N	N
F	I	I	F
V	F	F	V
V	F	V	F
V	F	N	N
V	F	I	F
V	V	F	F
V	V	V	V
V	V	N	N
V	V	I	V
V	N	F	N
V	N	V	N
V	N	N	N
V	N	I	N
V	I	F	F
V	I	V	V
V	I	N	N
V	I	I	V

GT1	GT2	GT3	ExactlyOne
N	F	F	V
N	F	V	N
N	F	N	N
N	F	I	N
N	V	F	N
N	V	V	N
N	V	N	N
N	V	I	N
N	N	F	N
N	N	V	N
N	N	N	N
N	N	I	N
N	I	F	N
N	I	V	N
N	I	N	N
N	I	I	N
I	F	F	V
I	F	V	F
I	F	N	N
I	F	I	F
I	V	F	F
I	V	V	V
I	V	N	N
I	V	I	V
I	N	F	N
I	N	V	N
I	N	N	N
I	N	I	N
I	I	F	F
I	I	V	V
I	I	N	N
I	I	I	I

Table 4.3: Truth table of an ExactlyOne compositor with three guarantee terms.

4.4 Recursive Evaluation

An SLA represents a hierarchical structure that contains compositor elements and guarantee terms. Moreover, the compositors can be composed of guarantee terms or other compositors. Hence, the evaluation of the SLA is performed by means of recursively evaluating its internal elements, taking both guarantee terms and compositors into account. In addition to this, it is worth mentioning that a WS-Agreement always specifies the content of the whole agreement under an *All* external compositor element so the evaluation of the SLA would be equivalent to the evaluation of such most external *All* element.

Figure 4.4 shows an example of this recursive evaluation. The SLA depicted in the figure contains three compositors and six guarantee terms under the most external *All* compositor. All of these elements are colored depending on their evaluation value (green for Fulfilled, red for Violated, grey for Not Determined and yellow for Inapplicable).

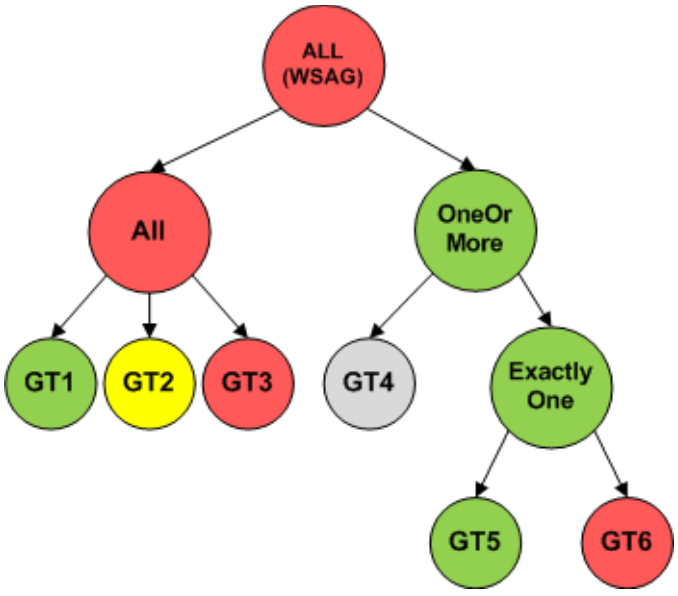


Figure 4.4: Example of recursive evaluation.

In the left part of the figure, the All compositor is evaluated as Violated because one of its guarantee terms (GT3) is evaluated as Violated. In the right part of the figure, the *ExactlyOne* compositor is evaluated as Fulfilled because only one of its guarantee terms is Fulfilled. Likewise, the *OneOrMore* compositor is evaluated as Fulfilled

because, at least, one of its internal elements (the ExactlyOne compositor, in this case) is evaluated as Fulfilled although the guarantee term GT4 is evaluated as Not Determined. The evaluation values of these elements determine that the most external *All* compositor and, consequently the whole SLA, is evaluated as Violated.

4.5 Summary

In this chapter we have presented the logic we have devised in order to evaluate both the individual Guarantee Terms of an SLA specified in the WS-Agreement standard language and their relationships by means of using the Compositor elements.

In the beginning of the chapter we have described why we consider that a binary logic with its classical values (true / false or its equivalent Fulfilled / Violated) is not enough to deal with all the potential situations derived from the evaluation of the SLA when the objective is to design tests.

After, we have defined a four-valued logic that allow us to address different activities involved in the SLATF framework presented in Chapter 3: on the one hand, the obtaining of the expected evaluation value of the SLA and, on the other hand, guiding the identification of test requirements and generation of test cases from the specification of the SLA.

In the following Chapter 5 and Chapter 6 we will describe the use of the proposed logic to identify test requirements taken the specification of the SLA into account.

Chapter 5

Guarantee Term Testing Level

Research is to see what everybody else has seen,
and to think what nobody else has though.

Albert Szent-Gyorgyi, 1893-1986
Hungarian Biochemist, 1937 Nobel Prize for Medicine

This chapter addresses the generation of tests considering the information contained in the individual SLA guarantee terms. It firstly outlines how to identify test requirements by using the evaluation logic presented in the previous chapter. After that, it describes how these test requirements are combined by tailoring different testing techniques, including the Classification Tree Method (CTM) and combinatorial testing. Finally it specifies a set of rules that allow avoiding the generation of non-feasible tests.

5.1 Introduction

In Chapter 3 we introduced SLATF, a framework that aims at testing SLA-aware service based applications. This framework takes the specification of the SLA as the test basis, distinguishing between two different testing levels depending on whether we analyze the individual guarantee terms or the complete logical structure of the SLA. As part of SLATF, in Chapter 4 we defined the logic that allows evaluating the elements of the SLA. In this chapter we focus on the first of the testing levels by considering the specification of the guarantee terms in WS-Agreement in order to identify the test requirements and derive the test cases.

To address this issue different tasks, which are the realization of the activities defined in SLATF, must be carried out. First of all, we have to identify the set of situations related to the SLA that need to be tested (Primitive Test Requirements). This identification is based on the syntax and semantics of each guarantee term and must take the evaluation logic described in Section 4.2 into account, trying to avoid the obtaining of situations that cannot be exercised.

Once the test requirements have been obtained, we have to combine them in order to derive the Combined Test Requirements. Such Combined TRs will later be exercised by means of the generation of test cases.

This process may present two main problems:

1. The obtaining of an unmanageable number of Combined Test Requirements.

2. The generation of invalid Combined TRs due to the non-feasible combinations of specific Primitive TRs.

We deal with the first of these problems by applying standard combinatorial testing techniques, which allow grading the intensity of the tests. Depending on the technique applied, we will obtain a specific coverage of the Primitive TRs, considering that the stronger coverage we expect, the higher number of Combined TRs will be obtained.

The second of the aforementioned problems is addressed by means of the definition of specific constraints to guide the combinations of the test requirements. These constraints may be obtained by analyzing the specification of the SLA and any knowledge regarding the behaviour of the service based application. By considering these constraints, we assure that the Combined TRs obtained do not present non-feasible combinations of Primitive TRs.

The identification and combination of the test requirements is a laborious as well as a tedious task so we have implemented SLACT, a tool that automates each of the tasks presented along this chapter. The details of such tool are described in Chapter 7.

5.2 Identification of Primitive Test Requirements

The first activity of SLATF involves the identification of test requirements by analyzing the information contained in the SLA. To address this identification, we make use of the four-valued logic. To be more specific, in this testing level we are considering the specification of the individual guarantee terms of the SLA as the test basis so we focus on the logic defined in Section 4.2 to evaluate a guarantee term. According to such logic, there are four different values of evaluation for a guarantee term. At first glance, we could consider that it is necessary to identify four different situations with the aim at achieving full coverage while evaluating the guarantee term. However, the internal syntactic structure and the semantics of a guarantee term specified in the WS-Agreement standard language require a more complete coverage criterion to represent all the potential situations that are interesting to observe or exercise from a testing point of view.

At this stage, it is important to distinguish between the concepts of evaluation value and test requirement. On the one hand, we described *evaluation value* in Chapter 4 (Definition 2) as the output provided by the mechanism that makes the decision about the fulfilment of a guarantee term, a compositor or an SLA. Such evaluation can take four different values: Fulfilled, Violated, Inapplicable and Not Determined. On the other hand, we described *test requirement* in Chapter 3 (Definition 1) as a situation that involves the evaluation of one or more guarantee terms and the exercitation of specific conditions.

In this context and during the exercitation of the constituent conditions of the test requirement, the four-valued logic defined in Chapter 4 is used to provide the final *evaluation value* for the *test requirement*. In Figure 5.1, we show an example of a Guarantee Term specified in WS-Agreement of the Travel Agency SBA, where a test requirement is identified when such guarantee term takes the Violated evaluation value.

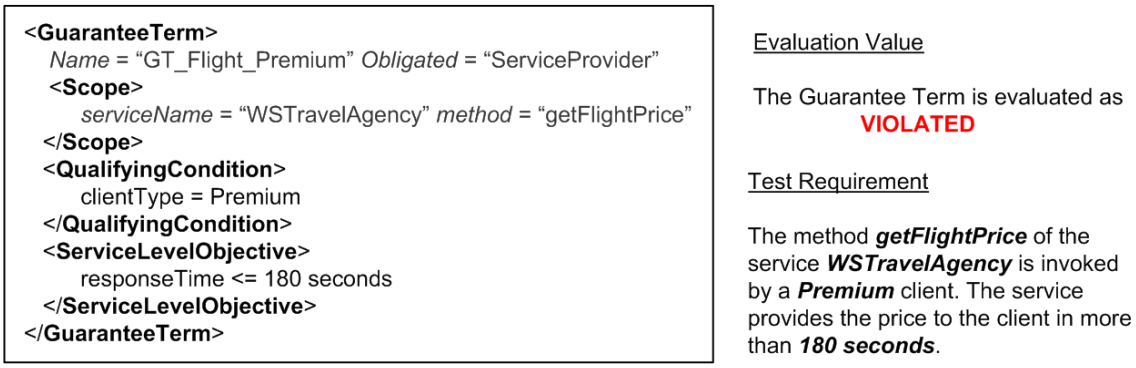


Figure 5.1: Relation Test Requirement – Evaluation Value.

This test requirement exercises the situation that involves the following conditions: the method getFlightPrice from the service WSTravelAgency specified in the Scope is executed, the invocation is performed by a Premium client so the Qualifying Condition is met and, finally, the response time is higher than 180 seconds so the Service Level Objective is not satisfied. Hence, the evaluation value for the *GT_Flight_Premium* guarantee term is Violated according to the aforementioned conditions.

5.2.1 General Case

Keeping the definition of an evaluation value in mind, a guarantee term may be evaluated with four different values but it may involve exercising different situations. From a testing point of view and according to the syntax of a Guarantee Term, these situations arise when checking the conditions specified in the internal elements of such guarantee term: Scope, Qualifying Condition and Service Level Objective. These elements are represented using a decision tree in Figure 5.2.

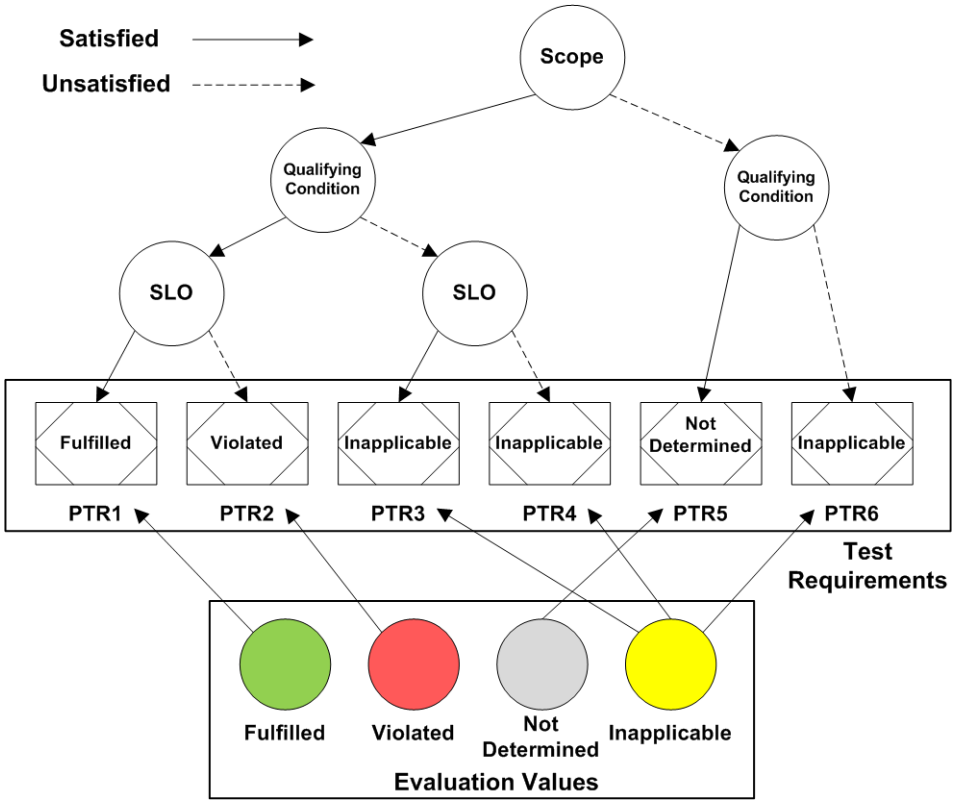


Figure 5.2: Test Requirements from a Guarantee Term.

First of all, we check whether the methods of the services specified in the Scope element have been executed or not (the verification of this condition is performed using satisfied/unsatisfied as outputs). After that, we check whether the Qualifying Condition of the term is satisfied or not. This condition determines whether the guarantee term becomes relevant or irrelevant during the evaluation process. Finally, we check whether the guarantee specified in the Service Level Objective is satisfied or not.

If we apply the multiple combinations of these three internal elements of a Guarantee Term, we will obtain 8 situations to test. However, as can be seen in the figure, only six situations are feasible according to the semantic meaning of such internal elements. The other two non-feasible situations relate to cases where the methods of the services specified in the Scope have not been executed so it is impossible to check whether the Service Level Objective has been satisfied or not (see right branch of the figure).

The feasible combinations of the internal elements of a guarantee term lead to the identification of six Primitive Test Requirements for such guarantee term (represented in the figure by PTR1-PTR6). In the left branch of the figure, four Primitive TRs identified as PTR1-PTR4 are obtained when the methods of the services specified in the Scope are executed:

PTR1 The methods of the services are invoked, the Qualifying Condition is satisfied and the Service Level Objective is satisfied (GT evaluated as Fulfilled).

PTR2 The methods of the services are invoked, the Qualifying Condition is satisfied and the Service Level Objective is unsatisfied (GT evaluated as Violated).

PTR3 The methods of the services are invoked, the Qualifying Condition is unsatisfied and the Service Level Objective is satisfied (GT evaluated as Inapplicable).

PTR4 The methods of the services are invoked, the Qualifying Condition is unsatisfied and the Service Level Objective is unsatisfied (GT evaluated as Inapplicable).

Apart from these four Primitive TRs, we also consider those situations where the methods of the services specified in the Scope element have not been invoked at the time of the evaluation (right branch of Figure 5.2). Namely, we include what happens when the Qualifying Condition is satisfied / unsatisfied while the methods of the services are not executed. For each Guarantee Term, other two test requirements identified as PTR5-PTR6 are identified as well.

PTR5 The methods of the services are not executed while the Qualifying Condition is satisfied (GT evaluated as Not Determined).

PTR6 The methods of the services are not executed while the Qualifying Condition is unsatisfied (GT evaluated as Inapplicable).

5.2.2 Particular Cases

In addition to the general case, we have to deal with an important issue regarding the identification of the Primitive Test Requirements. Depending on the internal syntax and semantics of the Guarantee Terms of WS-Agreement, we have to consider two particular cases where not all the six Primitive TRs are identified. These two cases are described below.

5.2.2.1 PC1: Guarantee terms without Qualifying Condition

The first particular case (PC1) arises when the Guarantee Term has no Qualifying Condition associated. The Qualifying Condition determines whether a term is relevant and it must be considered during the evaluation process or not. In this case and given that there is no Qualifying Condition the term is always relevant so only three Primitive Test Requirements (PTR1, PTR2 and PTR5) are identified. Furthermore, the specification of the Primitive Test Requirements PTR1 and PTR2 must be adapted as "The methods of the services are invoked and the Service Level Objective is satisfied / unsatisfied" respectively and test requirement PTR5 as "The methods of the services are not executed (GT evaluated as Not Determined)". In Figure 5.3 the application of this particular case is represented. At the top of the figure a guarantee term without Qualifying Condition is specified in WS-Agreement whereas the identified Primitive Test Requirements are represented at the bottom.

```
<GuaranteeTerm>
    Name = "GT_BookCar" Obligated = "ServiceProvider"
    <Scope>
        serviceName = "TravelAgency" method = "bookCar"
   </Scope>
   <ServiceLevelObjective>
        responseTime < 50
   </ServiceLevelObjective>
</GuaranteeTerm>
```

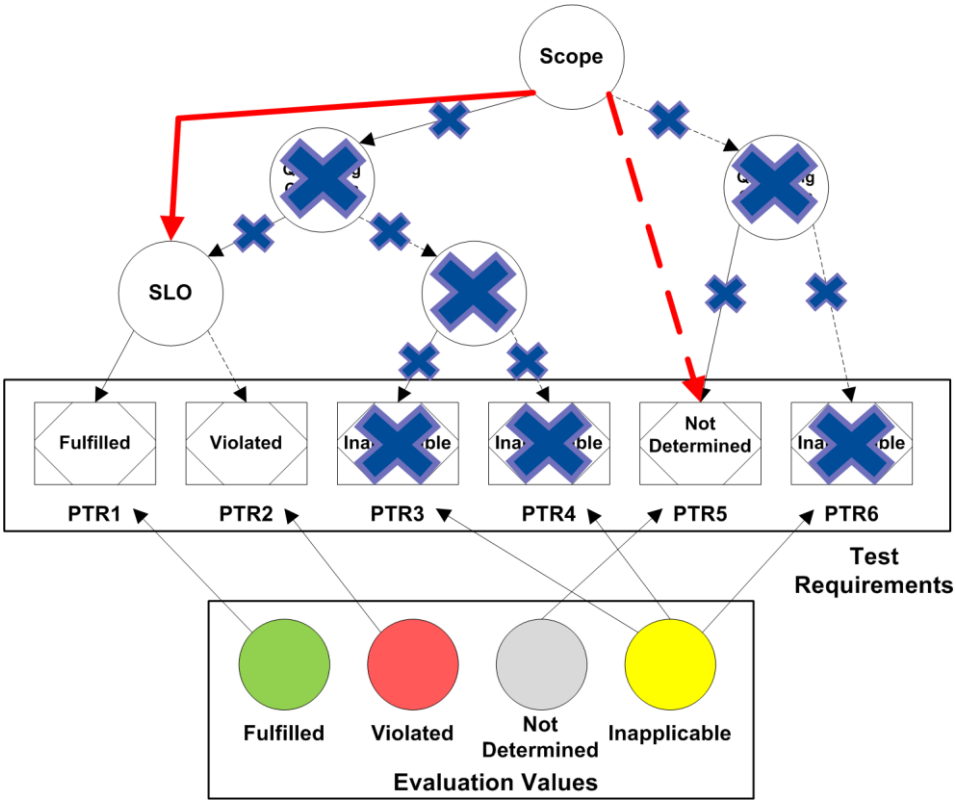


Figure 5.3: Particular Case 1: Guarantee Term without Qualifying Condition.

5.2.2.2 PC2: Qualifying Condition is an assertion over service attributes

WS-Agreement states in its specification that the Qualifying Condition is an assertion over service attributes and/or external factors. In the former case, for example, this condition may make reference to an input parameter or condition of the service while in the latter it can represent a specific state of the SUT. The second particular case (PC2) arises when the Qualifying Condition of the Guarantee Term is an assertion over the service attributes. This case occurs because the semantics of the Qualifying Condition also affect the identification of the test requirements. In this case, it is impossible to check the fulfilment of the QC if the methods of the services have not been executed so Primitive Test Requirements PTR5 and PTR6 will not be identified. In such case, test requirements PTR5 and PTR6 are joined in only one as "The methods of the services are not executed (GT evaluated as Not Determined)" so we would obtain one Primitive Test Requirement less than in the general case. In Figure 5.4 the application of this second particular case is represented. At the top of the figure a

guarantee term where the Qualifying Condition is an assertion over service attributes is depicted whereas the identified Primitive Test Requirements are represented at the bottom of such figure.

```
<GuaranteeTerm>
    Name = "clientPremium" Obligated = "ServiceProvider"
    <Scope>
        serviceName = "TravelAgency" method = "getPrice"
   </Scope>
   <QualifyingCondition>
        clientType = Premium
   </QualifyingCondition>
   <ServiceLevelObjective>
        responseTime < 10
   </ServiceLevelObjective>
</GuaranteeTerm>
```

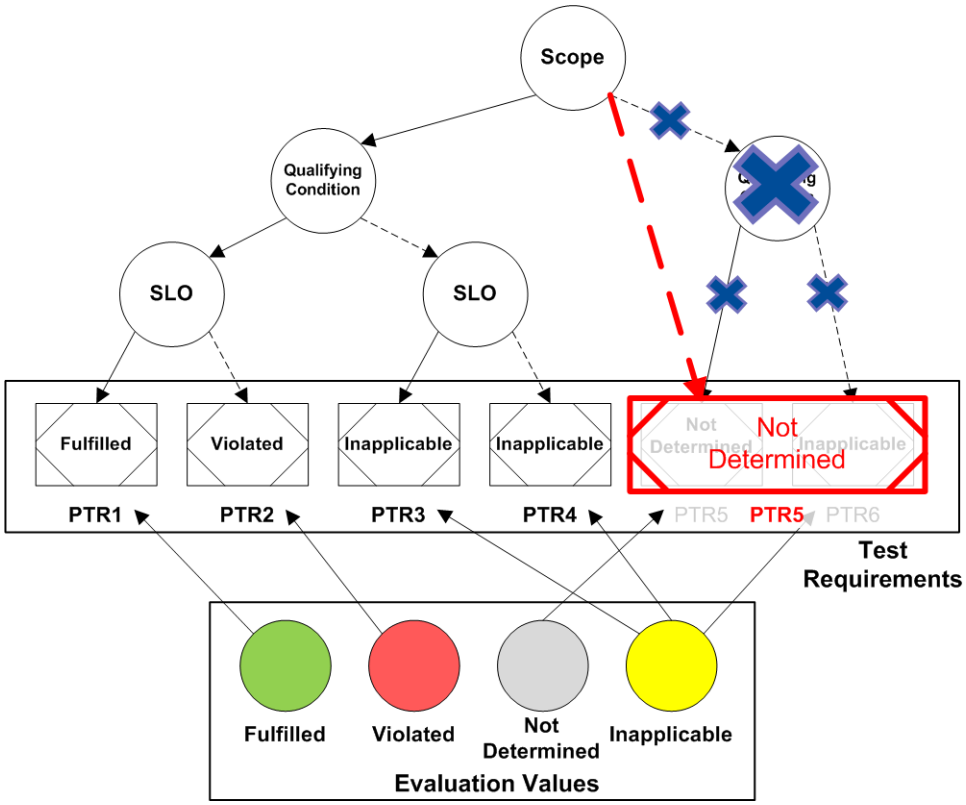


Figure 5.4: Particular Case 2: the Qualifying Condition is an assertion over the service attributes.

5.2.3 Categorization of Primitive Test Requirements

The aforementioned identification of the Primitive TRs is related to different testing objectives. Likewise, we have defined a categorization of such test requirements so as the tester has the capability to decide which testing objectives are going to be prioritized. Hence, this categorization can be used to identify a reduced set of test requirements instead of obtaining the whole set of Primitive Test Requirements from the terms of the SLA. Furthermore, although it is not an issue to address in the scope of this dissertation, this categorization may also be used to establish monitoring objectives, making a decision about the characteristics of the SUT that are more interesting to be observed at runtime.

Table 5.1 displays the categorization of test requirements according to their meaning or testing objective. The first column of this table shows the identifier of each category. The second column outlines the description of the testing objective of the category. Finally, last column lists the Primitive Test Requirements that are included in such category.

Category	Testing Objective	Primitive TRs
C1	Expected behaviour of the SUT	PTR1
C2	Test the behaviour after a term violation	PTR2
C3	Testing need indicator while monitoring	PTR3, PTR4
C3.1	Test the monitor to avoid false positives	PTR4
C4	Test the effects of not executing a service	PTR5, PTR6

Table 5.1: Primitive Test Requirements categorization.

Category 1 (C1) makes reference to the situations where the execution of the SUT satisfies the conditions specified in the guarantee term of the SLA so such term is evaluated as Fulfilled. From a monitoring point of view, these situations represent the expected behaviour of the SUT so they should be continuously exercised if no problem arises during the period of time the system is being observed. Test requirement PTR1 identified from the Guarantee Term is included in this category.

Category 2 (C2) represents those test requirements that involve a violation of any of the terms included in the SLA so test requirement PTR2 is included in this category. Even when an SLA violation arises, the application must deliver an expected behaviour

despite of any detected problem. Thus, the application will have to manage the violation according to the business values such as penalties specified in the SLA. Furthermore, the monitoring system must be able to detect the problem and report it in a proper way as well as evaluating the term as Violated. These situations are very interesting in both testing and monitoring approaches because their detection allows analyzing the information collected from the monitor and making a decision about any corrective action in order to solve the problem and avoid future consequences.

Category 3 (C3) includes those Primitive Test Requirements that represent executions where the services are invoked under circumstances that do not satisfy the Qualifying Condition so the terms become irrelevant and they must not be taken into account when evaluating the SLA. While monitoring, the systematic fulfilment of these requirements means that the application is continuously being executed under conditions that do not fulfil the QC so we do not have evidences about how the application would behave when the execution conditions change. Hence, they indicate the need of designing tests with the aim of checking whether the application is able to fulfil the GT in the future. Test requirements PTR3 and PTR4 are included in this category.

Within this category, there is a subcategory 3.1 (C3.1) of requirements that can be used to check the behaviour of the monitoring system that gathers information from the executions of the services and makes a decision about the evaluation of the SLA. More specifically, these requirements aim at checking that this monitor does not detect a false positive, that is to say, a violation in a term when such term is not relevant for the evaluation of the SLA. Test requirement PTR4 is included in this category. They represent situations where both the Qualifying Condition and the Service Level Objective are not satisfied so the monitoring system must be aware that this term is inapplicable and it cannot be evaluated as violated.

Category 4 (C4) includes those requirements where a service associated to a Guarantee Term is not executed so the term must be evaluated as Not Determined. The fulfilment of these requirements may represent a problem during the evaluation process because there is a lack of information to determine whether a term is being fulfilled or not. Due to this concern, these requirements are used to test whether the monitoring

system is able to perform the evaluation process properly even when a service (method) has not been executed. Furthermore, these tests may lead to detect problems not in the application but in the SLA specification itself so the agreement can be reviewed and updated accordingly. Test requirements PTR5 and PTR6 are included in this category.

5.3 Combination of Test Requirements

At this stage, we have already identified a set of Primitive Test Requirements from the specification of the SLA guarantee terms. In this section we address the generation of Combined Test Requirements by means of combining the previously identified Primitive TRs. We apply specific standard testing techniques to address such generation. This process of the combination of the Primitive TRs is designed to be carried out disregarding whether all the possible Primitive TRs have been identified or only a subset of them from any of the aforementioned categories.

First of all, we hierarchically represent the relevant information of the SLA in an adequate model using the Classification Tree Method (CTM) [53]. The resultant tree will later be used to derive the Combined Test Requirements. As we have outlined in Section 2.2, this method is expressed in terms of classifications and classes so, in the context of this SLA Guarantee Term testing level, such classifications and classes that will be used to build the tree need to be identified.

In order to build such tree, we use the hierarchy of the SLA specified in WS-Agreement from the most external All compositor to the Guarantee Terms. We build a node of the tree for each compositor and for each Guarantee Term. After that, we construct the leaves of the tree by means of representing the Primitive Test Requirements that have been identified according to Section 5.2 for each guarantee term. Consequently, the classifications of the tree represent the guarantee terms whereas the classes represent the Primitive TRs to be tested for each guarantee term (see Figure 5.5).

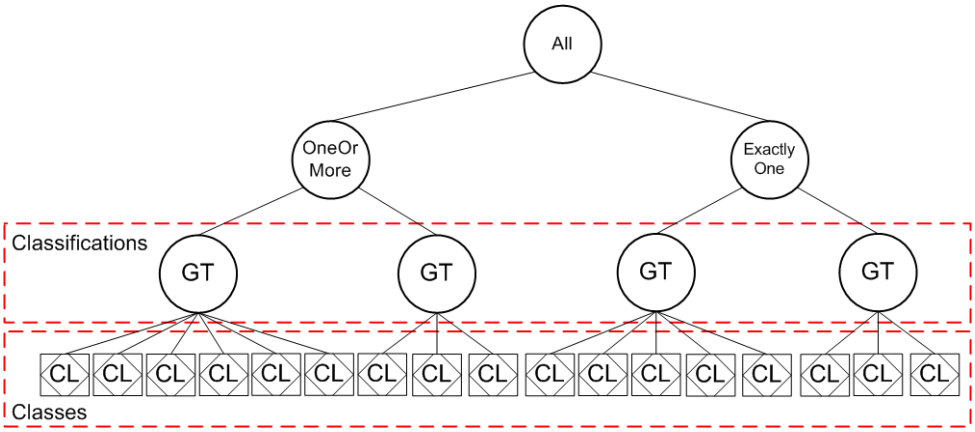


Figure 5.5: Structure of the Classification Tree.

According to this construction, we finally obtain a tree that contains one classification for each Guarantee Term specified in the SLA and each classification can have 6, 3 or 5 classes (represented in the leaves of the tree) depending on the application of the general and the particular cases, described in Section 5.2. With this approach, both the classifications and the classes fulfil the restriction of being disjoint partitions with respect to the SLA. Note also that in order to be consistent with the notation of the testing techniques described in the ISO/IEC 29119 [60], in the rest of this chapter we will use the concept of *class* (CL) when we refer to the different Primitive Test Requirements that arise from the evaluation of a Guarantee Term.

In Figure 5.6 we show an example of a tree constructed from the analysis of a WS-Agreement with three Guarantee Terms where no particular cases are applied to the first one, the particular case PC1 is applied to the Guarantee Term GT2 and the particular case PC2 is applied to the Guarantee Term GT3. The leaves that represent the classes are depicted with different colours depending on the evaluation value of the Guarantee Term during the exercitation of such class (green for Fulfilled, red for Violated, yellow for Inapplicable and grey for Not Determined).

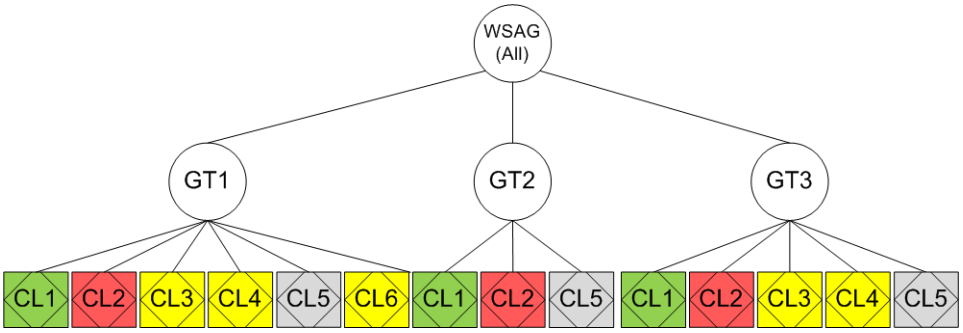


Figure 5.6: Example of a Classification Tree from an SLA.

5.3.1 Derivation of Combined TRs using Combinatorial Testing

Once we have constructed the resultant classification tree, we can make a decision about the parts of the tree that need to be tested with more thoroughness. To address this issue, we apply combinatorial testing [30] in order to derive the Combined Test Requirements for those tree parts that need to be tested. Furthermore, we have to consider that when deriving the Combined TRs, not all the combinations of classes will be used because we have to deal with the two aforementioned potential problems. The first one is related to the number of derived Combined TRS, which can be unmanageable if the SLA is complex and the second problem affects the testability of specific Combined TRs because there are combinations that lead to non-feasible situations to be tested.

To solve the first of these problems, we apply standard combinatorial testing techniques with the aim of obtaining a reduced (but significant) number of Combined TRs. To deal with the second problem, we define specific constraints that the Combined TRs have to satisfy to avoid generating non-feasible combinations of Primitive TRs.

5.3.2 Combinatorial Strategy

At this point and in order to derive the Combined TRs, we use different combinatorial testing techniques. These techniques are defined in terms of parameters and values. When testing the SLA based on the constructed tree, the parameters are the classifications that represent the Guarantee Terms and the values are the classes that represent the Primitive Test Requirements. Moreover, each Combined TR will contain one Primitive TR for each of the classifications of the tree.

After the identification of the parameters and their corresponding values, we derive the Combined TRs by means of applying any of the testing techniques standardized in the ISO/IEC 29119, which allow grading the intensity of the tests. These techniques are based on coverage and there are different coverage criteria that can be applied. The simplest coverage criterion is provided by *each choice testing* (also known as 1-wise) which requires that every class of every classification (Guarantee Term) must be exercised in at least one test case in the test suite. The most exhaustive coverage criterion is provided by a*ll combinations testing*, which requires that every possible combination of classes of all the classifications must be included in at least one test case. Between them, a widely used coverage criterion is provided by *pair-wise testing* (also known as all pairs or 2-wise). Pair-wise testing requires that every possible pair of classes of any two classifications represent the Combined TRs and they must be included in at least one test case.

In Figure 5.7 we show an example of the application of *each-choice testing* to the classification tree that was shown as example in Figure 5.6.

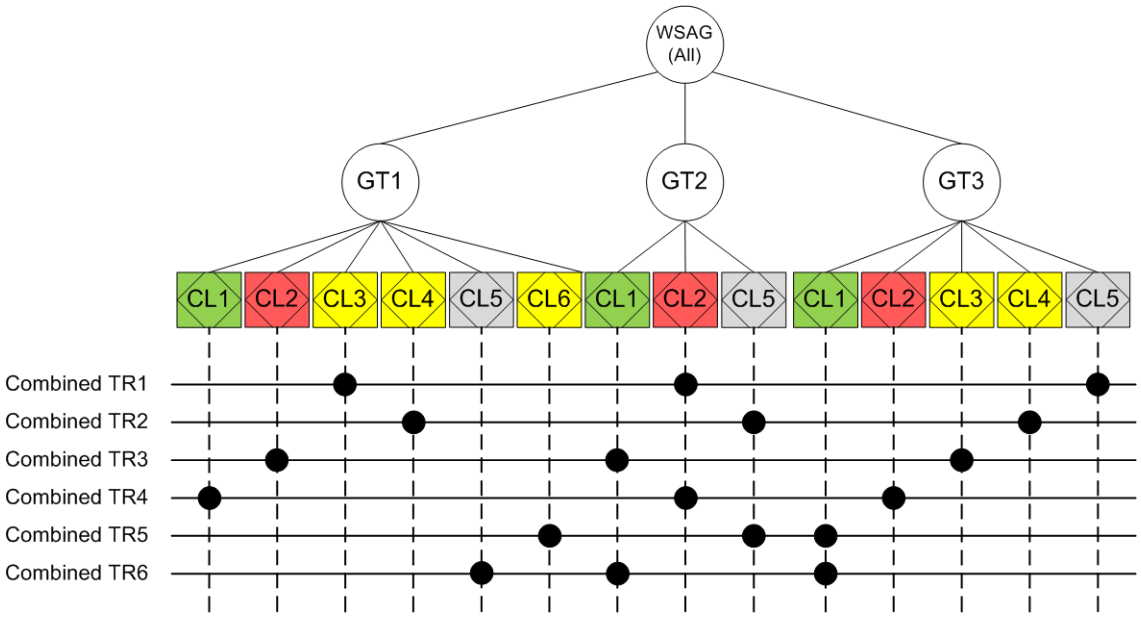


Figure 5.7: Example of the application of each-choice testing.

As can be seen, a total of 14 classes (Primitive Test Requirements) are identified for the three classifications (guarantee terms) of the tree. By means of applying each-choice testing, each of these classes is included in at least one Combined Test

Requirement. Furthermore, each Combined TR contains exactly three Primitive TRs, one for each of the guarantee terms of the SLA. For this example, it is enough to derive six Combined TRs in order to cover all the identified Primitive TRs.

In addition to existing testing techniques, we may define a strategy that guides the combinations depending on factors related to the content of the SLA and the behaviour of the SBA (e.g., critical SBA functionalities). This means that we may want to be more exhaustive and apply a combinatorial testing technique in a specific part of the tree (for example, a branch or a group of classifications) whereas a less exhaustive technique may be applied in a different part of the tree.

5.3.3 Definition of testability constraints

The derivation of the Combined Test Requirements may produce some combinations which do not make sense and lead to non-feasible test cases that cannot be executed. In this section we define specific constraints that allow excluding non-feasible Combined TRs.

We distinguish between two types of constraints: implicit and explicit. The implicit constraints are automatically obtained based on the information that is represented in the terms of the SLA. The explicit constraints are manually identified through the analysis of the SUT.

a) Implicit Constraints

Based on the syntax and semantics structure of WS-Agreement, we can identify a set of implicit constraints that help avoiding non-feasible combinations of classes used to derive the Combined TRs. These constraints are automatically obtained from the specification of the SLA.

We have defined the following set of implicit constraints for the general case where six classes are identified for each classification. If any of the two particular cases described in Section 5.2.2 has been applied to the involved classifications, these constraints must be suitably adapted.

Before discussing the constraints, let us assume that the selection of a class within a classification is represented by the function $ex\ (GT_x) = CLy$, which means that the class CLy of the classification GT_x is exercised.

I1: Guarantee Terms (GT) that affect the same method/service

Suppose that the method/service specified in the scope of the Guarantee Term GT1 is the same as the one specified in Guarantee Term GT2. If any of the classes CL5-CL6 of the classification that represents GT1 is selected to be combined in a Combined TR, then the method/service specified in the Scope of GT1 is not executed. Therefore one of the classes CL5-CL6 of the classification that represents GT2 must also be exercised. This constraint can be formally expressed as:

$$if\ ((\exists i,j \in [1,\mathrm{n}] : Scope(GT_i) = Scope(GT_j))\ \wedge$$
$$(ex(GT_i) \in \{CL5, CL6\}))\ \Rightarrow (ex(GT_j) \in \{CL5, CL6\})$$

I2: Guarantee Terms that have the same Qualifying Conditions

If a pair of Guarantee Terms shares the same Qualifying Condition and this is met, then all the classifications that represents these guarantee terms must take the values of the classes CL1, CL2 or CL5 but not CL3, CL4 or CL6. Likewise, if the Qualifying Condition is not met, then the classifications must take the values of the classes CL3, CL4 or CL6 but not CL1, CL2 or CL5.

$$if\ (\exists i,j \in [1,\mathrm{n}] : (QC(GT_i) = QC(GT_j))\ \wedge (ex(GT_i) \in \{CL1, CL2, CL5\}))$$
$$\Rightarrow (ex(GT_j) \in \{CL1, CL2, CL5\})$$
$$if\ (\exists i,j \in [1,\mathrm{n}] : (QC(GT_i) = QC(GT_j))\ \wedge (ex(GT_i) \in \{CL3, CL4, CL6\}))$$
$$\Rightarrow (ex(GT_j) \in \{CL3, CL4, CL6\})$$

I3: Guarantee Terms that have mutually disjoint Qualifying Conditions

If the Qualifying Condition of the first Guarantee Term is met then it is obvious that the Qualifying Condition of the second term must not be met and vice versa.

$$if\ (\exists i,j \in [1,\mathrm{n}] : (QC(GT_i) = !\,QC(GT_j))\ \wedge$$
$$(ex(GT_i) \in \{CL1, CL2, CL5\}))\ \Rightarrow (ex(GT_j) \notin \{CL1, CL2, CL5\})$$

$$if\ (\exists i,j \in [1,\mathrm{n}] : (QC(GT_i) = !\,QC(GT_j))\ \wedge$$
$$(ex(GT_i) \in \{CL3, CL4, CL6\}))\ \Rightarrow (ex(GT_j) \notin \{CL3, CL4, CL6\})$$

b) Explicit Constraints

In order to identify explicit constraints, an analysis of the business logic of the SUT must be carried out. These constraints refer to some specific situations concerning the possible behaviour of the SUT with regards to the ability to execute particular combinations of service methods, and affect the evaluation of the Guarantee Terms involved in the corresponding execution.

The set of explicit constraints includes the following:

E1: The execution of a method/service implies the non-execution of another method/service.

It means that if a method/service S_i (specified in the Scope of GT_i) is executed then the method/service S_j (specified in the Scope of GT_j) cannot be invoked or, formally:

$$if\ (\exists i,j \in [1,\mathrm{n}] : (ex(GT_i) \in \{CL1, CL2, CL3, CL4\}))$$
$$\Rightarrow (ex(GT_j)\ \in \{CL5, CL6\})$$

E2: The non-execution of a method/service implies the non-execution of another method/service

It means that if a method/service S_i (specified in the Scope of GT_i) is not executed then the method/service S_j (specified in the Scope of GT_j) cannot be invoked:

$$if\ (\exists i,j \in [1,\mathrm{n}] : (ex(GT_i) \in \{CL5, CL6\}))$$
$$\Rightarrow (ex(GT_j)\ \in \{CL5, CL6\})$$

E3: The execution of a method/service implies the execution of another method/service

It means that if a method/service S_i (specified in the Scope of GT_i) is executed then the method / service S_j (specified in the Scope of GT_j) must be invoked:

$$if\ (\exists i, j \in [1, \mathrm{n}] : (ex(GT_i) \in \{CL1, CL2, CL3, CL4\}))$$
$$\Rightarrow (ex(GT_j)\ \in \{CL1, CL2, CL3, CL4\})$$

E4: The non-execution of a method/service implies the execution of another method/service

It means that if a method / service S_i (specified in the Scope of GT_i) is not executed then the method / service S_j (specified in the Scope of GT_j) must be invoked:

$$if\ (\exists i, j \in [1, \mathrm{n}] : (ex(GT_i) \in \{CL5, CL6\}))$$
$$\Rightarrow (ex(GT_j)\ \in \{CL1, CL2, CL3, CL4\})$$

E5: The execution of a method/service is required

It means that a method / service S_i (specified in the Scope of GT_i) is mandatory to be invoked during the execution of the SUT:

$$(\exists i \in [1, \mathrm{n}] :\ ex(GT_i) \notin \{CL5, CL6\})$$

E6: Additional constraints

Depending on the content of QCs or SLOs, the use of a specific Primitive Test Requirement of GT (GT_i) may require also the use of a specific Primitive TR for another GT (GT_j). The specification of this rule (E6) depends on the information of the Guarantee Terms. For example, consider the following two guarantee terms (left part of Figure 5.8) and a subset of the identified classes (right part of Figure 5.8).

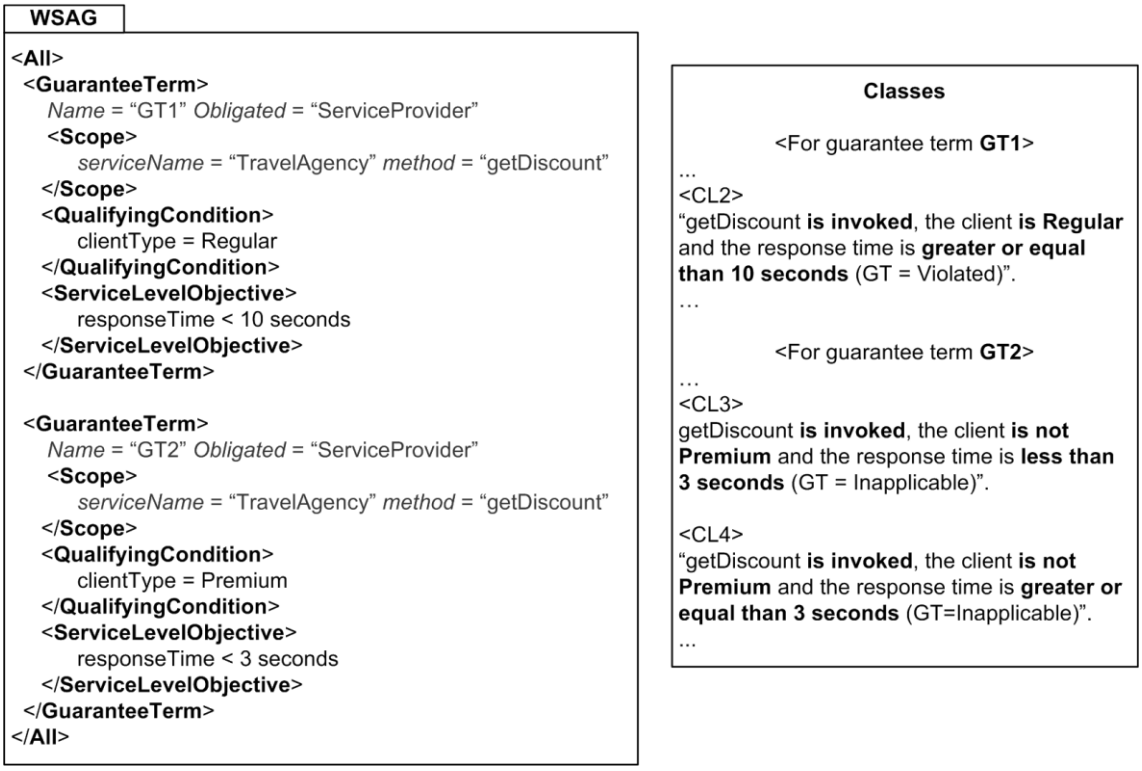


Figure 5.8: Excerpt of SLA Guarantee Terms and identified classes.

In this example, if GT1 is violated (exercising CL2) then GT2 must be evaluated as Inapplicable because the Qualifying Condition (client =Premium) is not met. In this case, the class CL4 must be exercised (note that CL3 could not be exercised because the response time forced by CL2 of GT1 is more than 10 seconds so the Service Level Objective of GT2 would never be met).

5.4 Derivation of test cases

Once we have identified the Primitive Test Requirements, the derivation of the Combined Test Requirements leads to the generation of test cases. We have previously said that a Combined TR contains one Primitive TR for each of the guarantee terms specified in the SLA. This means that, after exercising a Combined TR, all the guarantee terms of the SLA can be evaluated and, consequently, the SLA is also evaluated.

In this context, a Combined TR may already represent a complete scenario related to the SUT. Hence, a test case will usually cover one Combined TR although it is also possible to design test cases that sequentially exercise more than one Combined TR.

In addition to this information, it is necessary to have some knowledge about the behaviour of the SUT in order to specify the test case steps that exercise the Combined Test Requirements. For example, different sources of information can be used such as UML State Transition Diagrams or Sequence Diagrams.

5.5 Summary

In this chapter we have addressed the first of the testing levels defined in Section 3.2.2. From the specification of the individual guarantee terms of an SLA a set of Primitive Test Requirements are identified and organized in a Classification Tree. After that, these Primitive TRs are combined in order to derive the Combined Test Requirements by means of applying standard combinatorial testing techniques. Furthermore, the obtaining of non-feasible Primitive TRs as well as non-feasible combinations of such requirements in Combined TRs has been avoided through the definition of specific constraints regarding the specification of the SLA and the behaviour of the SBA. These Combined TRs are exercised through the design and execution of the test cases.

In the next Chapter 6 we will focus on the second of the testing levels, taking the logical relationships of the guarantee terms into account in order to design the tests.

In addition to this, in Chapter 7 we will present SLACT (SLA Combinatorial Testing), a tool we have developed to automate the tasks described in this chapter.

Chapter 6

Compositor Testing Level

Science is made up of mistakes, which in turn are
the steps towards the truth.

Jules Verne, 1828-1905
French novelist, poet and playwright

This chapter also addresses the generation of tests but, in this case, the logical relationships between the SLA guarantee terms are also considered. In order to identify the test requirements, SLACDC (SLA Condition Decision Coverage) criterion is presented. The generation of non-feasible tests is again avoided by means of the definition of specific rules.

6.1 Introduction

In Chapter 3 we presented SLATF, a framework that allows testing Service-Based Applications by means of analyzing the specification of the associated Service Level Agreement. In Chapter 4 we dealt with the evaluation of the SLA, which is an orthogonal activity that affects the rest of the activities involved in the testing process developed in SLATF. Based on this logic, in Chapter 5 we took the content of the SLA guarantee terms into account in order to identify a set of Primitive Test Requirements. The combination of these Primitive TRs allows deriving Combined Test Requirements that are finally used to generate the test cases. In this chapter we propose a further step by considering the logical relationships between the SLA guarantee terms in order to identify new Combined TRs.

When addressing this process, two issues may arise again, as in the Guarantee Term Testing Level, during the identification of the Combined TRs:

1. The combinatorial explosion depending on the size of the SLA.

2. The presence of non-feasible test requirements due to inadequate combinations.

We tackle the first of these issues by defining SLACDC (SLA Condition / Decision Coverage), a test criterion that is based on the MCDC (Modified Condition / Decision Coverage) criterion [116]. SLACDC allows identifying a manageable set of Combined TRs by means of providing a linear increase in the number of test requirements when the number of guarantee terms becomes higher.

Regarding the second issue, we define a set of rules that allow avoiding the identification of non-feasible Combined TRs by analyzing the specification of the guarantee terms and their relationships. The application of these rules implies the modification of the non-feasible test requirements in order to obtain other test requirements that represent feasible situations to be tested.

Both the identification of the test requirements as well as the application of the aforementioned rules are fully automated. The details about such automation are described in Chapter 7 of this dissertation.

6.2 SLACDC Test Criterion

The process of testing SLA-aware SBAs can be addressed by identifying test requirements from the specification of the SLA using a criterion based on the principle of the Modified Condition / Decision Coverage (MCDC). This criterion allows obtaining a cost-effective set of test requirements, representing situations that are interesting to exercise regarding the SLA and the SBA.

In this section we describe SLACDC, a test criterion that aims at identifying test requirements by analyzing the logical relationships between the SLA guarantee terms. First of all, we outline how to identify the Primitive Test Requirements. After that, the test criterion is defined and we describe the algorithms to combine the Primitive TRs and obtain the Combined TRs. Finally we define the rules that allow avoiding the obtaining of non-feasible test requirements.

6.2.1 Identification of Primitive Test Requirements

In Chapter 4 we define the logic to evaluate the guarantee terms and the compositor elements of the SLA. According to this logic, we outlined that each of the aforementioned elements are evaluated with four potential evaluation values. Furthermore, in Chapter 5 we described how this logic can be used to identify a set of Primitive Test Requirements from the specification of the guarantee terms. In the Guarantee Term Testing Level described in such chapter, six Primitive TRs were identified taking the structure of each guarantee term into account. In the present chapter, we focus on the logical relationships of the guarantee terms so we decide to select one Primitive TR for each of the evaluation values that a guarantee term can take, this is four Primitive Test Requirements for guarantee term. Each Primitive TR exercises the situation where the associated guarantee term is evaluated with one of the four potential values (Figure 6.1). Our objective with this design decision is to obtain a reduced but cost-effective set of Combined TRs that involves the evaluation of each guarantee term with all its potential values.

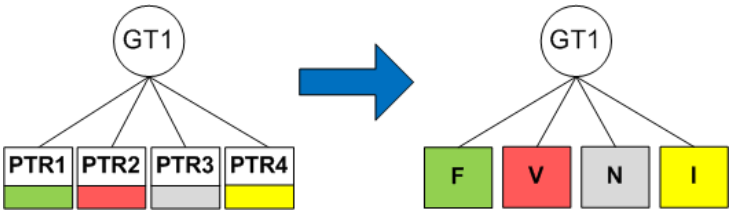


Figure 6.1: Relation Primitive Test Requirements – Evaluation values.

In such figure it can be seen that the Primitive Test Requirement PTR1 corresponds to the situation where the guarantee term is evaluated as Fulfilled, PTR2 is equivalent to Violated, PTR3 is equivalent to Not Determined and, finally, PTR4 is equivalent to Inapplicable. Hence, we will use the identifiers of the evaluation values (F, V, N, I) to represent the Primitive Test Requirements in order to make easier the comprehension of this chapter:

$$(ev(GT_1) = F) \equiv (ex(GT_1) = PTR1)$$
$$(ev(GT_1) = V) \equiv (ex(GT_1) = PTR2)$$
$$(ev(GT_1) = N) \equiv (ex(GT_1) = PTR3)$$
$$(ev(GT_1) = I) \equiv (ex(GT_1) = PTR4)$$

For example, when we say that the guarantee term GT_1 is evaluated as Violated, it means that the Primitive Test Requirement PTR2 of such guarantee term is exercised.

6.2.2 Four-valued MCDC Test Criterion

The Modified Condition Decision Coverage (MCDC), defined in the RTCA/DO-178B standard [116], is a broadly studied structural coverage criterion [137][27][63][64]. It has been used for test suite reduction and prioritization [65] because it provides a linear increase in the number of test requirements [38]. MCDC is a criterion that falls between condition/decision and multiple condition coverage [27]. This criterion has been shown to represent a good balance of test-set size and fault detecting ability [140][28] simultaneously. MCDC is defined as a conjunction of the following requirements:

- Every point of entry and exit in the program has been invoked at least once.
- Every condition in a decision in the program has taken all possible outcomes at least once.

- Every decision in the program has taken all possible outcomes at least once.
- Each condition in a decision has been shown to independently affect the decision's outcome (a condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions).

For example, consider the decision d = (a AND b) where a, b are two boolean conditions. To satisfy MCDC criterion, we need to generate three test cases (0,1) (1,1) (1,0) as described in Figure 6.2.

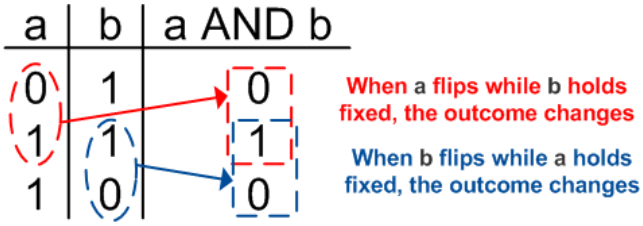


Figure 6.2: Example of application of MCDC with two evaluation values.

The MCDC criterion is usually defined for a binary logic. However, the application of MCDC when the logic allows four different evaluation values is more complex. So, in our approach:

***Definition 3**: A set of Combined Test Requirements satisfies the SLACDC (SLA Condition / Decision Coverage) criterion for a set of Guarantee Terms grouped within a compositor using the four-valued logic if and only if it fulfils the following sub-criteria:*

SC1 *Every guarantee term has taken all possible evaluation values at least once.*

SC2 *The compositor has taken all possible evaluation values as outcome at least once.*

SC3 *For each possible evaluation value of a guarantee term, a variation from a specific evaluation value to a different value has been shown to independently affect the evaluation of the compositor (this is, when we*

switch the evaluation value of the guarantee term while holding fixed the evaluation values of the rest of terms, the outcome of the evaluation of the compositor varies).

As an example, consider an *All* compositor element with two guarantee terms (GT1 and GT2) represented in Figure 6.3.

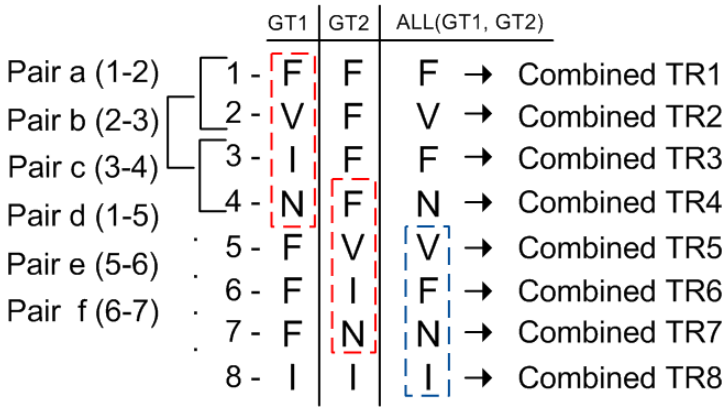


Figure 6.3: Example of application of SLACDC.

To address the identification of the Combined Test Requirements in the example, we start from the situation where both guarantee terms are evaluated as Fulfilled and, thus, the *All* compositor is also evaluated as Fulfilled (row 1 in the figure). Then, we set the second row obtaining the first pair (a), which allows us to switch the evaluation value of GT1 from Fulfilled to Violated and this change affects the evaluation value of the compositor, which also changes from Fulfilled to Violated. After this, we set the third row obtaining a new pair (b), where the evaluation value of GT1 switches from Violated to Inapplicable and, consequently, the evaluation value of the *All* compositor changes from Violated to Fulfilled.

This process continues until we obtain six different pairs (*a* to *f*) that fulfil the sub-criteria SC1 and SC3 of SLACDC criterion (Definition 3). At this stage, sub-criterion SC2 is not fulfilled because the *All* compositor has not been evaluated as Inapplicable yet. In order to satisfy sub-criterion SC2 we identify a new Combined TR (row 8) where both guarantee terms are evaluated as Inapplicable and, thus, the *All* compositor also takes the Inapplicable evaluation value. Hence, a final set of 8 Combined Test Requirements (CTR1-CTR8) is obtained (Figure 6.3) that satisfy the

criterion instead of the 16 test requirements that would be obtained using a complete combination (two guarantee terms with four evaluation values = 4^2).

6.2.3 Generation of Combined Test Requirements

In this section we present in detail the algorithms that are necessary to automate the obtaining of the Combined Test Requirements regarding the logical combinations of terms expressed by means of the compositor elements. For each compositor, we define the algorithm that obtains the Combined TRs and we illustrate the process with examples.

6.2.3.1 All Compositor

While testing the conditions specified in an *All* compositor, we check how the variation of a guarantee term evaluation affects the evaluation of the compositor while the rest of guarantee terms are evaluated as fulfilled. Hence, the algorithm to obtain the set of Combined TRs for an *All* compositor that groups n Guarantee Terms is as follows:

1. *Initialize the set with an initial Combined Test Requirement (CTR1) where all the guarantee terms are evaluated as Fulfilled.*

2. *For each GT_i in the All_Compositor:*

   Add a new Combined TR by means of switching the evaluation value of GT_i from Fulfilled (as it is in CTR1) to (Violated, Inapplicable, Not Determined) while the evaluation of GT_j with $j \neq i$ remains fixed to Fulfilled.

3. *Add a new Combined TR where all the guarantee terms are evaluated as Inapplicable in order to get the Inapplicable evaluation value in the All_Compositor.*

As an example, we partially illustrate the identification process of Combined TRs for an *All* compositor with 3 internal Guarantee Terms: ALL (GT1, GT2, GT3,).

Step1:

The set of test requirements is initialized with CTR1 where:

$$(ev(GT_1) = ev\ (GT_2) = ev(GT_3) = F)$$

Step2:

For each guarantee term we add three Combined TRs where the evaluation value of each guarantee term must be switched from Fulfilled to (Violated, Inapplicable, Not Determined) while holding the rest of terms fixed with Fulfilled:

$$a)\ (ev(GT_1) \in \{V, I, N\})\ \wedge\ (ev(GT_2) = ev(GT_3) = F)$$

$$b)\ (ev(GT_2) \in \{V, I, N\})\ \wedge\ (ev(GT_1) = ev(GT_3) = F)$$

$$c)\ (ev(GT_3) \in \{V, I, N\})\ \wedge\ (ev(GT_1) = ev(GT_2) = F)$$

The set of Combined TRs identified in this step is represented in Table 6.1 (CTR2-CTR10).

Step3:

We identify a new Combined TR where all the guarantee terms are evaluated with the Inapplicable value:

$$(ev(GT_1) =\ ev(GT_2) = ev(GT_3) = I)$$

The final set of Combined TRs identified for this compositor is represented in Table 6.1. The first column labels each Combined TR and the evaluations of the individual guarantee terms (GT) and the compositor are represented in the rest of the columns. The first row is remarked because it corresponds to the initial Combined TR and the cells that represent the guarantee terms that switch their evaluation values are grey shaded.

Test Req.	ev (GT_1)	ev (GT_2)	ev (GT_3)	ev (All)
CTR1	F	F	F	F
CTR2	V	F	F	V
CTR3	I	F	F	F
CTR4	N	F	F	N
CTR5	F	V	F	V
CTR6	F	I	F	F
CTR7	F	N	F	N
CTR8	F	F	V	V
CTR9	F	F	I	F
CTR10	F	F	N	N
CTR11	I	I	I	I

Table 6.1: Set of Combined TRs for an All compositor with three guarantee terms.

For example, the Combined Test Requirement CTR2 implies that the guarantee term GT1 is evaluated to Violated (exercising PTR2) whereas guarantee terms GT2 and GT3 are both evaluated to Fulfilled (exercising PTR1 for those terms). As we are dealing with an All compositor and one guarantee term has been violated, then the evaluation value of the compositor is also Violated.

The application of SLACDC criterion provides a linear number of combinations related to the number of conditions. In general, the number of combinations that satisfies MCDC for a binary logical decision is (n+1) where n is the number of conditions within the decision, there are two possible truth values (true/false) for each condition and the maximum number of combinations is 2*n [28]. In our case and dealing with a four-valued logic for the evaluation of the guarantee terms, the number of Combined TRs obtained with SLACDC criterion remains linear regarding the number of guarantee terms and evaluation values and can be obtained according to the following formula:

$$Num\_Test\_Req\_All = ((v-1)*n)+2 = 3n+2$$

Where n is the number of internal terms within the compositor and v the number of evaluation values of each guarantee term (in this case, v = 4). If we apply a complete combination using the four-valued logic, the number of obtained Combined Test Requirements would be 4^n.

6.2.3.2 OneOrMore Compositor

The algorithm to obtain the set of Combined Test Requirements from an *OneOrMore* compositor is similar to the one for *All* compositor, but in this case we want to exercise the variation of one term while the rest of guarantee terms have been violated. Thus, the algorithm for the identification of Combined TRs for an *OneOrMore* compositor is as follows:

1. *Initialize the set with an initial Combined Test Requirement (CTR1) where all the guarantee terms are evaluated as Violated.*

2. *For each GT_i in the OneOrMore_Compositor:*

   Add a new Combined TR by means of switching the evaluation value of GT_i from Violated (as it is in CTR1) to (Fulfilled, Inapplicable, Not Determined) while the evaluation of GT_j with $j \neq i$ remains fixed to Violated.

3. *Add a new Combined TR where all the guarantee terms are evaluated as Inapplicable in order to get the Inapplicable evaluation value in the OneOrMore_Compositor*.

We have omitted the explanation of the steps that perform the identification of test requirements for this compositor because the process is the same as for the All compositor. As an example, the test requirements identified for an OneOrMore compositor with 3 guarantee terms can be seen in Table 6.2.

Test Req.	ev (GT_1)	ev (GT_2)	ev (GT_3)	ev (OneOrMore)
CTR1	V	V	V	V
CTR2	F	V	V	F
CTR3	I	V	V	V
CTR4	N	V	V	N
CTR5	V	F	V	F
CTR6	V	I	V	V
CTR7	V	N	V	N
CTR8	V	V	F	F
CTR9	V	V	I	V
CTR10	V	V	N	N
CTR11	I	I	I	I

Table 6.2: Set of Combined TRs for an OneOrMore compositor with three guarantee terms.

The number of test requirements for an *OneOrMore* compositor is also given by the formula:

$$Num\_Test\_Req\_OneOrMore = ((v-1)*n)+2 = 3n+2$$

6.2.3.3 ExactlyOne Compositor

The identification of Combined Test Requirements from an *ExactlyOne* compositor varies a little regarding the two aforementioned algorithms for compositors *All* and *OneOrMore*. The reason is that two different scenarios need to be considered for this compositor:

1. Test the combinations where the evaluation value of the compositor varies due to the flip from none term evaluated as fulfilled to only one term fulfilled.

2. Test the combinations where the evaluation value of the compositor varies due to the flip from only one term evaluated as fulfilled to more than one term fulfilled.

The first scenario exercises the situation where all the guarantee terms are initially evaluated as Violated and we switch the evaluation value of each guarantee term to (Fulfilled, Inapplicable and Not Determined). Hence, it can be seen that this first scenario is exercised using the same set of Combined Test Requirements that we have described for the *OneOrMore* compositor. This means that the algorithm (A1) to test

this first scenario is the same and the Combined TRs obtained are represented in Table 6.3 (rows 1-11).

Row	Test Req.	ev (GT_1)	ev (GT_2)	ev (GT_3)	ev (ExOne)
1	CTR1	V	V	V	V
2	CTR2	F	V	V	F
3	CTR3	I	V	V	V
4	CTR4	N	V	V	N
5	CTR5	V	F	V	F
6	CTR6	V	I	V	V
7	CTR7	V	N	V	N
8	CTR8	V	V	F	F
9	CTR9	V	V	I	V
10	CTR10	V	V	N	N
11	CTR11	I	I	I	I
12	Duplicated (CTR5)	V	F	V	F
13	CTR12	F	F	V	V
14	CTR13	I	F	V	F
15	CTR14	N	F	V	N
16	Duplicated (CTR2)	F	V	V	F
17	Duplicated (CTR12)	F	F	V	V
18	CTR15	F	I	V	F
19	CTR16	F	N	V	N
20	Duplicated (CTR2)	F	V	V	F
21	CTR17	F	V	F	V
22	CTR18	F	V	I	F
23	CTR19	F	V	N	N
24	Duplicated (CTR11)	I	I	I	I

Table 6.3: Set of Combined TRs for an ExactlyOne compositor with three guarantee terms.

To exercise the second scenario, we have to obtain the Combined Test Requirements where there is already only one guarantee term evaluated as Fulfilled and we flip the evaluation of another guarantee term between the four possible evaluation values so the outcome of the compositor changes. The algorithm (A2) for the identification of these Combined TRs is as follows:

1. *Initialize an empty set of Combined Test Requirements.*

2. *For each GT_i in the ExactlyOne_Compositor:*

    a) *Add an initial Combined TR where one guarantee term GT_j with $j \neq i$ is evaluated as Fulfilled and the rest of guarantee terms are evaluated as Violated.*

    b) *Add a new Combined TR by means of switching the evaluation value of GT_i from Violated (as it is in the current initial test requirement) to (Fulfilled, Inapplicable, Not Determined) while the evaluation of GT_j with $j \neq i$ remains fixed to Fulfilled and the evaluation of the rest of the terms remains fixed to Violated.*

3. *Add a new Combined TR where all the guarantee terms are evaluated as Inapplicable in order to get the Inapplicable evaluation value in the ExactlyOne_Compositor.*

The Combined Test Requirements obtained with this algorithm (A2) are represented in Table 6.3 (rows 12-24). The cells that contain the initial Combined Test Requirement of step 2 for each guarantee term are remarked.

These two aforementioned scenarios may be tested independently and it is the tester who decides whether (s)he wants to exercise both scenarios or just one. In case the tester decides to test both scenarios, it is necessary to apply an additional step that involves the removal of duplicated Combined TRs that are identified for both algorithms (A1 and A2).

In Table 6.3 we have joined the set of Combined TRs obtained through the algorithm A1 and the algorithm A2 and we have marked the duplicated Combined TRs. In the first column we identify with a number all the Combined TRs obtained with both algorithms. In the second column we set an identifier to the final Combined TR or a brief description about the reason for removing such Combined TR. In the rest of column the evaluation values of the guarantee terms and compositor are represented. Furthermore, we have remarked the rows that represent the initial Combined TR in each

algorithm and those cells where the evaluation value of the guarantee term is switched (grey shaded).

After joining both sets and removing the duplicated Combined Test Requirements, a final number of 19 Combined TRs are identified. This number is obtained through the formula:

$$Num\_Test\_Req\_ExactlyOne = 6n + 1$$

Where n is the number of guarantee terms included in the *ExactlyOne* compositor. Thus, even applying these two algorithms to the compositor, we still provide a linear growth of Combined TRs regarding the number of guarantee terms included in such compositor.

6.2.4 Removing non-Feasible Test Requirements

The application of the aforementioned algorithms provides a set of Combined Test Requirements that satisfies the SLACDC criterion for the logical combinations of terms expressed by means of the compositors. However some of the identified Combined TRs correspond to situations that may be non-feasible to exercise due to the semantic information contained in the guarantee terms. Hence, we have to deal with these specific situations in order to refine the Combined TRs previously obtained. To address this improvement we define a set of rules that allow modifying the non-feasible Combined TRs and obtain other Combined TRs that represent feasible and interesting situations to be tested.

These rules are defined to keep fulfilling, as much as possible, the sub-criteria SC1 and SC2 of the criterion (Definition 3) whereas the sub-criterion SC3 will need to be relaxed. Thus, it cannot be assured that these three sub-criteria will finally be fulfilled in the resultant set of Combined Tests Requirements due to the dependencies between the conditions specified in the SLA.

The application of the rules involves identifying the Combined Test Requirements that are non-feasible in which certain evaluation values will be modified (a certain Primitive TR will be exercised instead of other) to obtain feasible Combined Test Requirements. This process requires that more than one evaluation value is switched

within the same Combined TR so SLACDC criterion is based on a specific form of MCDC named Masking MCDC, investigated by Chilenski [28], which allows more than one condition to vary at once ensuring that only the condition of interest influences the outcome.

6.2.4.1 Rule 1: Guarantee Terms without Qualifying Condition

This first rule is applied when some of the guarantee terms included in the compositor does not have Qualifying Condition. In this case, the Combined Test Requirements where such term is evaluated as Inapplicable (exercising the Primitive Test Requirement PTR4) must be removed. This means that:

$$Given\ a\ Compositor\ that\ contains\ \left(\bigcup_{i=1}^{n} GT_i\right)$$
$$if\ (\exists i \in [1,n] : \nexists QC(GT_i)\ \Rightarrow\ (ev(GT_i) \neq\ I)$$

In Figure 6.4 an example of the application of this rule over the Travel Agency scenario described in Section 4.1 and Section 5.2 is depicted.

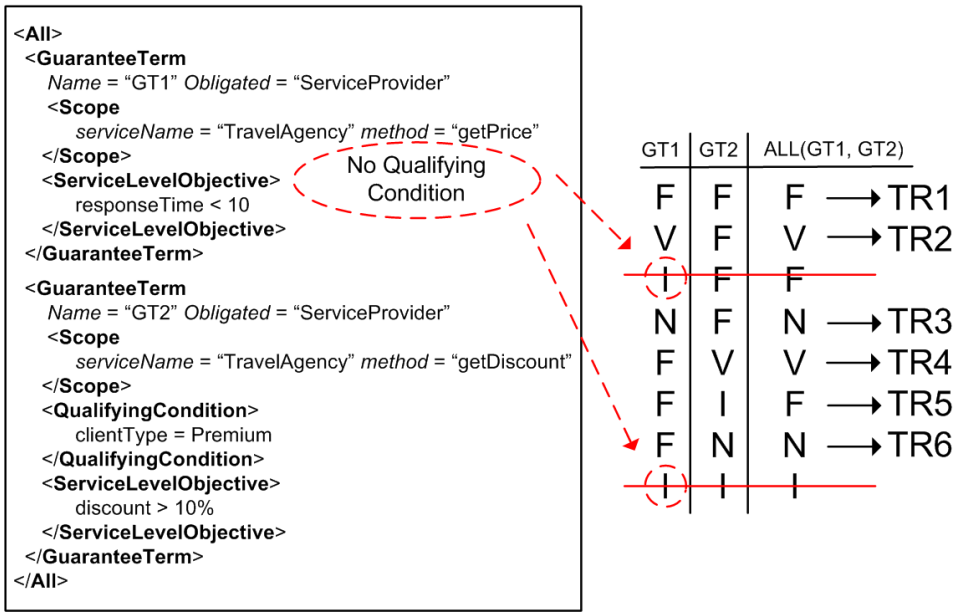


Figure 6.4: Example of application of Rule1: GTs without QC.

There is an *All* compositor with two internal guarantee terms. The first of them (GT1) does not have Qualifying Condition so the Combined Test Requirements where this term is evaluated as Inapplicable are removed. In the right part of the figure all the

guarantee terms obtained for the *All* compositor are represented. The Combined TRs where the current rule is applied are crossed out so we finally obtain a set of six Combined TRs instead of the original set of eight Combined TRs. In this case and due to the content of the SLA, the sub-criterion SC1 of SLACD is not satisfied because GT1 does not take the Inapplicable evaluation value. Furthermore, the sub-criterion SC2 is not satisfied either because the *All* compositor does not take the Inapplicable value.

6.2.4.2 Rule 2: Guarantee Terms with the same Scope

This second rule is applied when there are guarantee terms in a compositor that are related to the same method and service (Scope). In this case, the Combined Test Requirements that include these terms contain *coupled conditions* (in MCDC conditions that cannot be varied independently are said to be coupled [27]) or, in SLACDC criterion, better named as *coupled guarantee terms*. This implies that if one of these terms is evaluated as Not Determined (exercising PTR3 where the method/service is not invoked), then the other term must be evaluated as Not Determined (if the QC of the term is met) or Inapplicable (if the QC is no met).

This is:

$$Given\ a\ Compositor\ that\ contains\ \left(\bigcup_{i=1}^{n} GT_i\right)$$
$$if\ ((\exists i,j \in [1,n] : Scope(GT_i) = Scope(GT_j))$$
$$\wedge\ (ev(GT_i) = N)) \Longrightarrow (ev(GT_j) = N, I)$$

This means that, for example, if we have a compositor with two guarantee terms (GT1 and GT2) that affect the same method and service and GT1 has been evaluated as Not Determined, then the following Combined Test Requirements cannot be exercised:

(GT_1 = Not Determined and GT_2 = Fulfilled)

(GT_1 = Not Determined and GT_2 = Violated)

At this stage, if we have identified non-feasible Combined Test Requirements due to dependencies between the scopes of a pair of involved guarantee terms, we have to modify the evaluation value of one of these guarantee terms. The procedure we follow to change this value aims at keep fulfilling as much as possible the sub-criterion SC1 of

Definition 3, bearing in mind that sub-criteria SC2 and SC3 may be then relaxed. Then, the evaluation values Fulfilled / Violated will be the candidates to be modified because, by construction, they are much more common than the other value Not Determined.

According to this principle, we search the Combined TRs that contain pairs of guarantee terms affecting the same method and service. If one of the terms is evaluated as Not Determined and the other is not, we change the evaluation value of this last guarantee term to Not Determined. This process must be repeated for each pair of terms in a Combined TR that affect the same method and service. Furthermore, if the resultant Combined TR is already duplicated, it is removed.

To illustrate the application of this rule, we consider an example of an All compositor with three guarantee terms (GT1, GT2, GT3), all of them affecting the same method / service (represented in the left part of Figure 6.5).

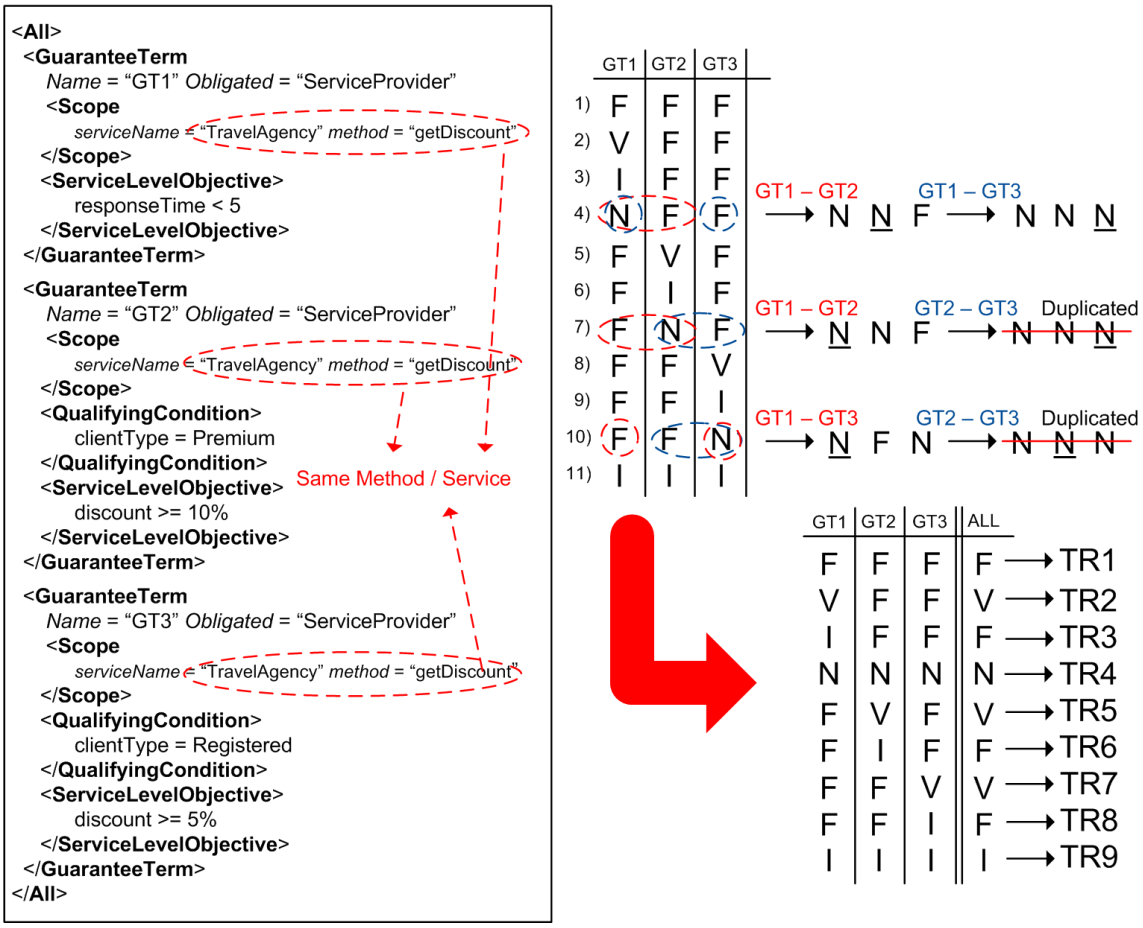


Figure 6.5: Example of application of Rule2: GTs with same Scope.

The set of Combined Test Requirements identified using the *All* compositor algorithm is represented in the first table within the top right part of the figure. From this requirements and applying this rule, we modify the specification of the Combined TRs 4, 7 and 8 in order to modify the non-feasible situations represented in such requirements. In the right part of the figure, we remark the involved guarantee terms in the modification, we underline the evaluation value that has been modified in each change and we cross out the removed test requirements for being duplicated. Finally, the resulting set of Combined Test Requirements is represented in the bottom right part of the figure. Despite of having modified the evaluation values in some Combined TRs, it is remarkable that, in this example, sub-criteria SC1 and SC2 of the criterion are still being fulfilled whereas sub-criterion SC3 has been relaxed for having switched more than one evaluation value in the same Combined Test Requirement.

6.2.4.3 Rule 3: Guarantee Terms that have exactly the same QC

This rule is applied when there are some terms within a compositor that specify exactly the same Qualifying Condition, which is a common situation in an SLA. If such Qualifying Condition is met, the guarantee terms can be evaluated as Fulfilled or Violated or Not Determined but never Inapplicable. If it is not met, the guarantee terms must be evaluated as Inapplicable (exercising PTR4). Hence, in this case we have again *coupled guarantee terms* and it does not make sense that some of these terms are evaluated as Inapplicable while the others are Fulfilled, Violated or Not Determined (exercising PTR1, PTR2 or PTR3). This is:

$$Given\ a\ Compositor\ that\ contains\ \left(\bigcup_{i=1}^{n} GT_i\right)$$
$$if\ (\exists i,j \in [1,n] : (QC(GT_i) = QC(GT_j))\ \wedge$$
$$(ev(GT_i) = I)) \Longrightarrow (ev(GT_j) = I)$$

This means that, for example, if a compositor contains two guarantee terms GT1 and GT2 that share the same Qualifying Condition, the following combinations cannot be exercised:

(GT_1 = Fulfilled and GT_2 = Inapplicable)

(GT_1 = Violated and GT_2 = Inapplicable)

(GT_1 = Not Determined and GT_2 = Inapplicable)

(GT_1 = Inapplicable and GT_2 = Fulfilled)

(GT_1 = Inapplicable and GT_2 = Violated)

(GT_1 = Inapplicable and GT_2 = Not Determined)

As we specified for the previous rule, we have to modify the Combined Test Requirements that contain these non-feasible combinations. Here again, we relax the sub-criterion SC3 of the SLACDC criterion but trying to respect sub-criteria SC1 and SC2 as much as possible.

To achieve this, we select the Combined TRs where this rule needs to be applied. As in the previous rule, the evaluation values Fulfilled and Violated are more usual than the Inapplicable so these are the values as well as Not Determined that will be modified to Inapplicable. Here again, this variation must be repeated for each pair of guarantee terms that contains the same Qualifying Condition within the compositor and resultant duplicated Combined Test Requirements should be removed.

To illustrate the application of this rule, we use an example of an *All* compositor with four guarantee terms (represented in the left part of the Figure 6.6) that affect different services.

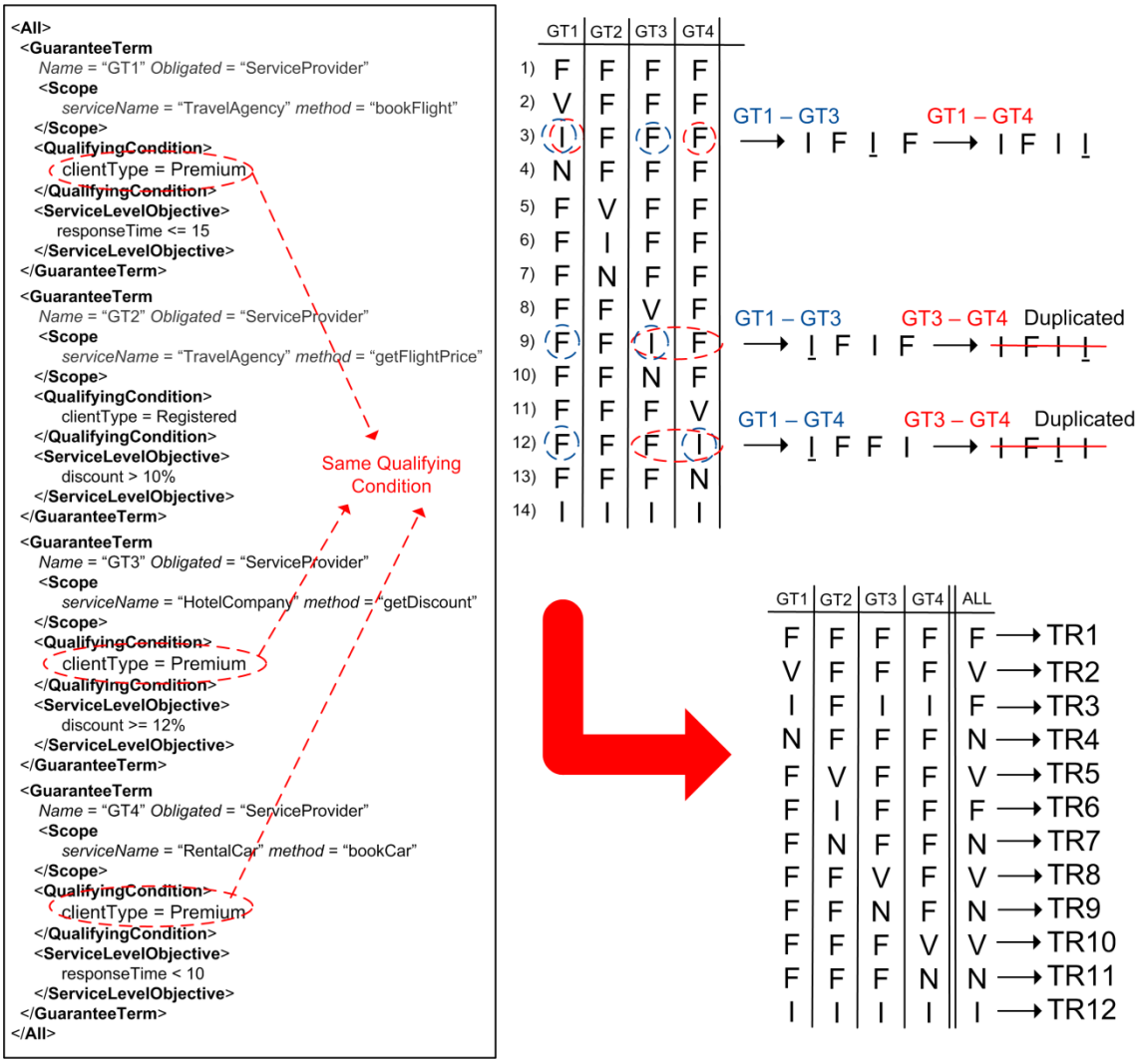


Figure 6.6: Example of application of Rule3: GTs with same QC.

Three of these terms (GT1, GT3 and GT4) specify the same condition in the Qualifying Condition element. Once we have identified the set of Combined Test Requirements by means of applying the algorithm for the *All* compositor, we have to select and modify those requirements that contain any of the non-feasible aforementioned combinations (Combined TRs 3, 9 and 12). In the right part of the figure, we perform the modifications, indicating the involved guarantee terms and crossing out the removed test requirements for being duplicated.

6.2.4.4 Rule 4: Guarantee Terms that have mutually disjoint QCs

This rule arises when, in a compositor, there are guarantee terms that contain Qualifying Conditions that are mutually disjoint. This means that, if the Qualifying

Condition of one term is met then the Qualifying Condition of the other term must not be met. Regarding the non-feasible Combined Test Requirements, if one of these terms is evaluated as Fulfilled or Violated (exercising PTR1 or PTR2) in a test requirement then the other one term must be evaluated as Inapplicable (PTR4). This is:

$$Given\ a\ Compositor\ that\ contains\ \left(\bigcup_{i=1}^{n} GT_i\right)$$
$$if\ ((\exists i,j \in [1,n] : QC(GT_i) = !\,QC(GT_j))\ \wedge$$
$$(ev(GT_i) \in \{F,V,N\}))\ \Longrightarrow\ ev(GT_j) \notin \{F,V,N\}$$

**** Note that in this context, the operator (!) does not mean that one Qualifying Condition is the opposite to the other. It really means that if the first QC is met then the second QC cannot be met.***

Thus and considering again the *coupled guarantee terms*, if we have a compositor element with two internal Guarantee Terms GT1 and GT2 that have mutually disjoint Qualifying Conditions, the following combinations do not make sense to be tested:

(GT_1 = Fulfilled and GT_2 = Fulfilled)

(GT_1 = Fulfilled and GT_2 = Violated)

(GT_1 = Fulfilled and GT_2 = Not Determined)

(GT_1 = Violated and GT_2 = Fulfilled)

(GT_1 = Violated and GT_2 = Violated)

(GT_1 = Violated and GT_2 = Not Determined)

(GT_1 = Not Determined and GT_2 = Fulfilled)

(GT_1 = Not Determined and GT_2 = Violated)

(GT_1 = Not Determined and GT_2 = Not Determined)

In order to avoid the appearance of these combinations in the final test suite, we have to modify the Combined Test Requirements that contain such combinations. The procedure is similar to the one performed in the previous pair of rules (Rule2 and

Rule3). In fact, this rule is practically the opposite as Rule3. Here again, we will change from the most common Fulfilled or Violated as well as Not Determined to the appropriate Inapplicable evaluation value.

According to this principle, we search the involved Combined Test Requirements. By construction, in each Combined TR there is a guarantee term whose evaluation value varied (named *pivot GT*) while the evaluation values of the other terms remained fixed. If the pair of terms that have mutually disjoint QC includes the pivot GT, then we always modify the evaluation value of the other term from Fulfilled, Violated or Not Determined to Inapplicable. On the other hand, if the pair of terms does not include the pivot GT, then we could modify the evaluation value of any of the two guarantee terms. As always, we have to repeat the process for each pair of terms that appear in the Combined TR and remove the Combined TRs that become duplicated.

In Figure 6.7 we show the application of this rule for an *All* compositor with two internal guarantee terms that present two mutually disjoint Qualifying Conditions in their specifications.

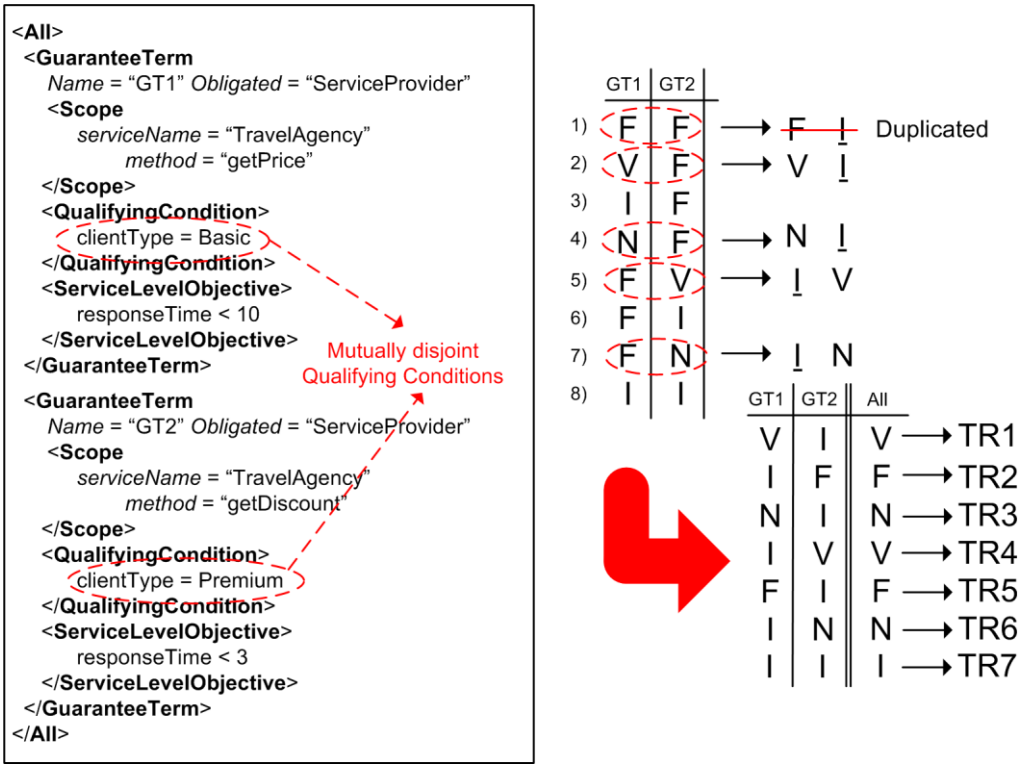


Figure 6.7: Example of application of Rule4: GTs with mutually disjoint QCs.

In the example, the first four Combined Test Requirements were obtained by holding fixed the value Fulfilled in the second guarantee term while switching the value of the first guarantee term (so the *pivot GT* is GT1). Hence, in Combined TRs 1, 2 and 4 we change the value of the second guarantee term from Fulfilled to Inapplicable as explained before. In test requirements 5 and 7 the *pivot GT* is GT2 so we modify the evaluation value Fulfilled of GT1 to Inapplicable.

6.2.5 Derivation of Test Cases

Once we have obtained the set of Combined Test Requirements from the compositor elements of the SLA, the next step involves the generation of a set of test cases that cover such Combined TRs. In this Compositor Testing Level, a Combined TR represents a partial situation related to the SUT. This means that the Combined TR exercises the specific conditions of the guarantee terms involved in the compositor. In this context, a complete scenario is designed by means of combining one Combined TR for each of the compositors specified in the SLA. This is equivalent to apply an each-choice testing strategy to the Combined TRs but bearing in mind that such Combined TRs have been properly elaborated before. Hence, a test case usually covers as many Combined TRs as the number of compositors in the SLA. Even in some occasions, a test case may exercise more than one complete scenario related to the SUT.

In addition to this, we have assured that in each Combined TR there are not non-feasible situations by applying the specific rules described in Section 6.2.4. However, this does not mean that any combination of the Combined TR in a test case makes sense. There may be Combined TRs that are incompatible to be combined within the same test case due to the specification of the SBA. As we mentioned in the Guarantee Term Testing Level, here again it may also be necessary to have some knowledge about the behaviour of the SBA in order to properly combine the test requirements and obtain an effective set of test cases.

6.3 Summary

In this chapter we have addressed the second of the testing levels defined in Section 3.2.2. We have focused on the logical relationships of the SLA guarantee terms in order to identify the test requirements. First of all, we have identified the Primitive

Test Requirements taking the potential evaluation values of a guarantee term into account. After that, we have defined SLACDC, a coverage-based criteria that allows obtaining a cost-effective set of Combined Test Requirements. These Combined TRs are later exercised through the design and execution of a set of test cases. This generation of test cases takes into account a set of rules that allow avoiding the derivation of non-feasible combinations of test requirements.

In Chapter 7 we will provide details about the level of automation of the tasks described in the present chapter.

In Chapter 8 we will apply the testing techniques presented in both Chapter 5 and Chapter 6 in a case study.

Chapter 7

Automation

In theory there is no difference between theory and practice.
In practice there is.

"Yogi" Berra
American former catcher

This chapter presents the automation of the testing techniques described in the previous chapters. It firstly introduces SLACT (SLA Combinatorial Testing), a tool that implements the identification of test requirements as well as their combinations to derive the test cases. After that, it also describes the SLACDC Generator, a prototype that automates the generation of tests by applying the aforementioned SLACD criterion.

7.1 Introduction

In the previous chapters we introduced the SLATF framework that allows testing SLA-aware service based applications and we defined a four-valued logic to evaluate the SLA based on the results of the executions. Furthermore, we addressed the generation of test requirements in the two testing levels described in Chapter 5 and Chapter 6. In the present chapter we provide the details about the automation of the different tasks presented in such chapters.

First of all we will introduce SLACT, a tool that allows analyzing the SLA, identifying the Primitive Test Requirements and the generation of the Combined Test Requirements in the Guarantee Term Testing Level. SLACT uses different combinatorial testing techniques in order to combine the Primitive TRs and derive the Combined TRs that lead to the test cases.

After that, we describe the level of automation of the Compositor Testing Level. We have developed a prototype that supports the generation of Combined Test Requirements from the logical relationships of the SLA terms. This prototype takes the SLACDC test criterion described in Chapter 6 into account in order to derive the test requirements.

It is worth mentioning that a proper combination of test requirements represents a complete scenario that can be tested. Bearing in mind that SLACT will allow obtaining only feasible combinations of test requirements, we consider that a tester or practitioner will be able to directly derive test cases from the aforementioned combinations provided by SLACT. Thus we will refer to the outcomes of SLACT as test cases along this chapter.

7.2 SLACT

SLACT (SLA Combinatorial Testing) [123] is a Java-based standalone application that implements the process of generating test cases by means of identifying the Primitive and the Combined Test Requirements from the specification of a Service Level Agreement, as described in the Guarantee Term Testing Level presented in Chapter 5. This generation includes the analysis of the SLA, the identification of

Primitive Test Requirements from the guarantee terms and the combinations of such requirements in order to derive the Combined Test Requirements. Each of these tasks involved in the generation of the test cases is therefore performed in an automatic way so the effort in terms of cost and time derived from the tester's work gets reduced significantly.

In this chapter the architecture of SLACT is presented and all the functionalities implemented within the components are described.

7.2.1 Architecture

The architecture of SLACT is depicted in Figure 7.1.

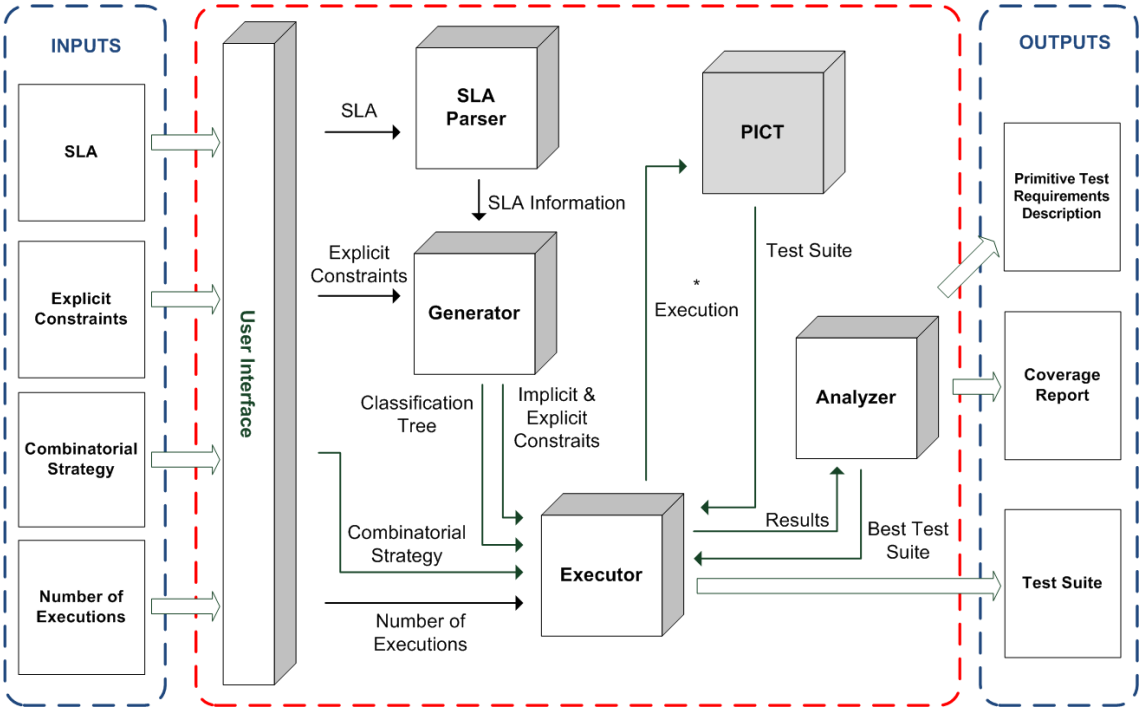


Figure 7.1: SLACT architecture.

The inputs are represented on the left of the figure whereas the outputs are represented on the right. In the centre of the figure we represent the following components that have been implemented in order automate the generation of the test requirements:

- Parser: receives the SLA as input and extracts the relevant information.

- Generator: analyzes the information extracted by the Parser, identifies the Primary Test Requirements and constructs the classification tree.

- Executor: obtains the sets of Combined Test Requirements by means of combining the Primary TRs identified by the Generator.

- Analyzer: analyzes the sets of Combined Test Requirements and obtains information related to the coverage of the Primary TRs. It also provides a description of such Primary TRs

- User Interface (UI): allows providing information that need to be taken into account in order to obtain the Combined Test Requirements.

Furthermore, SLACT makes use of an existing combinatorial testing tool developed by Microsoft, which is grey-coloured in the figure:

- PICT (Pairwise Independent Combinatorial Tool) [33][82]: performs the combinations of the Primary Test Requirements considering the specific combinatorial strategy selected by the tester.

In the following sections we present the language supported by SLACT as well as we define each of its components, describing in detail their inputs, the implemented functionalities and their outputs.

7.2.2 Syntax supported by SLACT

Currently, SLACT supports the use of SLAs written in the WS-Agreement standard language. WS-Agreement allows specifying the SLA using the guarantee terms that contain the Scope, the Qualifying Condition and the Service Level Objective. However, WS-Agreement leaves open the syntax for the specification of the conditions represented within such elements. Hence, the most usual way to express the conditions of the guarantee terms is by means of the definition of Domain Specific Languages (DSL), which are tailored to a specific application domain [79][133].

In our case, we have developed a particular DSL in order to specify the internal elements of the guarantee term. Hence, SLACT is able to parse the Scope, the

Qualifying Condition and the Service Level Objective of the guarantee terms when such elements are written using the DSL represented in Figure 7.2.

```
<wsag:GuaranteeTerm
      wsag:Name="name_of_the_guarante_term"
      wsag:Obligated="obligated_party">

      <wsag:ServiceScope
         wsag:ServiceName="name_of_the_service">
         <SLATest:Method>
           <NameMethod> name_of_the_method </NameMethod>
         </SLATest:Method>
      </wsag:ServiceScope>

      <wsag:QualifyingCondition>
         <SLATest:variable>
             name_of_the_variable
         </SLATest:variable>
         <SLATest:operator>
             possible_operators (eq | ne | gt | lt | gte | lte)
         </SLATest:operator>
         <SLATest:constant>
             constant_value
         </SLATest:constant>
      </wsag:QualifyingCondition>

... Rest of the Guarantee Term ...

</wsag:GuaranteeTerm>
```

Figure 7.2: DSL supported by SLACT.

7.2.3 How SLACT works

SLACT implements the aforementioned components in order to obtain the Combined Test Requirements. In this section, we describe how each component carries out its tasks, starting from the analysis of the information specified in the SLA and ending with the generation of the Combined TRs that cover the previously identified Primitive TRs.

7.2.3.1 Parser

This component receives the specification of the SLA in WS-Agreement as input. The parser is in charge of analyzing and extracting the relevant information contained in the SLA, including the compositor elements and their constituent guarantee terms.

With the information gathered from the agreement, SLACT constructs the first levels of the Classification Tree as described in Section 5.2, excluding the leaves of the tree, which represent the Primitive Test Requirements that will be later identified.

7.2.3.2 Generator

The first task implemented by the Generator is the identification of the Primitive Test Requirements from the specification of the guarantee terms of the SLA, as described in Section 5.2. To address this issue it uses the information previously obtained by the parser. The Generator takes the general case into account as well as the two particular cases. The first one (PC1) is automatically applied when the guarantee term does not present the Qualifying Condition. The second one (PC2) is required to be applied by the tester through the User Interface of SLACT. In Figure 7.3 we represent the area of the User Interface where the guarantee terms are showed and the application of the particular cases can be selected. In this figure, the information showed is related to the scenario of the Travel Agency we introduced in Section 4.1 of Chapter 4.

Guarantee Terms

Name	Service	Method	PC1	PC2
GT1	TravelAgency	bookFlight		✔
GT2	TravelAgency	getFlightPrice		✔
GT3	TravelAgency	bookHotel	✔	
GT4	TravelAgency	getDiscount		✔
GT5	TravelAgency	getDiscount		✔

Figure 7.3: Guarantee Terms and Particular Cases table.

The first column of the table shows the name of the guarantee term. The second and third columns represent the name of the service and method specified in the Scope, respectively. The forth column indicates the guarantee terms that are affected by the

first particular case (PC1). The last column allows selecting which guarantee terms should be affected by the second particular case (PC2).

After identifying the Primitive TRs, the Generator finishes the construction of the Classification Tree by representing such test requirements in the last level of the hierarchy of the tree. This classification tree is one of the outputs of the Generator.

Once the Primitive TRs are identified and the classification tree constructed, SLACT tries to avoid non-feasible combinations of the test requirements regarding the content of the SLA and the context of the service based application. To solve this problem, SLACT uses two types of constraints in order to guide the proper combination of the Primitive TRs, as described in Section 5.3.3:

- Implicit Constraints.
- Explicit constraints.

The specification of both implicit and explicit constraints is the second output of the Generator.

Implicit Constraint

The implicit constraints are automatically identified by the Generator taken the content of the guarantee terms into account. There are three different types of implicit constraints, as we described in Section 5.3.3 of this dissertation:

(**I1**): this constraint is identified when there is a set of terms that affect the same services, which are specified in the Scope element. When this happens, the Generator defines the rules using a specific syntax [33] that is appropriate for the Executor component. For example, let us consider two guarantee terms (GT1 and GT2) that satisfy the aforementioned condition, affecting the same services in the Scope. In this case, the Generator specifies the following rules:

```
                    #Terms : GT1, GT2
              ******************************
  IF [GT1] IN {"PTR5", "PTR6} THEN [GT2] IN {"PTR5", "PTR6"};
  IF [GT2] IN {"PTR5", "PTR6} THEN [GT1] IN {"PTR5", "PTR6"};
```

This pair of rules means that if the Primitive PTR5 or PTR6 of one of the guarantee terms are exercised, then any of the two Primitive TRs of the other guarantee term must also be exercised.

(**I2**): this constraint is identified when there is a set of terms that share the same Qualifying Condition. Again, we represent an example where three guarantee terms (GT1, GT2 and GT3) satisfy such conditions. In this case, the Generator defines the following pair of rules:

```
                    #Terms : GT1, GT2, GT3
                ********************************
  IF [GT1] IN {"PTR1", "PTR2", "PTR5"} THEN ([GT2] IN {"PTR1",
     "PTR2", "PTR5"} AND [GT3] IN {"PTR1", "PTR2", "PTR5"});
  IF [GT1] IN {"PTR3", "PTR4", "PTR6"} THEN ([GT2] IN {"PTR3",
     "PTR4", "PTR6"} AND [GT3] IN {"PTR3", "PTR4", "PTR6"});
                --------------------------------
  IF [GT2] IN {"PTR1", "PTR2", "PTR5"} THEN ([GT1] IN {"PTR1",
     "PTR2", "PTR5"} AND [GT3] IN {"PTR1", "PTR2", "PTR5"});
  IF [GT2] IN {"PTR3", "PTR4", "PTR6"} THEN ([GT1] IN {"PTR3",
     "PTR4", "PTR6"} AND [GT3] IN {"PTR3", "PTR4", "PTR6"});
                --------------------------------
  IF [GT3] IN {"PTR1", "PTR2", "PTR5"} THEN ([GT1] IN {"PTR1",
     "PTR2", "PTR5"} AND [GT2] IN {"PTR1", "PTR2", "PTR5"});
  IF [GT3] IN {"PTR3", "PTR4", "PTR6"} THEN ([GT1] IN {"PTR3",
     "PTR4", "PTR6"} AND [GT2] IN {"PTR3", "PTR4", "PTR6"});
```

The meaning of each pair of rules is as follows. If the Primitive PTR1 or PTR2 of one guarantee term are exercised then the other two guarantee terms must exercise the Primitive PTR1, PTR2 or PTR5. Likewise, if the Primitive PTR3 and PTR4 of one guarantee term are exercised then the other two guarantee terms must exercise the Primitive PTR3, PTR4 or PTR6.

(**I3**): this constraint is identified when there is a set of terms that present mutually disjoint Qualifying Conditions. For example, let us assume that there are a group of guarantee terms (GT1 and GT2) that have a Qualifying Condition that is mutually

disjoint regarding another group of terms (GT3, GT4 and GT5). The Generator will define the following pair of rules:

```
            #Terms : (GT1, GT2) vs (GT3, GT4, GT5)
               ********************************

   IF ([GT1] IN {"PTR1", "PTR2", "PTR5"} OR [GT2] IN {"PTR1",
"PTR2", "PTR5"}) THEN (NOT [GT3] IN {"PTR1", "PTR2", "PTR5"} AND
 NOT [GT4] IN {"PTR1", "PTR2", "PTR5"} AND NOT [GT5] IN {"PTR1",
                        "PTR2", "PTR5"});

   IF ([GT1] IN {"PTR3", "PTR4", "PTR6"} OR [GT2] IN {"PTR3",
"PTR4", "PTR6"}) THEN (NOT [GT3] IN {"PTR3", "PTR4", "PTR6"} AND
 NOT [GT4] IN {"PTR3", "PTR4", "PTR6"} AND NOT [GT5] IN {"PTR3",
                        "PTR4", "PTR6"});

               --------------------------------

   IF ([GT3] IN {"PTR1", "PTR2", "PTR5"} OR [GT4] IN {"PTR1",
 "PTR2", "PTR5"} OR [GT5] IN {"PTR1", "PTR2", "PTR5"}) THEN (NOT
   [GT1] IN {"PTR1", "PTR2", "PTR5"} AND NOT [GT2] IN {"PTR1",
                        "PTR2", "PTR5"});

   IF ([GT3] IN {"PTR3", "PTR4", "PTR6"} OR [GT4] IN {"PTR3",
 "PTR4", "PTR6"} OR [GT5] IN {"PTR3", "PTR4", "PTR6"}) THEN (NOT
   [GT1] IN {"PTR3", "PTR4", "PTR6"} AND NOT [GT2] IN {"PTR3",
                        "PTR4", "PTR6"});
```

The first pair of rules has the following meaning. If any of the guarantee terms of the first group exercises the Primitive PTR1, PTR2 or PTR5 then the guarantee terms of the second group must not exercise Primitive PTR1, PTR2 or PTR5. Likewise, if any of the guarantee terms of the first group exercise the Primitive PTR3, PTR4 or PTR6 then the guarantee terms of the second group must not exercise Primitive PTR3, PTR4 or PTR6.

The second pair of rules has the same meaning but changing the order of the groups of guarantee terms.

The identification of these constraints is automatically performed by SLACT. Furthermore and after that, the constraints are displayed in the User Interface and SLACT provides the opportunity to remove any of the identified constraints (Figure 7.4). The first column of the corresponding table numbers the rules. The second column represents the identifier of the type of implicit constraint. Finally, the third column shows the guarantee terms that are affected by the constraint.

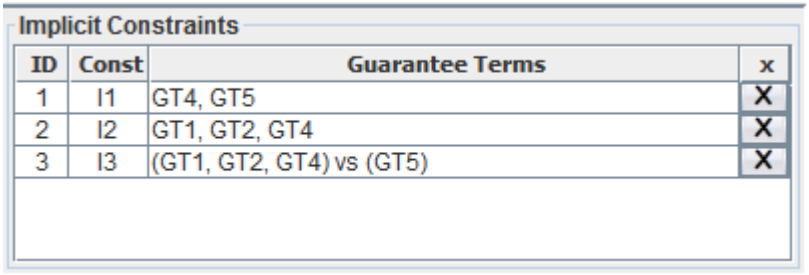


Figure 7.4: Implicit Constraints table.

Explicit Constraint

The explicit constraints are specified by the tester through the User Interface of SLACT (Figure 7.5). The potential explicit constraints to be defined are described in Section 5.3.3 of this dissertation. It is also possible to load the explicit constraints from a file or even to save the specified constraints to a file for their future use.

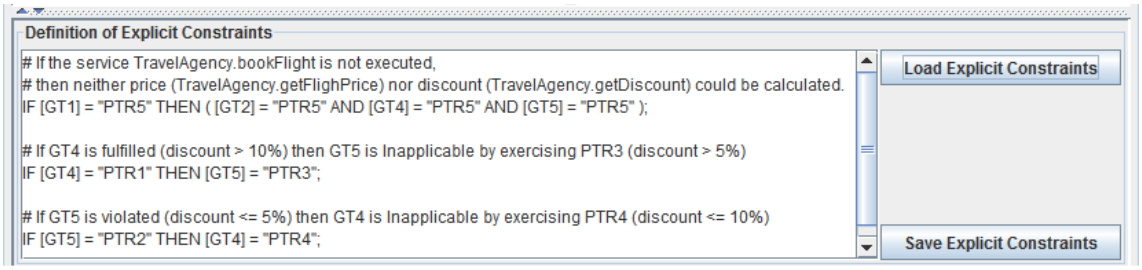


Figure 7.5: Explicit Constraints section.

7.2.3.3 Executor

The Executor is in charge of performing the combination of the Primitive Test Requirements in order to derive the Combined Test Requirements and obtain the test cases. To do this, it takes the constraints identified by the Generator into account. This component makes use of the Pairwise Independent Combinatorial Tool (PICT) [33][82], which is a free tool developed by Microsoft whose core generation algorithm is a greedy heuristics optimized for speed.

As we have outlined in Section 5.3, the combination of the Primitive Test Requirements is a very laborious task in the sense that the number of potential test requirements could be really high if the SLA is quite complex. To address this issue, in this dissertation we use standard combinatorial testing tools, broadly studied in the

scope of software testing [30]. Therefore SLACT allows the user to decide the combinatorial strategy through the User Interface (Figure 7.6), by means of giving the possibility to select between a weaker or stronger strategy. The weakest strategy would be selected by applying *each-choice testing* whereas the strongest strategy would be selected by applying *all combinations* (this is, selecting *base choice testing* and setting the General Combination Order with the value of the total number of guarantee terms).

Figure 7.6: Selection of the combinatorial strategy.

In the left part of the figure the four potential strategies are depicted. If the hybrid strategy is selected, in the middle part of this section it is possible to select the general order of combination and the individual strategy for specific groups of guarantee terms. The details of the selected hybrid strategy are showed in the table of the right part of the figure. In such example, a pair-wise strategy is used with the guarantee terms GT1 and GT4 whereas a 3-wise strategy is used with the guarantee terms GT1, GT3 and GT5. The rest of the combinations are performed using an each-choice testing (1-wise) as it is specified in the General Combination Order option.

The combinations of test requirements carried out by SLACT are based on the greedy algorithm of PICT. This means that, in different executions, the results of the combinations obtained by the tool may be different. Because of this, SLACT provides two different ways to be run:

- Performing multiple executions in order to obtain different sets of Combined Test Requirements and selecting the optimum among them (the one that achieves the selected coverage with the least number of Combined TRs). In this case, the number of executions is an input provided by the tester through the User Interface (Figure 7.7).

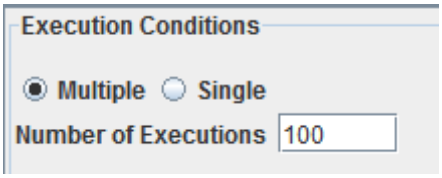


Figure 7.7: Selection of multiple executions.

- Performing a single execution in order to obtain one set of Combined TRs that satisfies the selected coverage. The type of this execution is selected by the tester through the User Interface (Figure 7.8), by deciding between deterministic or random.

Figure 7.8: Selection of single execution.

The output of the Executor is the specification of the test suite that contains the Combined Test Requirements (Figure 7.9). This specification may be obtained in a file with a csv / txt format. In the first column the identifier of the test case is represented. In the rest of the column the Primitive TRs exercised of each guarantee term are showed.

	A	B	C	D	E	F	G	H
1	Test Suite Specification							
2								
3	Each row represents a test case that exercises a Combined TR and							
4	each column represents the exercised Primitive Test Requirement for each involved Guarantee Term.							
5								
6		GT1	GT2	GT3	GT4	GT5		
7	Test Case TC1	PTR3	PTR3	PTR1	PTR3	PTR2		
8	Test Case TC2	PTR5	PTR1	PTR2	PTR1	PTR3		
9	Test Case TC3	PTR4	PTR4	PTR5	PTR4	PTR1		
10	Test Case TC4	PTR2	PTR2	PTR5	PTR2	PTR4		
11	Test Case TC5	PTR1	PTR5	PTR2	PTR5	PTR5		
12								

Figure 7.9: Specification of the Test Suite.

7.2.3.4 Analyzer

The Analyzer is in charge of two main different issues. On the one hand, it receives the results gathered from the multiple executions of the Executor and determines which test suite provides the selected coverage with the least number of Combined Test Requirements. On the other hand, it analyses the Combined TRs with the aim at assuring that all of the Primitive TRs have been covered.

The outputs of the Analyzer are:

1. A report that contains the statistics regarding the coverage of each Primitive TR in the resultant set of test cases (Figure 7.10). In such report, it is showed the number of test cases where each Primitive TR is exercised.

```
Number of Guarantee Terms: 5
Number of Combined Test Requirements 24

Number of times that each Primitive PTR is exercised in the test suite.
GT1 -> PTR1 = 4 times.
GT1 -> PTR2 = 4 times.
GT1 -> PTR3 = 4 times.
GT1 -> PTR4 = 4 times.
GT1 -> PTR5 = 8 times.

GT2 -> PTR1 = 5 times.
GT2 -> PTR2 = 4 times.
GT2 -> PTR3 = 4 times.
GT2 -> PTR4 = 4 times.
GT2 -> PTR5 = 7 times.

GT3 -> PTR1 = 8 times.
GT3 -> PTR2 = 7 times.
GT3 -> PTR5 = 9 times.

GT4 -> PTR1 = 5 times.
GT4 -> PTR2 = 4 times.
GT4 -> PTR3 = 4 times.
GT4 -> PTR4 = 3 times.
GT4 -> PTR5 = 8 times.

GT5 -> PTR1 = 4 times.
GT5 -> PTR2 = 3 times.
GT5 -> PTR3 = 5 times.
GT5 -> PTR4 = 4 times.
GT5 -> PTR5 = 8 times.
```

Figure 7.10: Report of statistics about coverage.

2. A detailed description of each Primitive TR identified from the guarantee terms of the SLA (Figure 7.11). This description shows the action to be executed in order to exercise each Primitive TR.

```
1  --------------------
2  Guarantee Term: GT1
3  --------------------
4  PTR1:   The service TravelAgency.bookFlight is executed, the Qualying Condition (clientType eq Premium)
   is fulfilled, and the Service Level Objective (responseTime lt 300) is fulfilled.
5
6  PTR2:   The service TravelAgency.bookFlight is executed, the Qualying Condition (clientType eq Premium)
   is fulfilled, and the Service Level Objective (responseTime lt 300) is violated.
7
8  PTR3:   The service TravelAgency.bookFlight is executed, the Qualying Condition (clientType eq Premium)
   is not fulfilled, and the Service Level Objective (responseTime lt 300) is fulfilled.
9
10 PTR4:   The service TravelAgency.bookFlight is executed, the Qualying Condition (clientType eq Premium)
   is not fulfilled, and the Service Level Objective (responseTime lt 300) is violated.
11
12 PTR5:   The service TravelAgency.bookFlight is not executed.
13
14 --------------------
15 Guarantee Term: GT2
16 --------------------
17 PTR1:   The service TravelAgency.getFlightPrice is executed, the Qualying Condition (clientType eq
   Premium) is fulfilled, and the Service Level Objective (responseTime lt 30) is fulfilled.
18
19 PTR2:   The service TravelAgency.getFlightPrice is executed, the Qualying Condition (clientType eq
   Premium) is fulfilled, and the Service Level Objective (responseTime lt 30) is violated.
20
21 PTR3:   The service TravelAgency.getFlightPrice is executed, the Qualying Condition (clientType eq
   Premium) is not fulfilled, and the Service Level Objective (responseTime lt 30) is fulfilled.
```

Figure 7.11: Description of the Primitive Test Requirements.

With the specification of a Combined Test Requirement, the description of the involved Primitive TRs in such Combined TR as well as some knowledge about the behaviour of the service-based application, a test case that exercises the specific scenario represented by the Combined TR is obtained.

7.2.3.5 User Interface

The User Interface (UI) of SLACT is depicted in Figure 7.12 and allows the tester to select the XML file that contains the SLA to be analyzed.

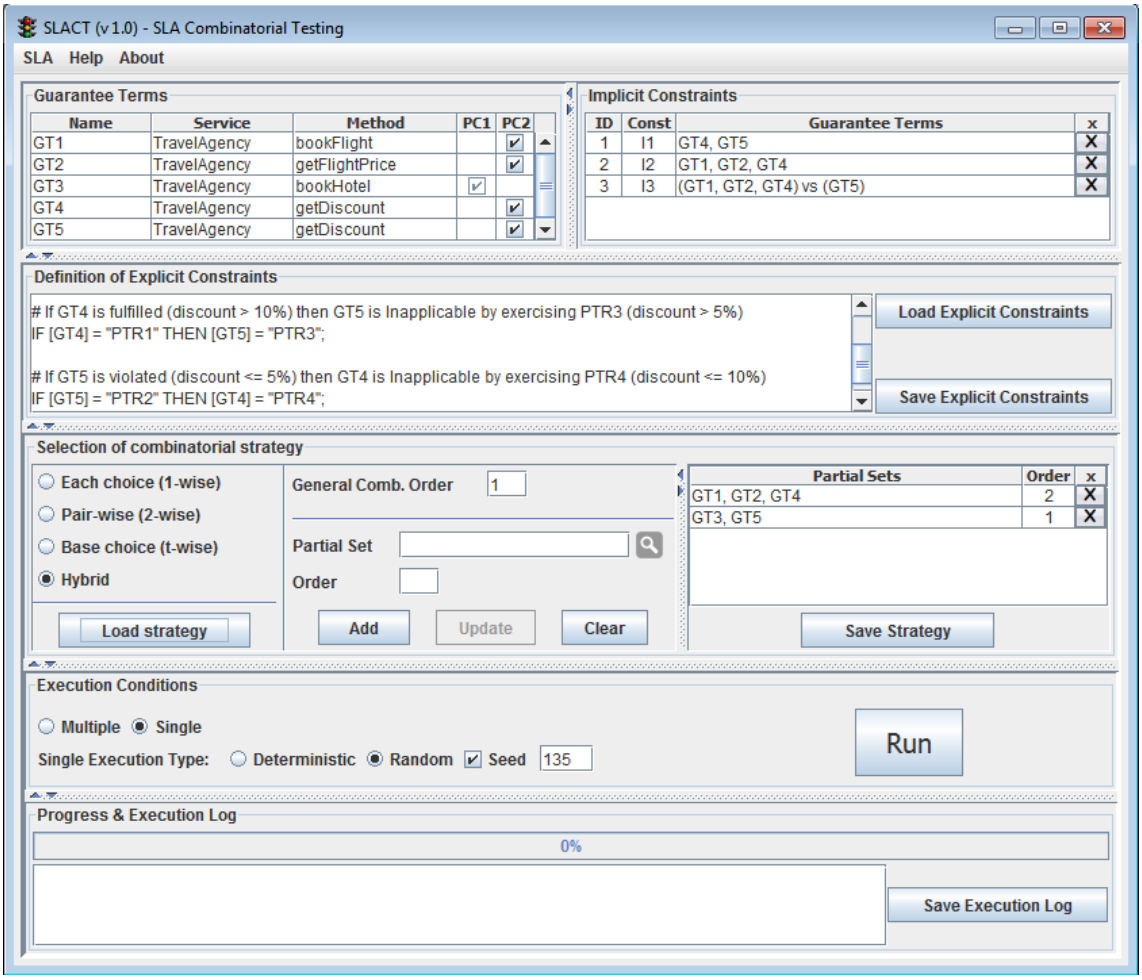


Figure 7.12: SLACT User Interface (UI).

In the top section of the interface, the information obtained by the Parser is showed and it allows deciding whether the particular case PC2 must be applied during the identification of the Primitive Test Requirements. The user interface also shows the obtained implicit constraints and allows removing any constraints if necessary.

In the following section the explicit constraints can be manually specified as well as loaded from a file. Furthermore, the specification of such constraints can be saved to a file.

After that, it is possible to select the combinatorial strategy by means of deciding the testing technique to be applied. If the hybrid strategy is selected, the details of such strategy can be specified or loaded from a file. After the complete definition of the hybrid strategy, this can be saved to a file.

The last section requires deciding whether SLACT needs to be executed several times or just one. If the multiple execution option is selected, the number of times must be specified. If the tester selects the single option, (s)he also has to specify whether that single execution will be deterministic or random.

Finally and after launching the execution through the "Run" button, the details of such execution as well as the progress will be displayed in the bottom part of the interface. Once SLACT has finished the execution, it is possible to save the information displayed in the log to a file.

7.3 SLACDC Tool Support

SLACT has been designed and implemented to automate the generation of Combined Test Requirements within the Guarantee Term Testing Level, described in Chapter 5. To address the identification of the Combined TRs in the Compositor Testing Level as described in Chapter 6, a prototype has been implemented. Currently, this prototype is developed as a standalone java component named ***SLACDC_Generator***. In future work, this component will be integrated with the rest of functionalities provided by SLACT.

The *SLACDC_Generator* takes advantage of some of the outputs provided by SLACT. Specifically, we have used the analysis of the SLA and the extraction of relevant information performed by the Parser component. The *SLACDC_Generator* is in charge of analyzing the information gathered from the Parser with the aim at identifying new test requirements. It focuses on the hierarchical structure of the SLA represented by means of the compositors and their guarantee terms. This component automatically identifies a set of Combined Test Requirements for each of the existing compositors, which satisfy the conditions of the SLACDC Test Criterion defined in Section 6.2.2. Furthermore, it deals with the problem regarding the obtaining of non-feasible test requirements by means of implementing the applications of the four specific rules described in Section 6.2.4.

The output of this component is the specification of the Combined Test Requirements. In Figure 7.13 an excerpt of such specification is showed.

1					
2	**Compositor: ExactlyOne**				
3					
4		**GT1**	**GT2**	**ev (ExactlyOne)**	
5	**Combined TR 1**	Inapplicable	Violated	Violated	
6	**Combined TR 2**	Not Determined	Not Determined	Not Determined	
7	**Combined TR 3**	Violated	Inapplicable	Violated	
8	**Combined TR 4**	Inapplicable	Inapplicable	Inapplicable	
9	**Combined TR 5**	Inapplicable	Fulfilled	Fulfilled	
10	**Combined TR 6**	Fulfilled	Inapplicable	Fulfilled	
11					
12					
13	**Compositor: All**				
14					
15		**GT3**	**GT4**	**GT5**	**ev (All)**
16	**Combined TR 7**	Violated	Fulfilled	Inapplicable	Violated
17	**Combined TR 8**	Not Determined	Fulfilled	Inapplicable	Not Determined
18	**Combined TR 9**	Fulfilled	Violated	Inapplicable	Violated
19	**Combined TR 10**	Fulfilled	Inapplicable	Fulfilled	Fulfilled
20	**Combined TR 11**	Fulfilled	Not Determined	Not Determined	Not Determined
21	**Combined TR 12**	Fulfilled	Inapplicable	Violated	Violated
22	**Combined TR 13**	Fulfilled	Fulfilled	Inapplicable	Fulfilled

Figure 7.13: Excerpt of the Combined TRs in the Compositor Testing Level.

For each compositor, the Combined Test Requirements are represented in rows. These Combined TRs include the specification of the Primitive Test Requirements. As we described in Section 6.2.1, in this Compositor Testing Level a Primitive TR is identified for each of the evaluation value of the guarantee term so the Primitive TR is represented with the name of the evaluation value. We have represented these Primitive TRs in the mid-columns of the figure. Finally, in the last column the evaluation value of the compositor based on the evaluation of their guarantee terms is presented.

7.4 Summary

In this chapter we have described the level of automation of the testing techniques described in Chapter 5 and Chapter 6.

On the one hand, we have presented SLACT (SLA Combinatorial Testing), a tool that allows identifying the Primitive Test Requirements in the Guarantee Term Testing Level as well as the generation of test cases by means of deriving the Combined Test Requirements. To address these issues, SLACT implements different components and

makes use of an independent testing tool provided by Microsoft named PICT (Pairwise Independent Combinatorial Tool).

On the other hand, we have described the prototype we have implemented to deal with the generation of the Combined TRs in the Compositor Testing Level. To address this task, we have developed a component named *SLACDC_Generator* that allows obtaining a set of Combined TRs that fulfil the SLACDC test criterion defined in Section 6.2.2 of this dissertation.

Both SLACT and the *SLACDC_Generator* will be used to generate the test requirements in a case study, which is fully described in the next Chapter 8.

Chapter 8

Case Study

The scientist is free, and must be free to ask any question, to doubt any assertion, to seek for any evidence, to correct any errors.

J. Robert Oppenheimer, 1904-1967
American physicist and the scientific director of the Manhattan Project

This chapter describes the application of the testing techniques proposed in this dissertation. It firstly presents the eHealth service-based scenario, including its constituent services and the associated Service Level Agreement. After that, it addresses the generation of tests by using the developed tool presented in the previous chapter.

8.1 Introduction

In previous chapters of this dissertation we introduced the SLATF framework that allows testing SLA-aware service –based applications (Chapter 3) and the logic that evaluates the SLA and its internal elements (Chapter 4). After that, we distinguished between two different testing levels depending whether we take the individual guarantee terms into account (Chapter 5) or we consider the logical relationships of such terms (Chapter 6) in order to design the tests. Furthermore, we developed SLACT tool (Chapter 7), which allows automating most of the tasks involved in such levels. In the present chapter we apply the aforementioned knowledge in a real application by using SLACT.

The software under test (SUT) is an eHealth service-based scenario that was proposed in the context of the PLASTIC Project [111], funded by the European Commission under the FP6 contract number 026955. This scenario has been used in previous service-aware testing approaches [10][44][13][5].

8.2 eHealth Service Based Application

In this section we describe the business logic of the eHealth service-based application as well as the content of the SLA that specifies the conditions to be fulfilled during the execution of the services. After that, we apply the testing techniques described in Chapter 5 and Chapter 6 of this dissertation in order to identify the test requirements and generate the test cases.

8.2.1 Description

The behaviour of the eHealth case study is represented in Figure 8.1.

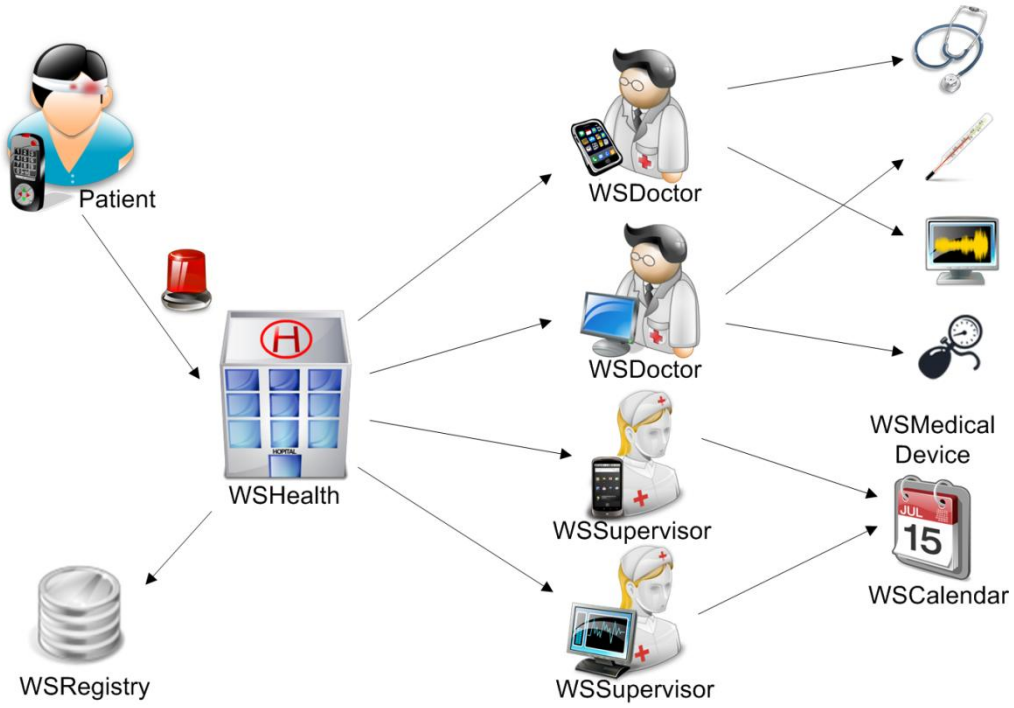


Figure 8.1: eHealth scenario.

The application is deployed as a composite web service (WSHealth) that receives an alarm from patients and triggers appropriate actions to solve such alarms. There are two different types of alarms: *Emergencies* and *No Confirmation*. When an alarm arrives at the system, this service finds the list of professionals who can take responsibility for handling the incident by invoking a service called WSRegistry. This registry provides a list of IP addresses of the professionals who are available at that moment depending on the type of the alarm: doctors (WSDoctor) if it is an Emergency and supervisors (WSSupervisor) if it is No Confirmation. These professionals are connected to the system through wired or mobile devices. Thus, the conditions related to these connections are different. If a doctor is contacted, (s)he may get measures from medical devices (available as WSMedicalDevice services) deployed in the patient's location. If the contacted agent is a supervisor, (s)he should arrange an appointment for the patient using the calendar service WSCalendar.

8.2.2 SLA details

The conditions that govern the execution of this eHealth service based system are specified in an SLA using WS-Agreement. This SLA is included in Appendix 1:

eHealth SLA of this dissertation and can also be publicly downloaded [125]. A graphical representation of the SLA is depicted in Figure 8.2.

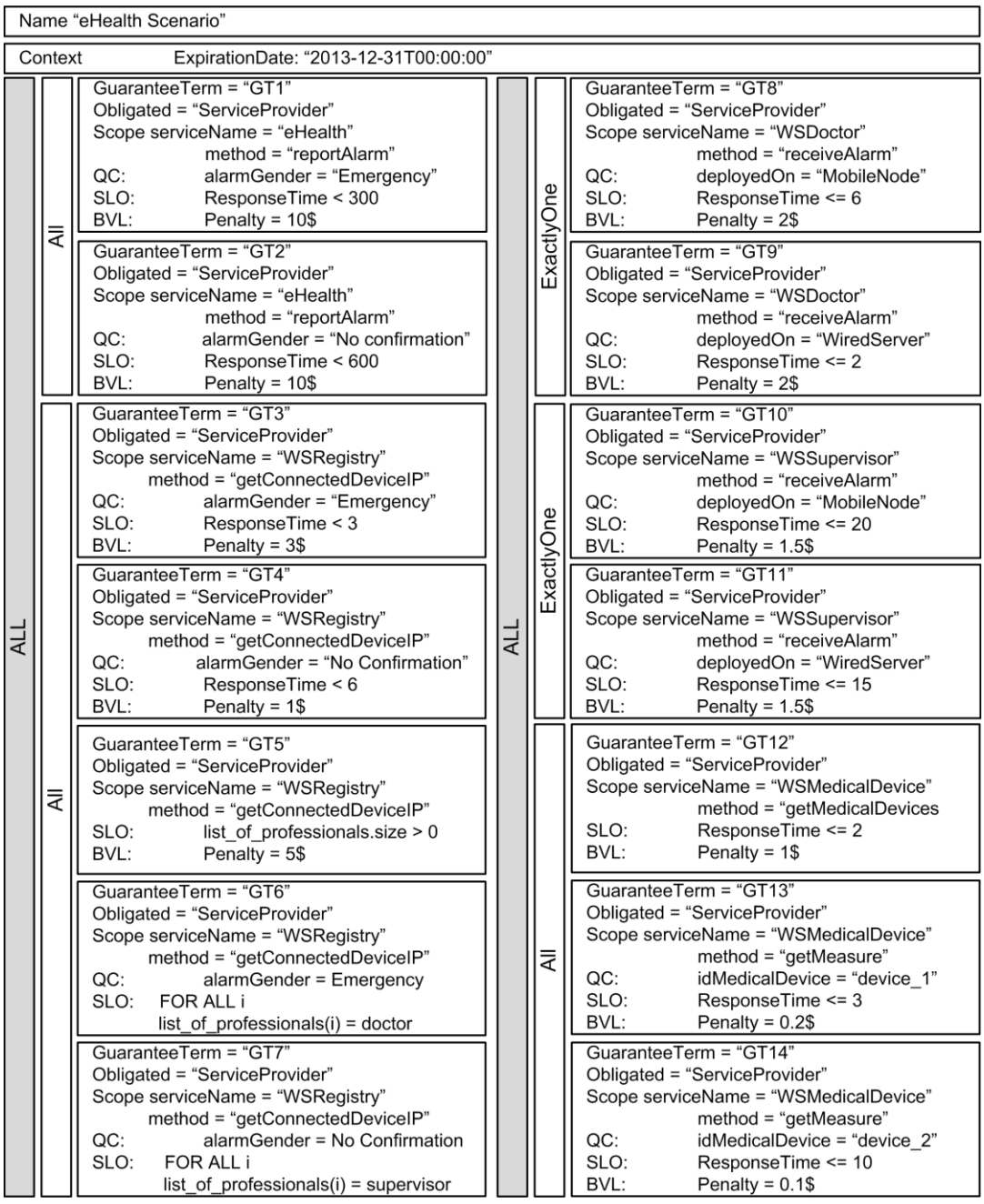


Figure 8.2: eHealth SLA.

This SLA contains 14 Guarantee Terms, which are logically grouped using 5 compositors under the most external and mandatory *All* compositor. In Table 8.1 we represent the distribution of the guarantee terms in each of these compositors.

Compositor	Guarantee Terms
All (1)	GT1, GT2
All (2)	GT3, GT4, GT5, GT6, GT7
ExactlyOne (1)	GT8, GT9
ExactlyOne (2)	GT10, GT11
All (3)	GT12, GT13, GT14

Table 8.1: Structure of the eHealth SLA.

The guarantee terms of the SLA are related to 6 different services and 9 service methods. Twelve of these terms present the whole structure of a Guarantee Term, i.e., a scope, a Qualifying Condition and a Service Level Objective. The other two Guarantee Terms do not have Qualifying Condition.

8.3 Guarantee Term Testing Level

In the first of the testing levels, we address the identification of the Primitive and the Combined Test Requirements as well as the generation of test cases. To do this, we apply the testing techniques described in Chapter 5.

8.3.1 Construction of the Classification Tree

First of all, we apply the steps described in Section 5.2 in order to construct a classification tree by means of identifying the classifications and classes from the eHealth SLA. As we previously stated, the classifications are constructed by representing the compositor elements and the guarantee terms whereas the classes represent the Primitive Test Requirements identified by applying the general case and the particular cases. The results of this process are summarized in Table 8.2.

Service	Method	Classif.	Partic. Cases	Classes
WSHealth	reportAlarm	GT1	PC2	CL1, CL2, CL3, CL4, CL5
		GT2	PC2	CL1, CL2, CL3, CL4, CL5
WSRegistry	getResidentialGateway	-	-	-
	getConnectedDeviceIP	GT3	PC2	CL1, CL2, CL3, CL4, CL5
		GT4	PC2	CL1, CL2, CL3, CL4, CL5
		GT5	PC1	CL1, CL2, CL5
		GT6	PC2	CL1, CL2, CL3, CL4, CL5
		GT7	PC2	CL1, CL2, CL3, CL4, CL5
WSDoctor	receiveAlarm	GT8	PC2	CL1, CL2, CL3, CL4, CL5
		GT9	PC2	CL1, CL2, CL3, CL4, CL5
WSSupervisor	receiveAlarm	GT10	PC2	CL1, CL2, CL3, CL4, CL5
		GT11	PC2	CL1, CL2, CL3, CL4, CL5
WS MedicalDevice	getMedicalDevices	GT12	PC1	CL1, CL2, CL5
	getMeasure	GT13	PC2	CL1, CL2, CL3, CL4, CL5
		GT14	PC2	CL1, CL2, CL3, CL4, CL5
WSCalendar	getAppointment ByMonth	-	-	-
	getAppointment	-	-	-
	TOTAL	14		66

Table 8.2: eHealth Classifications and Classes *.

* Each class CL1-CL6 represents the situation where the Primitive Test Requirement PTR1-PTR6 is exercised.

The services and methods that constitute the case study are represented in the first and second column. The classifications at the lowest level of the tree are represented in the third column. The particular cases applied to identify the classes are outlined in the forth column. The identifiers of the classes of the tree are represented in the last column. In this scenario, the case C1 is applied to GT5 and GT12 (only CL1, CL2 and CL5 are identified) and the case C2 is applied to all the other Guarantee Terms (class CL6 is not identified).

This table is a simplified representation of the classification tree without including the nodes that represent the compositor elements of WS-Agreement. The classifications at the lowest level represent the Guarantee Terms of the SLA and the classes represented in the leaves of the tree are related to the identified Primitive TRs for each Guarantee Term. Hence, the number of identified classifications is 14 and the number of classes is 66. The complete classification tree is represented in Figure 8.3 - Figure 8.8.

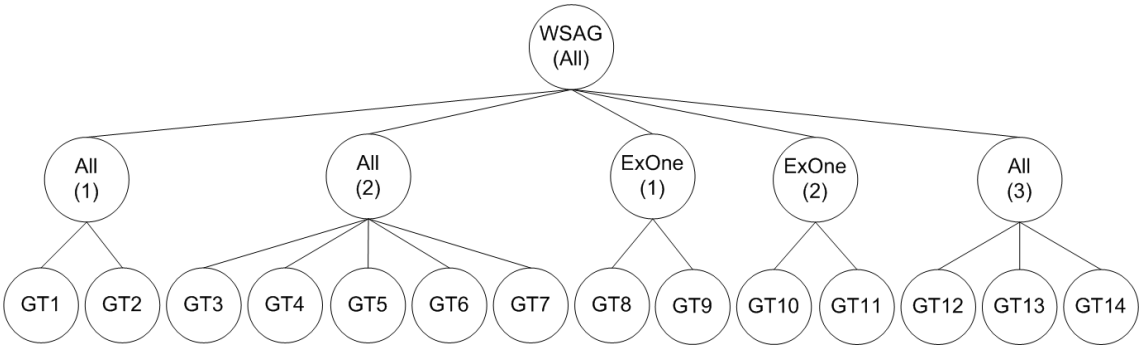


Figure 8.3: Classification Tree (top levels).

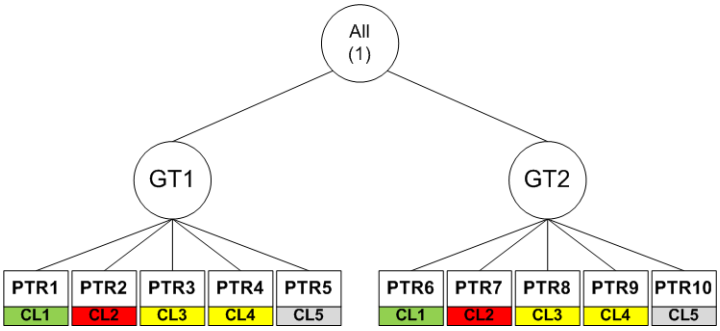


Figure 8.4: All (1) Classification Tree.

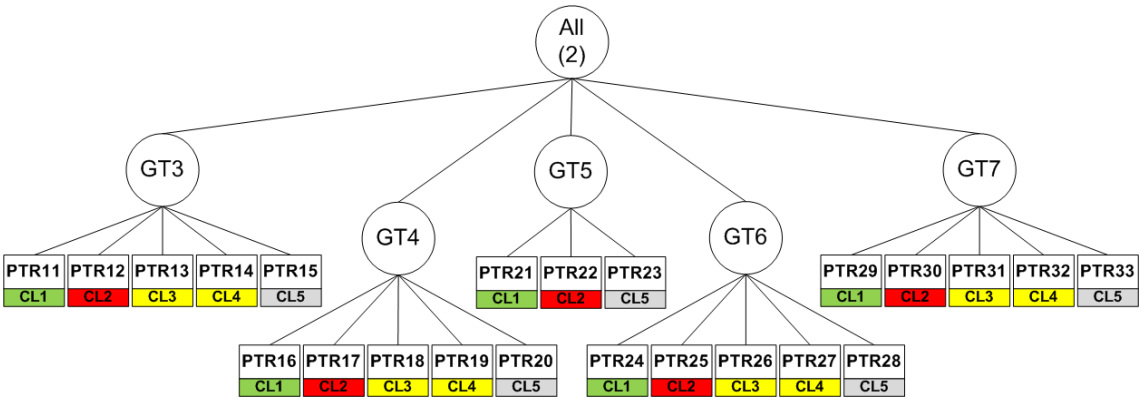


Figure 8.5: All (2) Classification Tree

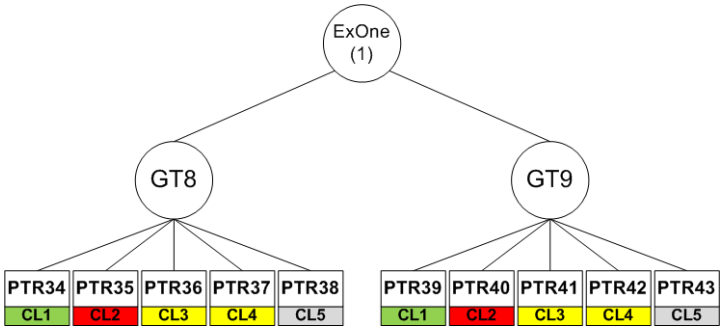


Figure 8.6: ExactlyOne (1) Classification Tree.

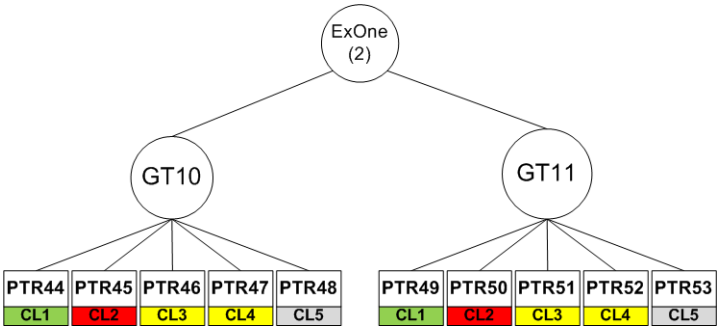


Figure 8.7: ExactlyOne (2) Classification Tree.

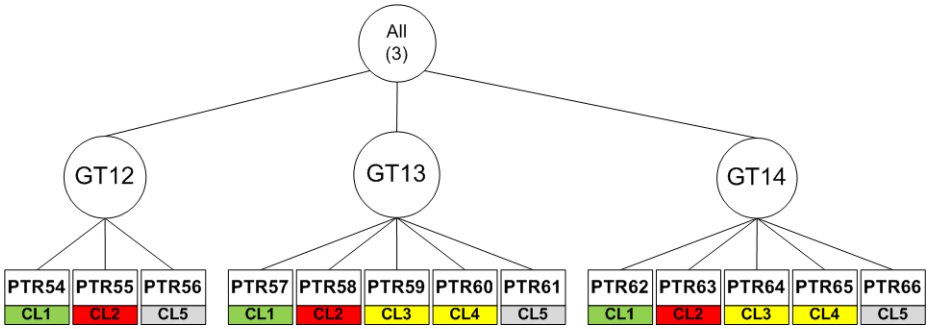


Figure 8.8: All (3) Classification Tree.

This model is the basis to derive the Combined Test Requirements. This task is performed by combining the classes (Primitive TRs) using specific combinatorial criteria and specifying the rules the guide such combinations. In this case, the derivation of the Combined TRs involves the combination of the 14 classifications. Twelve of these classifications have 5 classes each whereas the other two classifications have 3 classes each.

8.3.2 Derivation of Combined Test Requirements

In addition to the construction of the classification tree, we have identified the set of implicit and explicit constraints that will guide the combinations of the classifications and their classes. The implicit constraints are automatically obtained by SLACT whereas the explicit constraints are specified through the User Interface. After analyzing the content of the SLA and relevant information regarding the behaviour of the SUT, 26 constraints (12 implicit and 14 explicit) have been identified in order to guide the derivation of the Combined TRs. All these implicit and explicit constraints are represented in Table 8.3 and Table 8.4 respectively.

ID	Rule	Constrained Guarantee Terms	Explanation
1	I1	GT1, GT2	Both GTs are related to WSHealth.reportAlarm
2	I1	GT3, GT4, GT5, GT6, GT7	All these GTs are related to WSRegistry.getConnectedDeviceIP
3	I1	GT8, GT9	Both GTs are related toWSDoctor.receiveAlarm
4	I1	GT10, GT11	Both GTs are related to WSSupervisor.receiveAlarm
5	I1	GT13, GT14	Both GTs are related to WSMedicalDevice.getMeasure
6	I2	GT1, GT3, GT6	All these GTs have the same QC: alarmType = Emergency
7	I2	GT2, GT4, GT7	All these GTs have the same QC: alarmType = No Confirmation
8	I2	GT8, GT10	Both GTs have the same QC: deployedOn = MobileNode
9	I2	GT9, GT11	Both GTs have the same QC: deployedOn = WiredServer
10	I3	(GT1, GT3, GT6) vs (GT2, GT4, GT7)	Both sets of GTs have mutually disjoint QCs: (alarmType = Emergency) vs (alarmType = No Confirmation)
11	I3	(GT8, GT10) vs (GT9, GT11)	Both sets of GTs have mutually disjoint QCs: (deployedOn = MobileNode) vs (deployedOn = WiredServer)
12	I3	GT13 vs GT14	Both GTs have mutually disjoint QCs: (idDevice = device_1) vs (idDevice = device_2)

Table 8.3: eHealth Implicit Constraints.

ID	Rule	Constrained Guarantee Terms	Explanation
13	E1	GT10, GT11	If the type of the alarm is an Emergency, a supervisor cannot be invoked.
14	E1	GT8, GT9	If the type of the alarm is No Confirmation, a doctor cannot be invoked.
15	E2	GT8, GT9, GT10, GT11	If the registry is not invoked, neither a doctor nor a supervisor can be invoked
16	E3	GT10, GT11	If a doctor is invoked, a supervisor cannot be invoked
17	E3	GT8, GT9	If a supervisor is invoked, a doctor cannot be invoked
18	E4	GT12, GT13, GT14	If a doctor is not invoked, the medical devices cannot be invoked
19	E5	GT1, GT2	WSHealth.reportAlarm must always be invoked.
20	E6	GT8, GT9, GT10, GT11	If no professionals are found, no doctor nor supervisor can be invoked
21	E6	GT13, GT14	If medical devices IPs are not found, no medical devices can be invoked
22	E6	GT1	If GT2 is violated, then GT1 is exercised through CL4.
23	E6	GT3	If GT4 is violated, then GT3 is exercised through CL4.
24	E6	GT9	If GT8 is violated, then GT9 is exercised through CL4.
25	E6	GT11	If GT10 is violated, then GT11 is exercised through CL4.
26	E6	GT13	If GT14 is violated, then GT13 is exercised through CL4.

Table 8.4: eHealth Explicit Constraints.

The identifier of each constraint is represented in the first column. The reference to the implicit or explicit applied constraint is represented in the second column. The Guarantee Terms whose values will be affected by the constraint are represented in the third column. Finally, a brief explanation of the constraint is provided in the last column.

Once we have identified the constraints that should influence the generation of test cases, it is necessary to select the strategy for combining the parameters and their values, and obtaining the Combined TRs that will be used during testing. Three

different strategies are applied in order to grade the level of intensity of the obtained sets of Combined TRs:

i. *Each choice testing* (1-wise) to all the classifications.

ii. *Pair-wise testing* (2-wise) to all the classifications.

iii. *Hybrid.*

The third strategy is a hybrid of the other two which involves applying *pair-wise* testing to a specific set of Guarantee Terms and *each choice* to the rest. In this eHealth scenario, we have applied pair-wise to the most critical functionalities of the SUT (the actions that are triggered after receiving an alarm of type *Emergency*). It is remarkable that the Guarantee Terms that are related to the arrival of an Emergency are scattered in the SLA and, thus, the classifications that represent such Guarantee Terms (GT1, GT3, GT8, GT9, GT12, GT13 and GT14) are represented in different branches of the tree. This hybrid strategy provided an intermediate level of intensity between the weakest coverage provided by the *each choice* coverage and the strongest intensity provided by the *pair-wise* coverage.

Considering that SLACT allows applying a non-deterministic algorithm and, therefore, obtaining different sets of Combined TRs for the same input, we have executed the tool for each of the three coverage strategies, and run the combinations for each strategy several times and get the output with the lowest number of Combined TRs that satisfies such coverage strategy. In order to check the behaviour of the combinations, for each strategy we have run the combinatorial testing tool 3000 times and we have obtained a minimum number of 10, 42 and 32 Combined TRs for the each choice, pair-wise and hybrid strategies respectively.

The results of these multiple executions are represented in Figure 8.9. The x-axis in the figure represents the number of Combined TR generated for each strategy by the tool. The y-axis represents the number of times each size is obtained. For example, for the *each choice* strategy, a set of 11 generated Combined TRs has been obtained more than 1200 times.

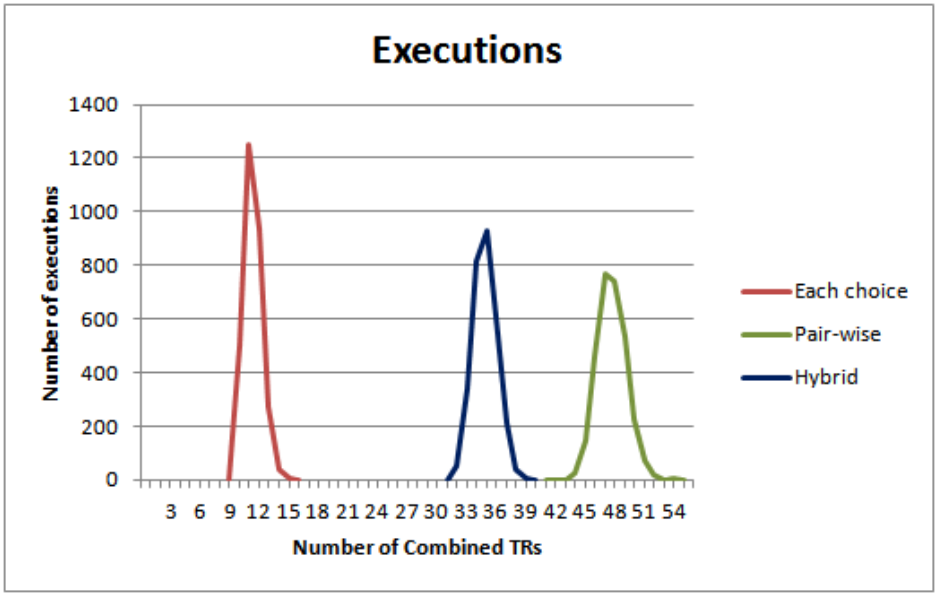


Figure 8.9: Executions of the three combinatorial strategies.

Each choice strategy

In the case of the *each choice* strategy, the results obtained for SLACT presents a *mean* (μ) of 11.37 (number of Combined TRs) and a *standard deviation* (σ) of 0.91. To be more specific, 95% of the sets contain approximately between 10 and 13 Combined TRs. In the case of the *pair-wise* strategy, the parameters are μ = 47.68, σ = 1.48 and 95% of the executions have provided a set with a number of Combined TRs between 45 and 50. Finally, the hybrid strategy is represented by μ = 34.84, σ = 1.22 and 95% of the sets would contain between 33 and 37 Combined TRs.

Analyzing the results for each applied coverage strategy and starting with the first one (*each choice*), we have obtained a set of 10 Combined TRs (the output file is represented in Figure 8.10). In this file, the classifications of the tree (Guarantee Terms) are represented in columns and the Combined TRs obtained through the combination of the classes are represented in rows.

	GT1	GT2	GT3	GT4	GT5	GT6	GT7	GT8	GT9	GT10	GT11	GT12	GT13	GT14
CTR1	CL2	CL4	CL1	CL3	CL1	CL1	CL4	CL3	CL1	CL5	CL5	CL2	CL1	CL3
CTR2	CL4	CL2	CL4	CL2	CL2	CL3	CL2	CL5	CL5	CL5	CL5	CL5	CL5	CL5
CTR3	CL3	CL1	CL3	CL1	CL1	CL4	CL1	CL5	CL5	CL3	CL1	CL5	CL5	CL5
CTR4	CL1	CL3	CL2	CL4	CL1	CL2	CL3	CL2	CL4	CL5	CL5	CL1	CL2	CL4
CTR5	CL3	CL1	CL4	CL1	CL1	CL3	CL2	CL5	CL5	CL4	CL2	CL5	CL5	CL5
CTR6	CL4	CL2	CL3	CL1	CL1	CL3	CL1	CL5	CL5	CL2	CL4	CL5	CL5	CL5
CTR7	CL2	CL4	CL1	CL3	CL1	CL2	CL4	CL1	CL3	CL5	CL5	CL1	CL3	CL1
CTR8	CL4	CL1	CL5	CL5	CL5	CL5	CL5	CL5	CL5	CL5	CL5	CL5	CL5	CL5
CTR9	CL2	CL3	CL1	CL3	CL1	CL1	CL3	CL4	CL2	CL5	CL5	CL2	CL4	CL2
CTR10	CL4	CL2	CL4	CL1	CL1	CL4	CL1	CL5	CL5	CL1	CL3	CL5	CL5	CL5

Figure 8.10: Combined TRs output file.

The classification tree as well as the Primitive TRs covered in each Combined TR in the each-choice strategy is represented in Figure 8.11 (GT1-GT7) and Figure 8.12 (GT8-GT14).

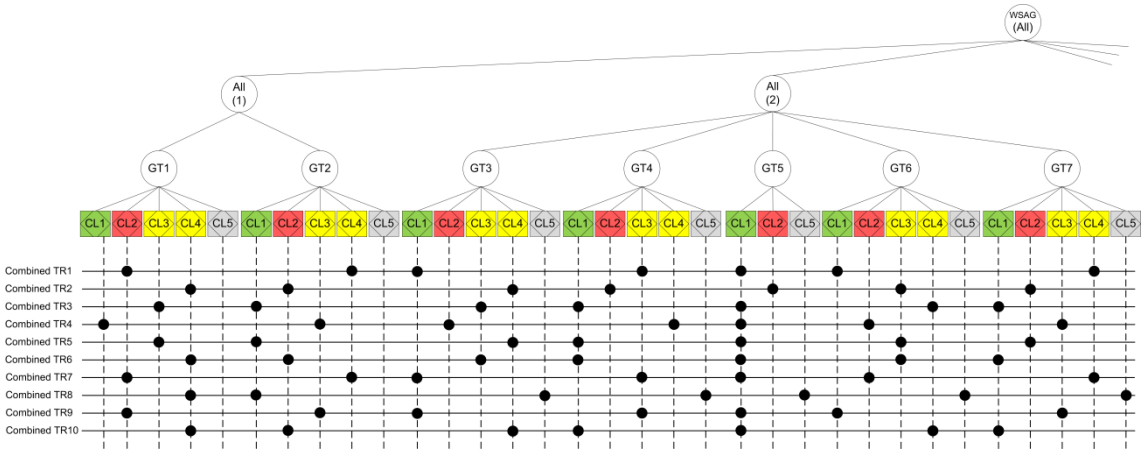


Figure 8.11: Classification tree of Combined TRs for each choice (I).

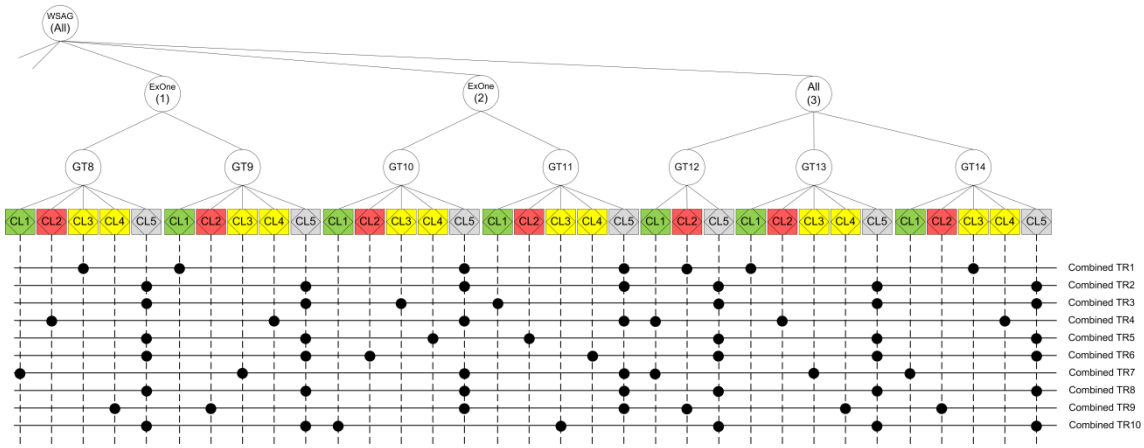


Figure 8.12: Classification tree of Combined TRs for each choice (II).

With these ten Combined TRs, 64 of the 66 classes are exercised at least once (except CL5 for GT1 and GT2 that are constrained by the explicit rule with ID 19 of Table 8.4 and, thus, they are impossible to be exercised) and the coverage report provided by the Analyzer component of SLACT is represented in Table 8.5.

	Each choice				
	CL1	**CL2**	**CL3**	**CL4**	**CL5**
GT1	3	3	2	2	0
GT2	3	1	3	3	0
GT3	3	2	3	1	1
GT4	3	1	3	2	1
GT5	8	1	-	-	1
GT6	3	2	2	2	1
GT7	2	2	4	1	1
GT8	1	1	1	1	6
GT9	1	1	1	1	6
GT10	1	1	1	1	6
GT11	1	1	1	1	6
GT12	2	2	-	-	6
GT13	1	1	1	1	6
GT14	1	1	1	1	6

Table 8.5: Number of times the classes are covered in: each choice.

In the first column we represent the set of Guarantee Terms and in the first row we represent the classes obtained for each GT. In the table, each cell specifies the number of times such class is exercised within this set of Combined TRs. For example, the class CL1 identified from the specification of GT5 is exercised in 8 Combined TRs for this strategy. If there is a hyphen (-) in a cell, it means that the class represented in such column was not identified due to the application of the particular cases explained in Section 5.2.2.

Pair-wise strategy

Regarding the second of the applied coverage strategies (*pair-wise*), the set with the least number of Combined TRs that we obtained contained 42 Combined TRs. The number of Combined TRs obtained is higher than in the 1-wise strategy because, now, each potential pair of classes of different classifications (Guarantee Terms) is included in at least one Combined TR. The classification tree with the Primitive TRs and the Combined TRs for this strategy is represented in Figure 8.13 (GT1-GT7) and Figure 8.14 (GT8-GT14).

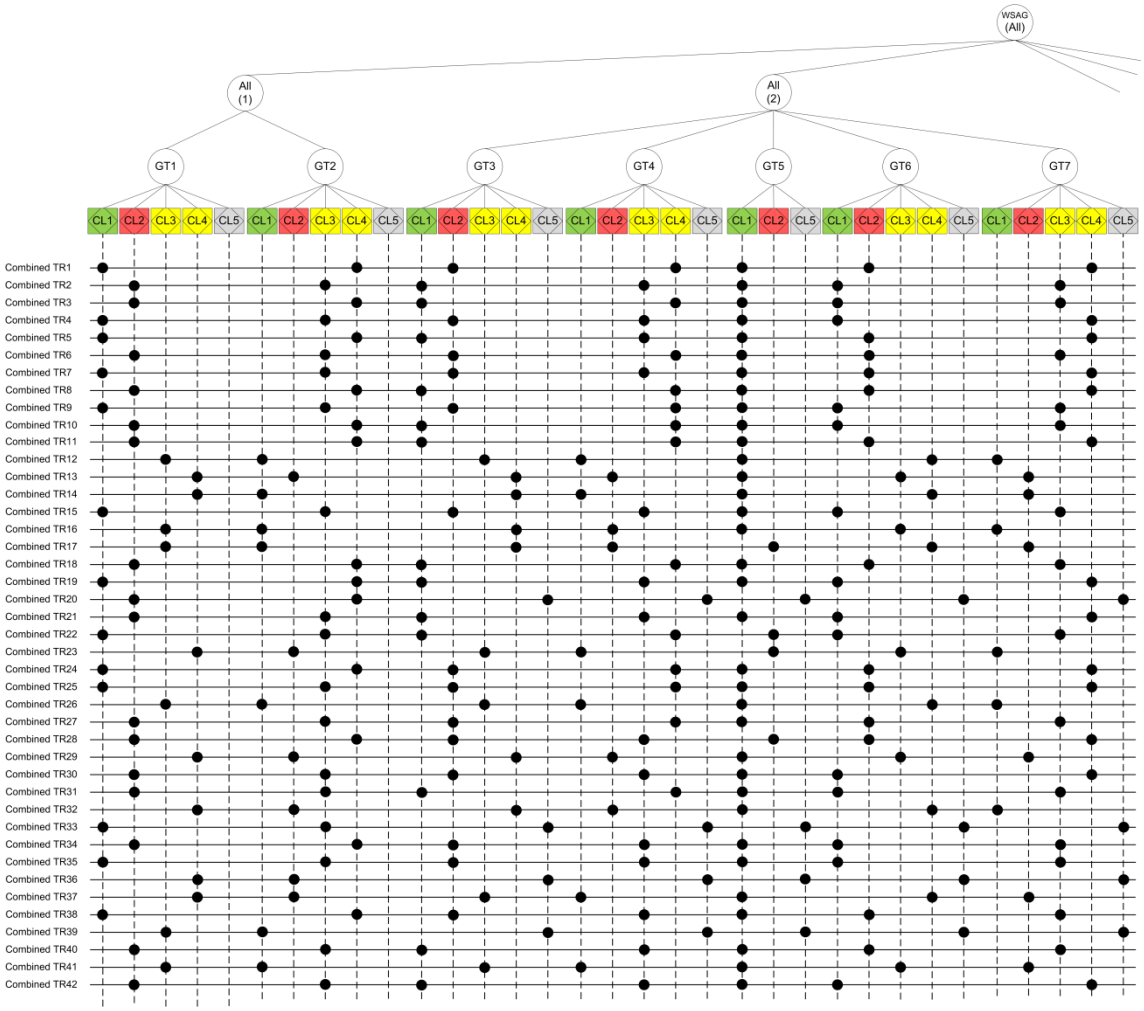


Figure 8.13: Classification tree of Combined TRs for pair-wise (I).

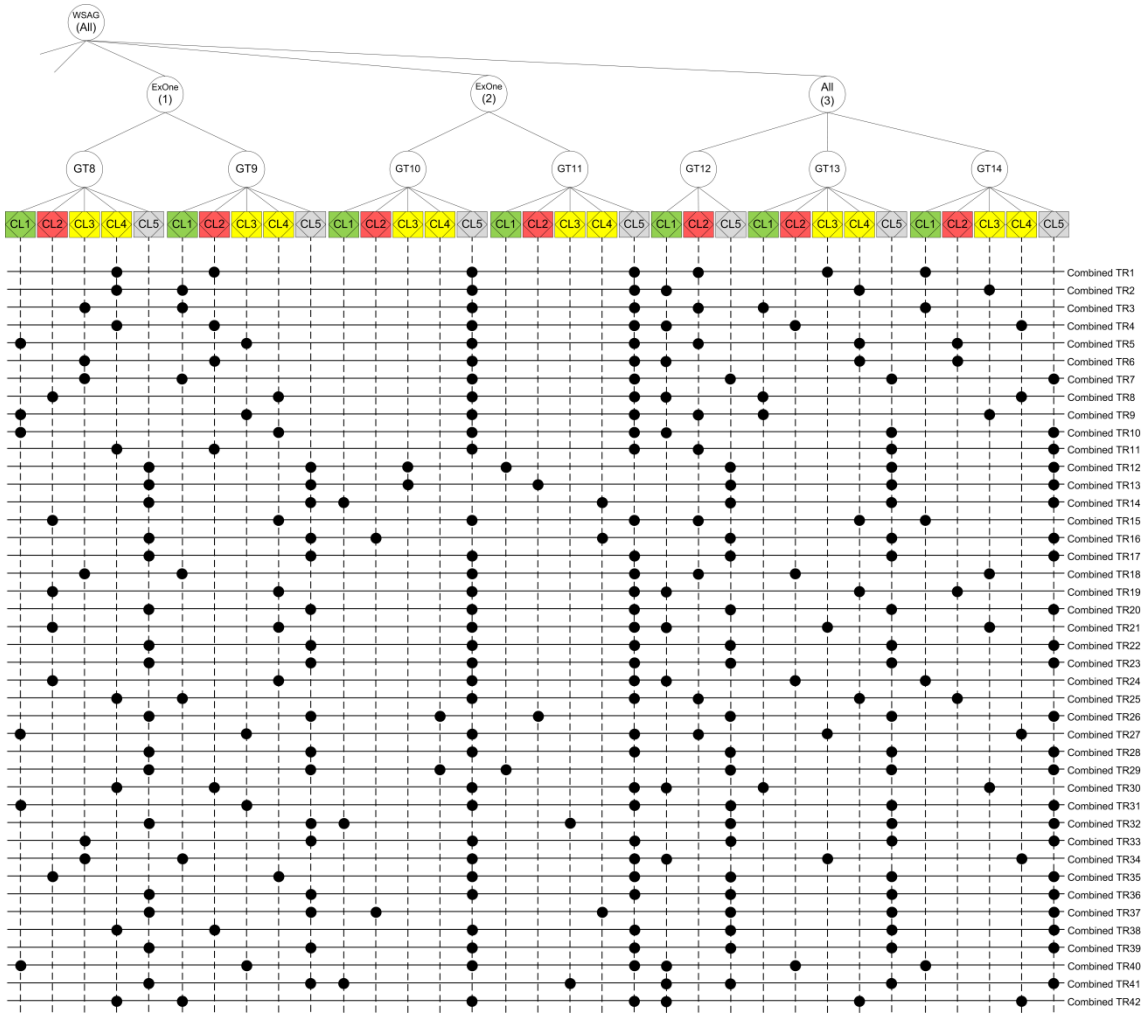


Figure 8.14: Classification tree of Combined TRs for pair-wise (II).

The results provided by the Analyzer regarding the coverage for the classes of each Guarantee Term are also represented in Table 8.6 (b).

	Each choice (a)					Pair-wise (b)				
	CL1	CL2	CL3	CL4	CL5	CL1	CL2	CL3	CL4	CL5
GT1	3	3	2	2	0	13	16	6	7	0
GT2	3	1	3	3	0	7	7	16	13	0
GT3	3	2	3	1	1	13	13	5	6	4
GT4	3	1	3	2	1	6	6	14	13	4
GT5	8	1	-	-	1	34	34	-	-	4
GT6	3	2	2	2	1	14	14	5	6	4
GT7	2	2	4	1	1	5	5	14	13	4
GT8	1	1	1	1	6	6	6	5	8	17
GT9	1	1	1	1	6	7	7	5	7	17
GT10	1	1	1	1	6	3	3	2	2	33
GT11	1	1	1	1	6	2	2	2	3	33
GT12	2	2	-	-	6	12	12	-	-	21
GT13	1	1	1	1	6	4	4	4	7	23
GT14	1	1	1	1	6	5	5	5	5	23

Table 8.6: Number of times the classes are covered in: each choice (a) and pair-wise (b).

As it can be seen in this table, all classes in the 2-wise strategy have been exercised more times than in the case of *each choice*. This indicates a higher level of intensity in the tests. Here again and due to the specification of the explicit rule 19, there are two classes that are never exercised (CL5 for GT1 and GT2).

Hybrid-wise strategy

Finally, we have also applied the hybrid-wise strategy with the aim of grading the intensity of the tests depending on the critical functionality of the eHealth system. With this strategy, the smallest set we have obtained contains 32 Combined TRs. The classification tree with the test requirements is represented in Figure 8.15 (GT1-GT7) and Figure 8.16 (GT8-GT14).

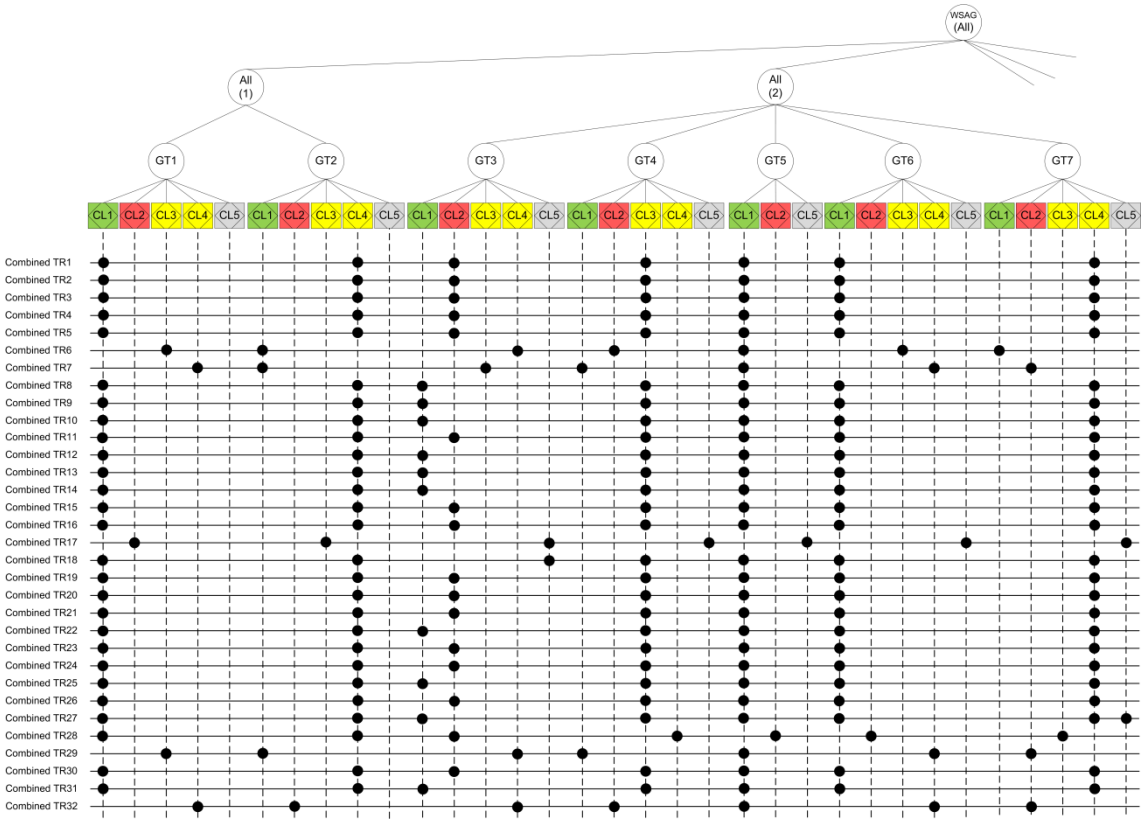


Figure 8.15: Classification tree of Combined TRs for Hybrid-wise (I).

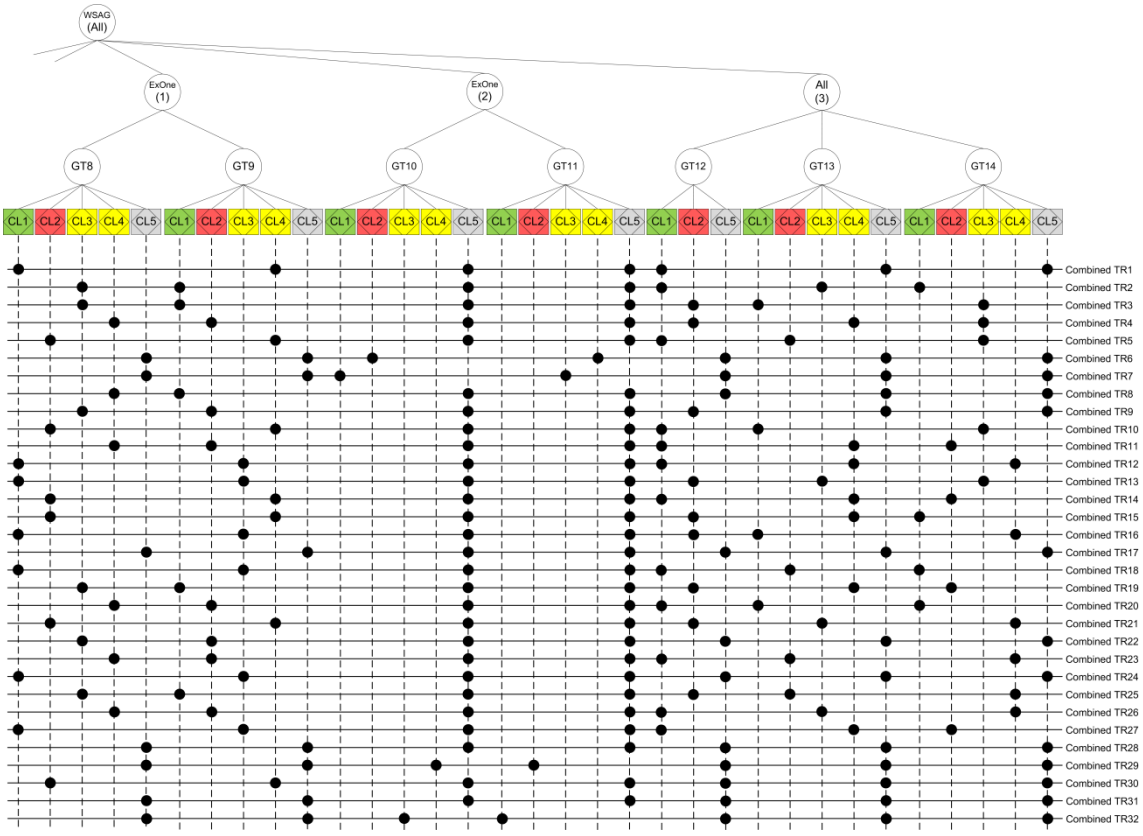


Figure 8.16: Classification tree of Combined TRs for Hybrid-wise (II).

The results provided by the Analyzer are represented in Table 8.7 (c)

	Each choice (a)					Pair-wise (b)					Hybrid (c)				
	CL1	CL2	CL3	CL4	CL5	CL1	CL2	CL3	CL4	CL5	CL1	CL2	CL3	CL4	CL5
GT1	3	3	2	2	0	13	16	6	7	0	27	1	2	2	0
GT2	3	1	3	3	0	7	7	16	13	0	3	1	2	26	0
GT3	3	2	3	1	1	13	13	5	6	4	11	16	1	3	1
GT4	3	1	3	2	1	6	6	14	13	4	2	2	26	1	1
GT5	8	1	-	-	1	34	34	-	-	4	30	1	-	-	1
GT6	3	2	2	2	1	14	14	5	6	4	26	1	1	3	1
GT7	2	2	4	1	1	5	5	14	13	4	1	3	1	26	1
GT8	1	1	1	1	6	6	6	5	8	17	7	6	6	6	7
GT9	1	1	1	1	6	7	7	5	7	17	5	7	6	7	7
GT10	1	1	1	1	6	3	3	2	2	33	1	1	1	1	28
GT11	1	1	1	1	6	2	2	2	3	33	1	1	1	1	28
GT12	2	2	-	-	6	12	12	-	-	21	12	9	-	-	11
GT13	1	1	1	1	6	4	4	4	7	23	4	4	4	7	13
GT14	1	1	1	1	6	5	5	5	5	23	4	4	5	6	13

Table 8.7: Number of times the classes are covered in: each choice (a), pair-wise (b) and hybrid (c).

There are some classes that are as much tested as in the pair-wise strategy because they are related to Guarantee Terms that affect the more critical part of the SUT (Emergencies). On the other hand, there are other classes that are covered with less intensity, representing non-critical situations of the SUT.

All the results derived from the coverage of the different classes are synthesised in Figure 8.17.

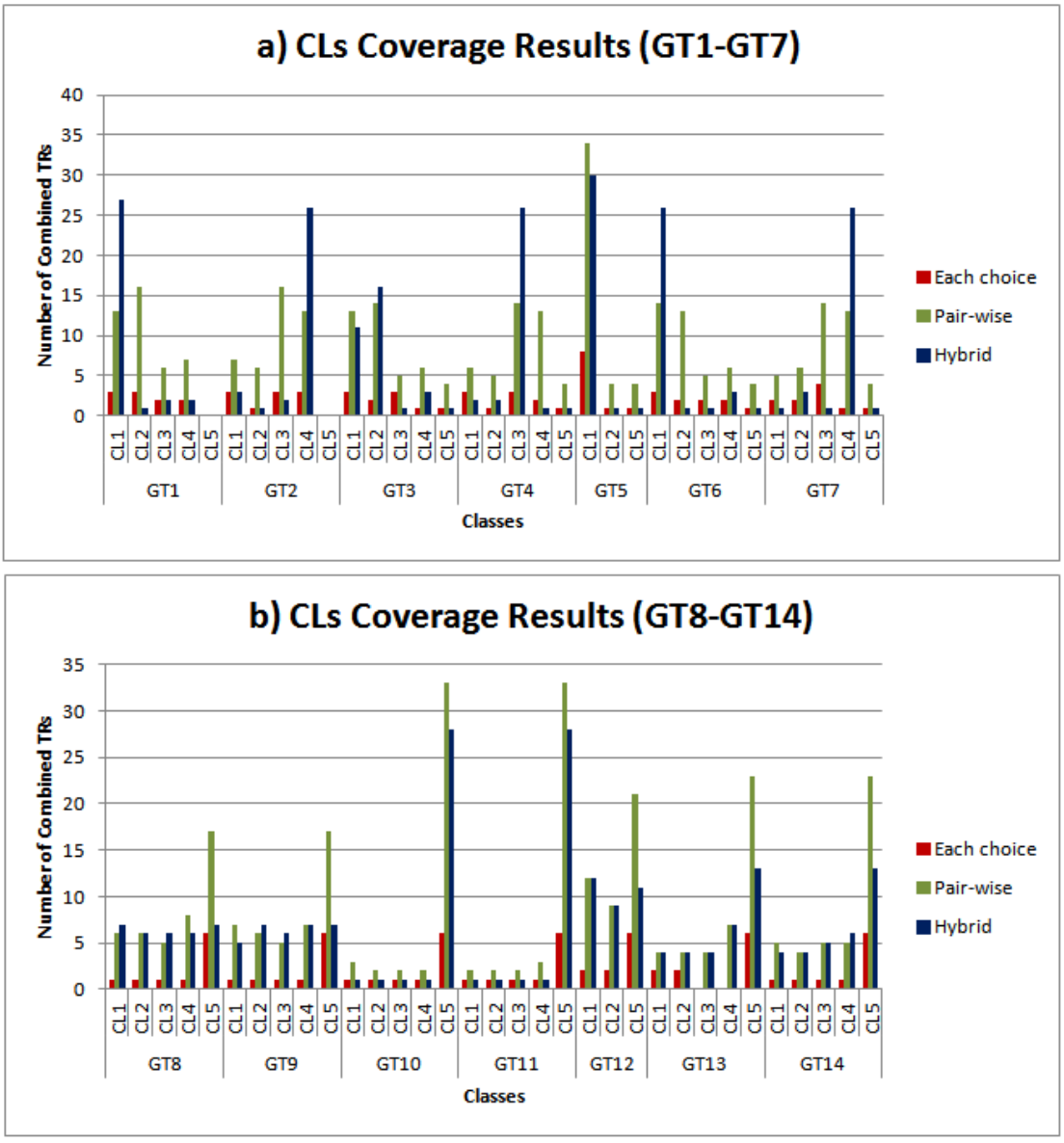


Figure 8.17: Classes coverage results.

In the figure, the x-axis represents the Guarantee Terms and their corresponding classes and the y-axis represents the number of times each such class is exercised within the applied coverage strategy. As shown in the figure, in the hybrid strategy there are specific classes of Guarantee Terms that are much more exercised than others (for example, CL1 of GT1, CL2 and CL3 of GT2 or CL1 of GT5). These classes are related to situations that are considered critical for the behaviour of the SUT (e.g., the arrival of an alarm of type Emergency). Thus, we have decided to combine them more thoroughly than classes related to a non-critical behaviour of the SUT.

8.3.3 Generation of Test Cases

Once we have obtained the Combined Test Requirements for each of the selected coverage strategies, the last step involves the generation of the test cases that exercise such Combined TRs. As it is stated in Section 5.3, in this testing level a Combined TR represents a complete scenario regarding the service-based application so we will generate one test case for each of the Combined TRs identified. This means that we will have 10 test cases for the first strategy (*each-choice*), 42 test cases for the second strategy (*pair-wise*) and 32 test cases for the last strategy (*hybrid*).

To address the generation of the test cases, it is necessary to have some knowledge about the behaviour of the service-based application. In this case, we make use of an UML sequence diagram, also provided within the context of the PLASTIC project (represented in Figure 8.18), in order to manage the order of the service invocations.

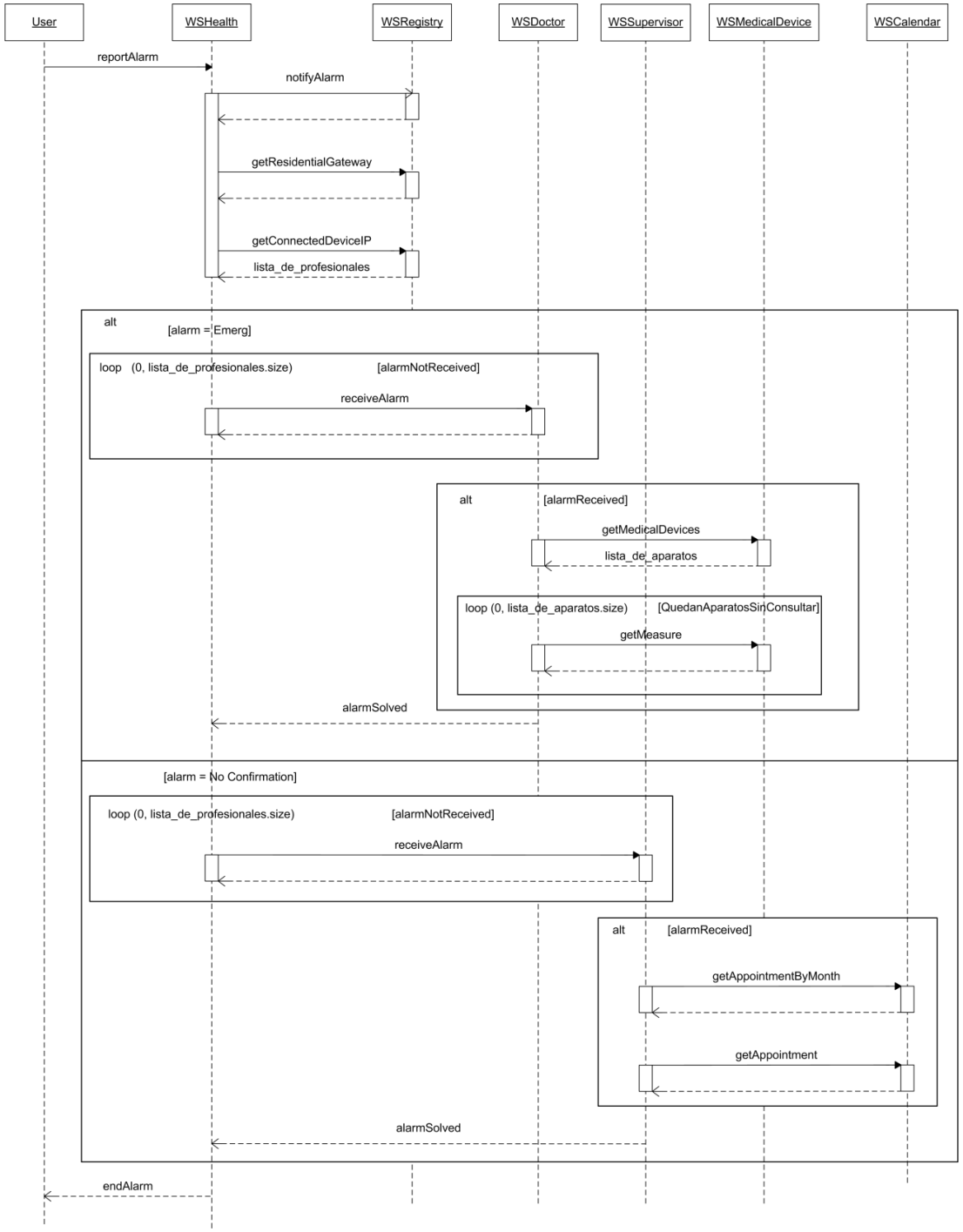


Figure 8.18: eHealth UML sequence diagram.

8.3.3.1 Each choice strategy

In this section, the ten test cases generated from the Combined TRs in the *each choice* strategy are specified in Table 8.8 - Table 8.17. The process for testing the scenario represented in such test case is described in the first column. The expected

output is showed in the second column. The values inherent to the evaluation of the guarantee terms are represented in the third column. Finally, the Primitive TRs (*classes* in CTM) exercised within the test case are represented in the last column.

For the first test case we also represent in Figure 8.19 the specification of the Combined TR so we can see the co-relation between the content of such Combined TR and how the Primitive TRs are exercised in the test case.

	GT1	GT2	GT3	GT4	GT5	GT6	GT7	GT8	GT9	GT10	GT11	GT12	GT13	GT14
CTR1	CL2	CL4	CL1	CL3	CL1	CL1	CL4	CL3	CL1	CL5	CL5	CL2	CL1	CL3

Figure 8.19: CTR1 of each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty and correct list of professionals in less than 3 seconds. The list contains only doctors.	ev(GT3) = F ev(GT4) = I ev(GT5) = F ev(GT6) = F ev(GT7) =I	ex(GT3) = CL1 ex(GT4) = CL3 ex(GT5) = CL1 ex(GT6) = CL1 ex(GT7) = CL4
A doctor connected to the system through a wired device is contacted.	(s)he provides a response in less than 2 seconds.	ev(GT8) =I ev(GT9) = F	ex(GT8) = CL3 ex(GT9) = CL1
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	ex(GT10) = CL5 ex(GT11) = CL5
The doctor finds the list of devices deployed in the patient's home.	This invocation spends more than 2 seconds.	ev(GT12) = V	ex(GT12) = CL2
The doctor gets the measure of device_1.	The measure is provided in less than 3 seconds.	ev(GT13) = F	ex(GT13) = CL1
The doctor does not consult the measure of device_2.		ev(GT14) = I	ex(GT14) = CL3
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending more than 600 seconds.	ev(GT1) = V ev(GT2) = I	ex(GT1) = CL2 ex(GT2) = CL4
In this scenario, the guarantee terms GT1 and GT12 have been violated so the corresponding penalties should be applied.			

Table 8.8: Test Case 1 for the each-choice strategy.

The rest of the test cases are represented in the tables shown hereafter.

Process	Expected Output	Eval. Values	PTRs exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides an empty list of supervisors, spending more than 6 seconds to give the response.	ev(GT3) = I ev(GT4) = V ev(GT5) = V ev(GT6) = I ev(GT7) = V	ex(GT3) = CL4 ex(GT4) = CL2 ex(GT5) = CL2 ex(GT6) = CL3 ex(GT7) = CL2
No doctors are contacted.		ev(GT8) = N ev(GT9) = N	ex(GT8) = CL5 ex(GT9) = CL5
No supervisors are contacted.		ev(GT10) = N ev(GT11) = N	ex(GT10) = CL5 ex(GT11) = CL5
No medical devices are consulted.		ev(GT12) = N ev(GT13) = N ev(GT14) = N	ex(GT12) = CL5 ex(GT13) = CL5 ex(GT14) = CL5
	The eHealth system provides a response to the patient spending more than 600 seconds.	ev(GT1) = I ev(GT2) = V	ex(GT1) = CL4 ex(GT2) = CL2
In this scenario, the guarantee terms GT2, GT4, GT5 and GT7 have been violated so the corresponding penalties should be applied.			

Table 8.9: Test Case 2 for the each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty and correct list of professionals in less than 3 seconds. The list contains only supervisors.	ev (GT3) = I ev(GT4) = F ev(GT5) = F ev(GT6) = I ev(GT7) = F	ex(GT3) = CL3 ex(GT4) = CL1 ex(GT5) = CL1 ex(GT6) = CL4 ex(GT7) = CL1
A supervisor connected to the system through a wired device is contacted.	(s)he provides a response in less than 15 seconds	ev(GT10) = I ev(GT11) = F	ex(GT10) = CL3 ex(GT11) = CL1
As a supervisor has been contacted, no doctors are invoked.		ev(GT8) = N ev(GT9) = N	ex(GT8) = CL5 ex(GT9) = CL5
As the alarm is managed by a supervisor, no medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	ex(GT12) = CL5 ex(GT13) = CL5 ex(GT14) = CL5
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending less than 300 seconds.	ev(GT1) = I ev(GT2) = F	ex(GT1) = CL3 ex(GT2) = CL1
In this scenario, none of the guarantee terms has been violated so no consequences are derived.			

Table 8.10: Test Case 3 for the each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in more than 6 seconds. The list contains both doctors and supervisors.	ev(GT3) = V ev(GT4) = I ev(GT5) = F ev(GT6) = V ev(GT7) = I	ex(GT3) = CL2 ex(GT4) = CL4 ex(GT5) = CL1 ex(GT6) = CL2 ex(GT7) = CL3
A doctor connected to the system through a mobile device is contacted.	(s)he provides a response in more than 6 seconds.	ev(GT8) = V ev(GT9) = I	ex(GT8) = CL2 ex(GT9) = CL4
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	ex(GT10) = CL5 ex(GT11) = CL5
The doctor finds the list of devices deployed in the patient's home.	This invocation spends less than 2 seconds.	ev(GT12) =F	ex(GT12) = CL1
The doctor gets the measure of device_1.	The measure is provided in more than 10 seconds.	ev(GT13) = V	ex(GT13) = CL2
The doctor does not consult the measure of device_2.		ev(GT14) = I	ex(GT14) = CL4
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending less than 300 seconds.	ev(GT1) = F ev(GT2) = I	ex(GT1) = CL1 ex(GT2) = CL3
In this scenario, the guarantee terms GT3, GT6, GT8 and GT13 have been violated so the corresponding penalties should be applied.			

Table 8.11: Test Case 4 for the each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in a time between 3 seconds and than 6 seconds. The list contains both doctors and supervisors.	ev (GT3) = I ev(GT4) = F ev(GT5) = F ev(GT6) = I ev(GT7) = V	ex(GT3) = CL4 ex(GT4) = CL1 ex(GT5) = CL1 ex(GT6) = CL3 ex(GT7) = CL2
A supervisor connected to the system through a wired device is contacted.	(s)he provides a response in more than 20 seconds.	ev(GT10) = I ev(GT11) = V	ex(GT10) = CL4 ex(GT11) = CL2
As a supervisor has been contacted, no doctors are invoked.		ev(GT8) = N ev(GT9) = N	ex(GT8) = CL5 ex(GT9) = CL5
As the alarm is managed by a supervisor, no medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	ex(GT12) = CL5 ex(GT13) = CL5 ex(GT14) = CL5
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending less than 300 seconds.	ev(GT1) = I ev(GT2) = F	ex(GT1) = CL3 ex(GT2) = CL1
In this scenario, the guarantee terms GT7 and GT11 have been violated so the corresponding penalties should be applied.			

Table 8.12: Test Case 5 for the each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 3 seconds. The list contains only supervisors.	ev(GT3) = I ev(GT4) = F ev(GT5) = F ev(GT6) = I ev(GT7) = F	ex(GT3) = CL3 ex(GT4) = CL1 ex(GT5) = CL1 ex(GT6) = CL3 ex(GT7) = CL1
A supervisor connected to the system through a wired device is contacted.	(s)he provides a response in more than 20 seconds.	ev(GT10) = V ev(GT11) = I	ex(GT10) = CL2 ex(GT11) = CL4
As a supervisor has been contacted, no doctors are invoked.		ev(GT8) = N ev(GT9) = N	ex(GT8) = CL5 ex(GT9) = CL5
As the alarm is managed by a supervisor, no medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	ex(GT12) = CL5 ex(GT13) = CL5 ex(GT14) = CL5
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending more than 600 seconds.	ev(GT1) = I ev(GT2) = V	ex(GT1) = CL4 ex(GT2) = CL2
In this scenario, the guarantee terms GT2 and GT10 have been violated so the corresponding penalties should be applied.			

Table 8.13: Test Case 6 for the each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 3 seconds. The list contains both doctors and supervisors.	ev(GT3) = F ev(GT4) = I ev(GT5) = F ev(GT6) = V ev(GT7) = I	ex(GT3) = CL1 ex(GT4) = CL3 ex(GT5) = CL1 ex(GT6) = CL2 ex(GT7) = CL4
A doctor connected to the system through a mobile device is contacted.	(s)he provides a response in less than 2 seconds.	ev(GT8) = F ev(GT9) = I	ex(GT8) = CL1 ex(GT9) = CL3
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	ex(GT10) = CL5 ex(GT11) = CL5
The doctor finds the list of devices deployed in the patient's home.	This invocation spends less than 2 seconds.	ev(GT12) =F	ex(GT12) = CL1
The doctor does not consult the measure of device_1.		ev(GT13) = I	ex(GT13) = CL3
The doctor gets the measure of device_2.	The measure is provided in less than 3 seconds.	ev(GT14) = F	ex(GT14) = CL1
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending more than 600 seconds.	ev(GT1) = V ev(GT2) = I	ex(GT1) = CL2 ex(GT2) = CL4
In this scenario, the guarantee terms GT1 and GT6 have been violated so the corresponding penalties should be applied.			

Table 8.14: Test Case 7 for the each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is not queried.		ev(GT3) = N ev(GT4) = N ev(GT5) = N ev(GT6) = N ev(GT7) = N	ex(GT3) = CL5 ex(GT4) = CL5 ex(GT5) = CL5 ex(GT6) = CL5 ex(GT7) = CL5
As the list of professionals is not available, neither doctors nor supervisors are invoked.		ev(GT8) = N ev(GT9) = N ev(GT10) = N ev(GT11) = N	ex(GT10) = CL5 ex(GT11) = CL5 ex(GT8) = CL5 ex(GT9) = CL5
No medical devices are consulted.		ev(GT12) =N ev(GT13) =N ev(GT14) =N	ex(GT12)= CL5 ex(GT13)= CL5 ex(GT14)= CL5
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending between 300 and 600 seconds	ev(GT1) = I ev(GT2) = F	ex(GT1) = CL4 ex(GT2) = CL1
In this scenario, none of the guarantee terms has been violated so no consequences are derived.			

Table 8.15: Test Case 8 for the each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in more than 6 seconds. The list contains only doctors.	ev(GT3) = F ev(GT4) = I ev(GT5) = F ev(GT6) = F ev(GT7) = I	ex(GT3) = CL1 ex(GT4) = CL3 ex(GT5) = CL1 ex(GT6) = CL1 ex(GT7) = CL3
A doctor connected to the system through a wired device is contacted.	(s)he provides a response in more than 6 seconds.	ev(GT8) = I ev(GT9) = V	ex(GT8) = CL4 ex(GT9) = CL2
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	ex(GT10) = CL5 ex(GT11) = CL5
The doctor finds the list of devices deployed in the patient's home.	This invocation spends more than 2 seconds.	ev(GT12) =V	ex(GT12) = CL2
The doctor does not consult the measure of device_1.		ev(GT13) = I	ex(GT13) = CL4
The doctor gets the measure of device_2.	The measure is provided in more than 10 seconds.	ev(GT14) = V	ex(GT14) = CL2
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending between 300 and 600 seconds.	ev(GT1) = V ev(GT2) = I	ex(GT1) = CL2 ex(GT2) = CL4
In this scenario, the guarantee terms GT1, GT9 and GT12 have been violated so the corresponding penalties should be applied.			

Table 8.16: Test Case 9 for the each-choice strategy.

Process	Expected Output	Eval. Values	PTRs exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in a time between 3 and 6 seconds. The list contains only supervisors.	ev(GT3) = I ev(GT4) = F ev(GT5) = F ev(GT6) = I ev(GT7) = F	ex(GT3) = CL4 ex(GT4) = CL1 ex(GT5) = CL1 ex(GT6) = CL4 ex(GT7) = CL1
A supervisor connected to the system through a mobile device is contacted.	(s)he provides a response in less than 15 seconds.	ev(GT10) = F ev(GT11) = I	ex(GT10) = CL1 ex(GT11) = CL3
As a supervisor has been contacted, no doctors are invoked.		ev(GT8) = N ev(GT9) = N	ex(GT8) = CL5 ex(GT9) = CL5
As the alarm is managed by a supervisor, no medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	ex(GT12) = CL5 ex(GT13) = CL5 ex(GT14) = CL5
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending between 300 and 600 seconds.	ev(GT1) = I ev(GT2) = V	ex(GT1) = CL4 ex(GT2) = CL2
In this scenario, the guarantee term GT2 has been violated so the corresponding penalty should be applied.			

Table 8.17: Test Case 10 for the each-choice strategy.

8.3.3.2 Discussion about the different test suites

So far, we have specified the scenarios associated to the test cases for the first of the applied testing strategies (*each-choice* testing). For the description of these scenarios we have used the natural language since the specification of the test cases is fully represented by the rows of the trees of Figure 8.11 - Figure 8.16. Due to this, in this section, instead of specifying the scenarios of the sets of 42 and 32 test cases obtained within the other two strategies (*pair-wise* and *hybrid* respectively) we will state some differences that arise from the derivation of these test suites.

With the generation of the test suite for the each-choice strategy, all the previously identified Primitive TRs are exercised at least once. The execution of the test cases

allows us to be able to detect specific problems in the eHealth system. For example, we are testing that both types of alarm arrive to the system, we assure that doctors and supervisors connected to the system with both wired and mobile devices are contacted during the execution of the test cases, we force that the doctors consult the measure of the existing medical devices and so on. However, there are other situations that are unexercised so potential problems could remain covered so a more exhaustive criterion may need to be applied.

The application of the second strategy (pairwise), more exhaustive than the first one, generates a larger set of test cases so the cost of generating and executing such test cases is also higher. However, the obtained test suite allows exercising new situations that may uncover hidden problems. Below, we will state some representative scenarios that are exercised within the test cases generated by applying the *pairwise* strategy (and also the *hybrid* strategy) and that remained unexercised with the *each-choice* strategy.

Firstly, there are some scenarios related to the invocation of the medical devices that we have to consider. As it can be seen in Figure 8.20 (a), in the set of Combined TRs of the *each-choice* strategy and, therefore, in the set of derived test cases, the two medical devices are never invoked within the same execution of the eHealth system (CL1 or CL2 of GT13 are never combined with CL1 or CL2 of GT14).

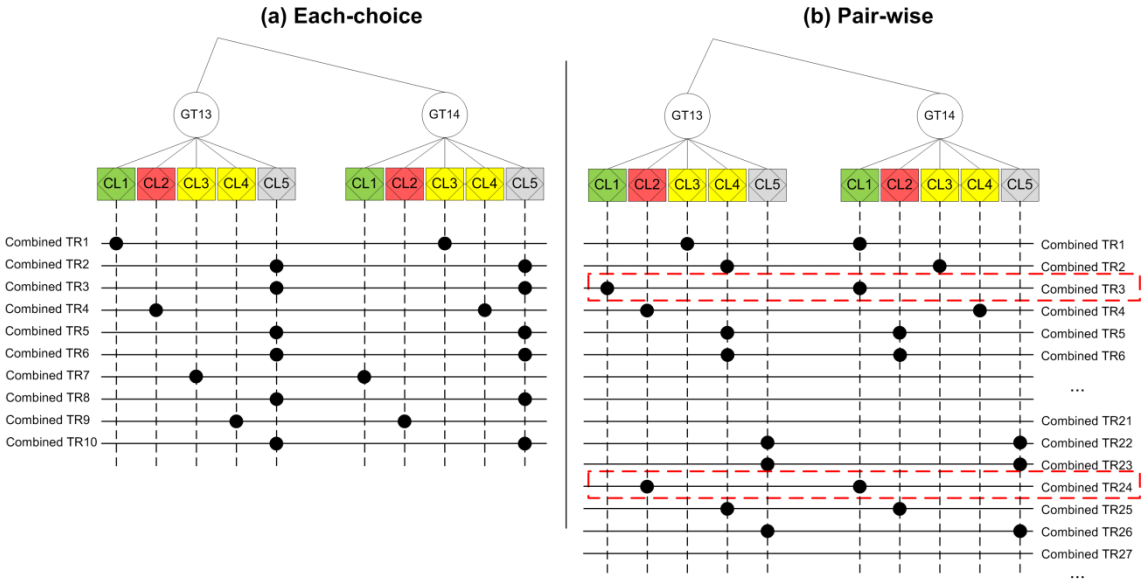


Figure 8.20: Invocation of both medical devices.

In the *pair-wise* strategy, we have obtained Combined TRs that address the invocation of both medical devices within the same execution of the system, for example, in the Combined TR3 and Combined TR24 remarked in Figure 8.20 (b). This means that there are specific test cases that exercise situations where both medical devices are consulted. These scenarios are represented using the natural language in Table 8.18 - Table 8.19. Furthermore, it is worth mentioning that such situations are also exercised in the test suite generated within the *hybrid* strategy. This is because the invocation of the medical devices is carried out when an Emergency arrives in the system and we have defined in the *hybrid* strategy that the guarantee terms associated to an Emergency are tested using pairwise testing.

Process	Expected Output	Eval. Values	PTRs exercised
An *Emergency* alarm arrives to the eHealth system.			
	...		
The doctor gets the measure of device_1.	The measure is provided in less than 3 seconds.	ev(GT13) = F	ex(GT13) = CL1
The doctor gets the measure of device_2.	The measure is provided in less than 10 seconds.	ev(GT14) = F	ex(GT14) = CL1
	...		

Table 8.18: Partial Test Case for the Combined TR3.

Process	Expected Output	Eval. Values	PTRs exercised
An *Emergency* alarm arrives to the eHealth system.			
	...		
The doctor gets the measure of device_1.	The measure is provided in more than 3 seconds.	ev(GT13) = V	ex(GT13) = CL2
The doctor gets the measure of device_2.	The measure is provided in less than 10 seconds.	ev(GT14) = F	ex(GT14) = CL1
	...		

Table 8.19: Partial Test Case for the Combined TR24.

In addition to these situations, there are more scenarios that are tested within the pairwise and hybrid strategies but not in the each-choice. For example, in the test suite generated in the each-choice strategy, there is only one test case that exercises the situation where a doctor connected to the system through a mobile device consults the

measure of the first medical device (see Combined TR4 of Figure 8.21 (a)). In that situation, both the doctor and the medical device spend more time to answer than the specified in the SLA (exercising CL2 of GT8 and CL2 of GT13).

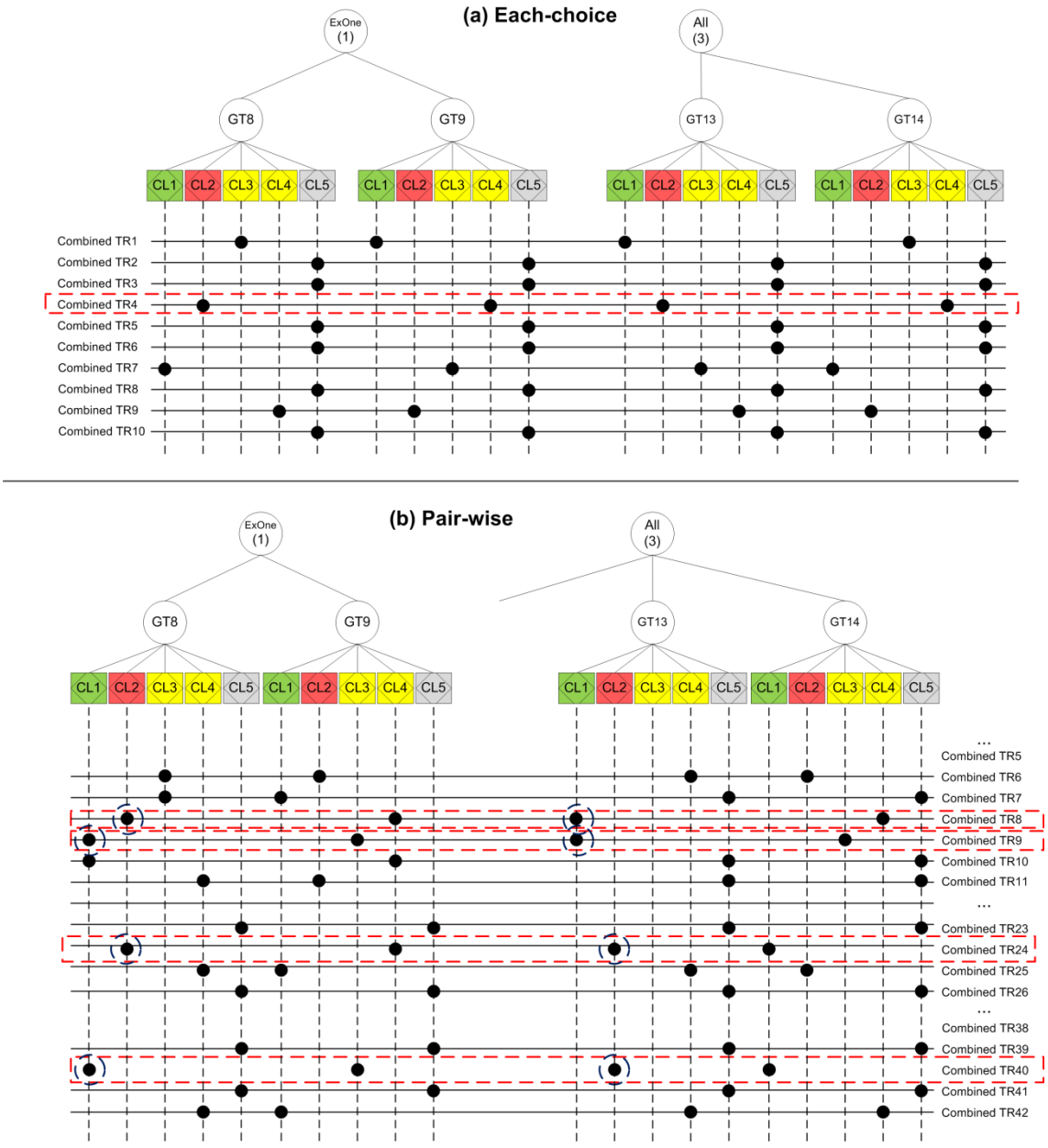


Figure 8.21: Invocation of doctors and medical devices.

In the pair-wise strategy, there are at least four test cases where a doctor connected through a mobile node consults the measure of the first device. These test cases exercise the situations represented in the Combined TR8, TR9, TR24 and TR40

(remarked in Figure 8.21 (b)). As can be seen, in these test cases we are testing the combinations where:

a) The doctor spends more time to answer than the specified in the SLA (exercising CL2 of GT8) whereas the medical devices spends less time that the allowed (exercising CL1 of GT13) -> Combined TR8.

b) Both the doctor and the medical device spend less time to answer than the specified in the SLA (exercising CL1 of GT8 and GT13) -> Combined TR9.

c) Both the doctor and the medical device spend more time to answer than the specified in the SLA (exercising CL2 of GT8 and GT13) -> Combined TR24.

d) The doctor spends less time to answer than the specified in the SLA (exercising CL1 of GT8) whereas the medical devices spends more time that the allowed (exercising CL2 of GT13) -> Combined TR40.

These scenarios are not considered within the each-choice strategy but they are exercised in both pair-wise and hybrid strategies. The same occurs with the doctor connected to the system through a wired device and the measurement provided by the second medical device.

Apart from this, there are other scenarios that remain unexercised in the less exhaustive strategies but not in the more exhaustive ones. These scenarios are related, for example, to the enquiry of the registry, which may answer with different response times and may provide empty and non-empty list of professionals. Other situations that are considered are those where the supervisors or the doctors are contacted when the registry has successfully provided its results or when a problem with such results has previously been detected.

In addition to the potential advantages of using different combinatorial strategies, the identification of test requirements guided by the four-valued logic allows testing specific scenarios of the service-based application. For example, the behavior of the SBA when a specific service is not executed is tested due to the Not Determined evaluation value. Another interesting test situations is when one of the medical devices

is queried whereas the other is not, exercising the Inapplicable evaluation value in the corresponding guarantee term.

8.4 Compositor Testing Level

In this second testing level, again we address the identification of new test requirements as well as the generation of the test cases. In this case, we focus on the logical relationships of the guarantee terms involved in the SLA. To do this, we use the ***SLACDC_Generator*** presented in Section 7.3, which implements the algorithms described in Section 6.2.3 that allow obtaining Combined Test Requirements fulfilling the SLACDC criterion.

8.4.1 Identification of Primitive Test Requirements

The identification of the Primitive Test Requirements is performed taking the potential evaluation values of a guarantee term into account. To be more specific and as we explained in Section 6.2.1, we identify one Primitive TR for each of the four evaluation values that a guarantee term can take.

As it is described in Section 8.2.2 of this chapter, the SLA that governs the executions of the services in the eHealth application contains 14 individual guarantee terms. Hence, we identify a set of 56 Primitive Test Requirements (14 GTs * 4 evaluation values / GT). These Primitive TRs will be combined in the following section depending on the compositor where each guarantee term is specified.

8.4.2 Derivation of Combined Test Requirements

The algorithms described in Section 6.2.3 have been automatically applied in order to obtain the initial set of Combined Test Requirements. The SLA includes 5 compositors that contain the 14 guarantee terms so 5 sets of Combined TRs are obtained, one for each compositor. The compositors of the SLA are represented in the first column of Table 8.20 and the number of initial Combined TRs for each compositor is represented in the second column.

Compositor	Initial	Rule1	Rule2		Rule3		Rule4		Total
		R	M	R	M	R	M	R	
All (1)	8	0	2	1	0	0	3	1	**6**
All (2)	17	2	5	4	4	2	6	1	**8**
ExOne (1)	13	0	4	3	0	0	4	4	**6**
ExOne (2)	13	0	4	3	0	0	4	4	**6**
All (3)	11	2	2	1	0	0	5	1	**7**
Total	**62**	4	17	12	4	2	22	11	**33**

Table 8.20: Combined Test Requirements in the Compositor Testing Level.

As we have previously stated, some of these test requirements may be non-feasible so the rules defined in Section 6.2.4 have also been applied. As a result, Table 8.20 also displays the number of Combined TRs that have been modified (M) or removed (R) after applying each rule (middle columns). Lastly, the last column outlines the final number of Combined TRs obtained for each compositor.

Initially, a set of 62 Combined Test Requirements are identified by applying the SLACDC criterion. These Combined TRs fulfil the conditions specified in such criterion, which assures that every Guarantee Term and every Compositor take the four potential evaluation values and the variation of any value affects the output of the evaluation. After that, the rules we have defined in Section 6.2.4 are automatically applied in order to avoid the obtaining of non-feasible combination of Primitive Test Requirements.

The set of Combined Test Requirements contains a total number of 33 requirements, which are represented in trees in Figure 8.22 - Figure 8.26.

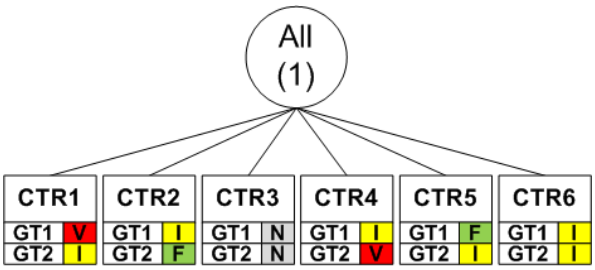


Figure 8.22: Combined Test Requirements of the All (1) Compositor.

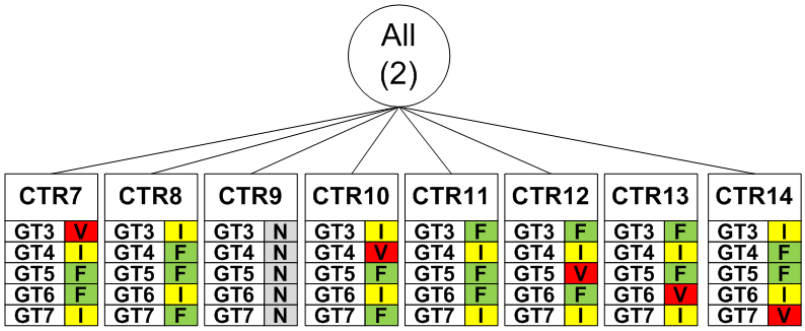


Figure 8.23: Combined Test Requirements of the All (2) Compositor.

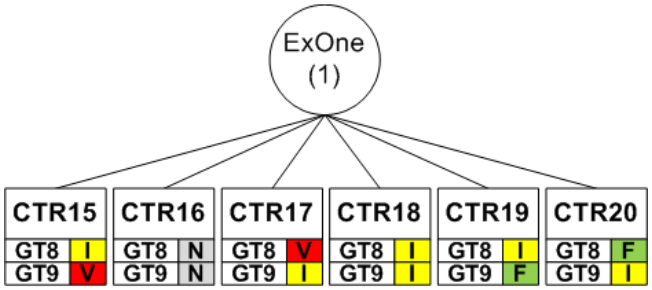


Figure 8.24: Combined Test Requirements of the ExactlyOne (1) Compositor.

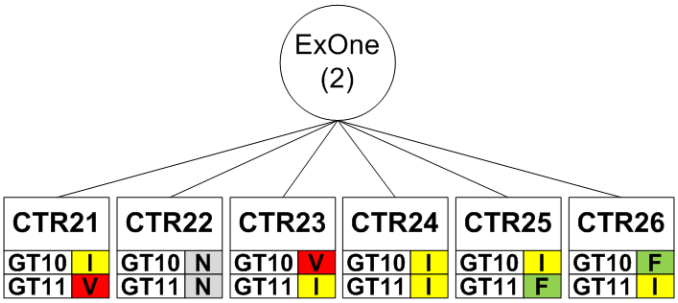


Figure 8.25: Combined Test Requirements of the ExactlyOne (2) Compositor.

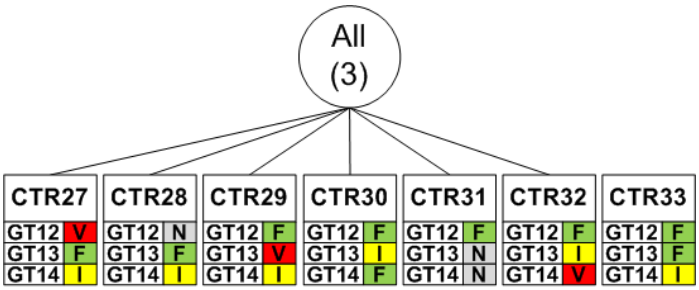


Figure 8.26: Combined Test Requirements of the All (3) Compositor.

This number is significantly lower than the number of Combined TRs we had obtained if we had applied a complete combination using the four-valued logic in each compositor. In that case, we had initially obtained a set of 1136 test requirements (4^n for

each compositor, where n is the number of involved GTs). From this set, we would also have to identify the non-feasible Combined TRs and modify or remove them accordingly

8.4.3 Generation of Test Cases

In this section we address the generation of test cases by means of composing the Combined Test Requirements in order to obtain a complete scenario of the SBA. This is a task that is manually done by the tester, taking some issues into account.

First, the objective is to obtain a test suite that exercises all the Combined TRs with the least number of test cases. Hence, when generating a new test case the tester tries to exercise Combined TRs that have not been exercised in previous test cases. This means that we are somehow applying each-choice testing with combinations of test requirements that have previously been elaborated (the Combined Test Requirements).

Second, depending on the behaviour of the SBA there may be combinations of Combined TRs that may be non-feasible. Thus, the tester must carefully select the Combined TRs to be composed within a test case.

Third, the Combined TRs of Figure 8.22 - Figure 8.26 were identified using the SLACDC criterion. This means that such identification is based in the content of the SLA but it does not take into account aspects related to the behaviour of the SBA. This implies that some of the Combined TRs may represent situations that cannot be exercised in the SBA so the tester should adapt or remove such test requirements. Furthermore and due to the same reason, when generating a test case it may be necessary to create a new Combined TR in order to obtain a feasible complete scenario to be tested in the SBA.

Bearing these considerations in mind, during the generation of the test cases we become aware that seven Combined TRs represent non-feasible situations regarding the SBA. Below we describe the reasons of each Combined TR.

- The Combined Test Requirement CTR3 requires the non-execution of the eHealth system. This invocation is mandatory to manage the arrival of an alarm

so the non-invocations of this service are situations that cannot be tested. Thus, we have removed this Combined TR.

- The Combined Test Requirements CTR6 requires that an alarm with a different type of Emergency and No Confirmation arrives at the system, which is not allowed because the system only receives alarms of such types. In this case, one test case is designed to test how the system behaves when a non-accepted alarm arrives.

- The Combined Test Requirements CTR18 and CTR24 are identified to test the management of an alarm by a doctor or supervisor which is not connected to the system through a wired or mobile device. As it is specified that these are the only two ways to be connected to the system, the situations exercised by CTR18 and CTR24 are non-feasible. Furthermore, these two CTRs were obtained in order to evaluate the corresponding compositors as Inapplicable. Thus, they are not designed to test a specific change in the evaluation value of specific guarantee terms so we have removed such test requirements from the set of Combined TRs.

- In the Combined Test Requirement CTR12 the violation of the GT5 implies that the registry provides an empty list of professionals as output. However, the fulfilment of GT6 requires that the list of professionals provided by the registry must contain only doctors. As the combination of these two situations becomes non-feasible, the Combined TR12 has to be adapted or removed. The violation of GT5 implies that we need to make a decision about the evaluation of GT6 when the list of professionals provided by the registry is empty. In this case, we have decided that this test requirement will not be adapted and we have also removed it. However, we have to take into account that the aforementioned situation is really interesting and should be tested, disregarding the evaluation value of GT6.

- Finally, the Combined Test Requirement CTR28 requires that the doctor does not consult the list of medical devices deployed in the patient's home so, in that situation, such doctor is not capable to get the measures from the devices. As

this test requirement aims at testing the non-execution of the service that provides the list of medical devices, we have adapted the Combined TR so, now, it also implies the non-execution of the services specified in GT13 and GT14 (Figure 8.27).

All (3)	GT12	GT13	GT14
CTR28	N	N	N

Figure 8.27: Modified CTR28.

With this modified set of Combined TRs, we have derived eleven test cases that exercise all the resultant Combined TRs by means of applying *each-choice* testing to the compositors. The exercitation of the Combined TRs in each test case is represented in Table 8.21. The identifier of each test case is showed in the first column. In the rest columns the exercised Combined TRs from each compositor are also outlined.

Test Case	All (1)	All(2)	ExOne(1)	ExOne(2)	All(3)
TC1	CTR1	CTR7	CTR15	CTR22	CTR27
TC2	CTR5	CTR11	CTR17	CTR22	CTR29
TC3	CTR2	CTR8	CTR16	CTR21	CTR28
TC4	CTR4	CTR10	CTR16	CTR23	CTR28
TC5	CTR5	CTR11	CTR20	CTR22	CTR30
TC6	CTR5	CTR13	CTR19	CTR22	CTR31
TC7	CTR2	CTR14	CTR16	CTR25	CTR28
TC8	CTR5	CTR11	CTR20	CTR22	CTR32
TC9	CTR2	CTR8	CTR16	CTR26	CTR28
TC10	CTR5	CTR11	CTR20	CTR22	CTR33
TC11	CTR6	CTR9	CTR16	CTR22	CTR28

Table 8.21: Combined TRs exercised in each test case.

The Combined Test Requirements exercised in each test case are also specified in trees in Figure 8.28 and Figure 8.29.

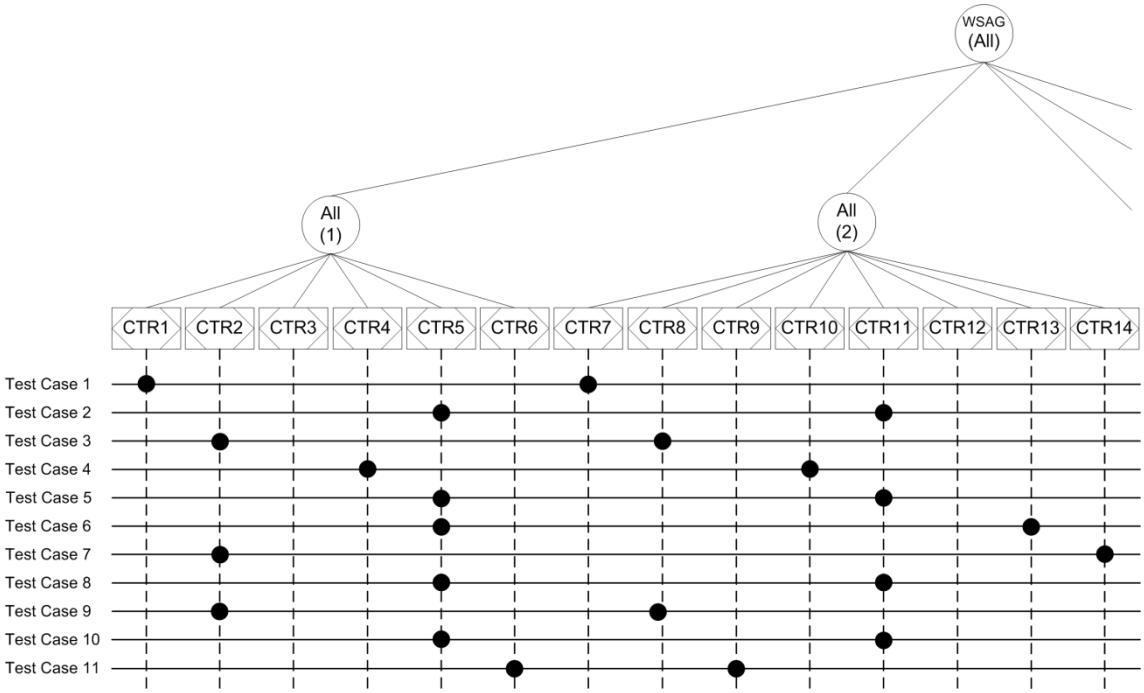


Figure 8.28: Combined Test Requirements – Test Cases tree (I).

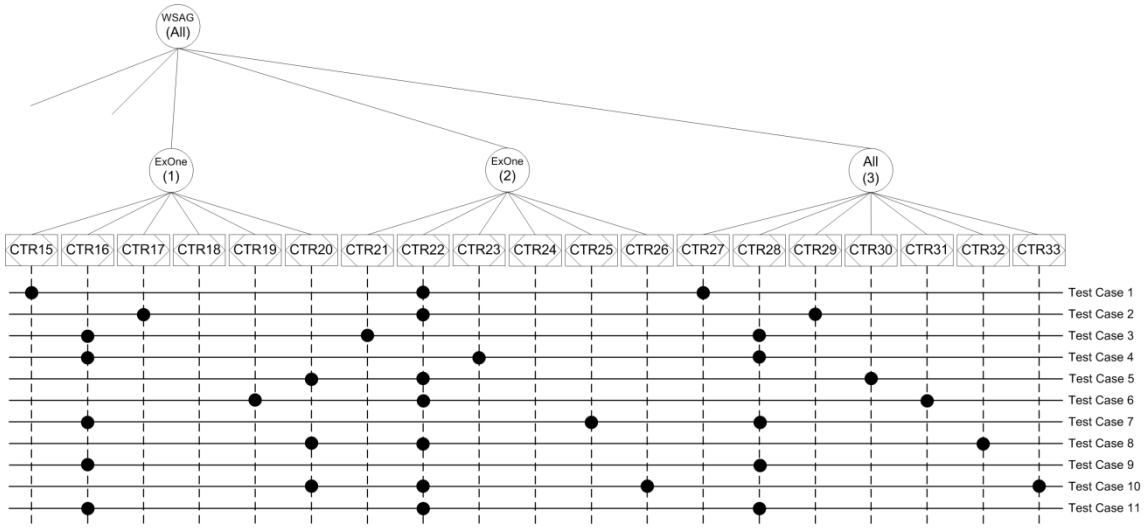


Figure 8.29: Combined Test Requirements – Test Cases tree (II).

The specifications of all the test cases are described in the following Table 8.22 - Table 8.32. The process for testing the scenario represented in such test case is described in the first column. The expected output is showed in the second column. The values inherent to the evaluation of the guarantee terms are represented in the third column. Finally, the Combined TRs exercised within the test case are represented in the last column.

Process	Expected Output	Eval. Values	CTR exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals spending more than 3 seconds. The list contains only doctors.	ev(GT3) = V ev(GT4) = I ev(GT5) = F ev(GT6) = F ev(GT7) = I	CTR7
A doctor connected to the system through a wired device is contacted.	(s)he provides a response spending more than 2 seconds.	ev(GT8) = I ev(GT9) = V	CTR15
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	CTR22
The doctor finds the list of devices deployed in the patient's home.	This invocation spends more than 2 seconds.	ev(GT12) =V	CTR27
The doctor gets the measure of device_1.	The measure is provided in less than 3 seconds.	ev(GT13) = F	
The doctor does not consult the measure of device_2.		ev(GT14) = I	
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending more than 300 seconds.	ev(GT1) = V ev(GT2) = I	CTR1
In this scenario, the guarantee terms GT1, GT3, GT9 and GT12 have been violated so the corresponding penalties should be applied.			

Table 8.22: Test Case 1 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 3 seconds. The list contains only doctors.	ev(GT3) = F ev(GT4) = I ev(GT5) = F ev(GT6) = F ev(GT7) = I	CTR11
A doctor connected to the system through a mobile device is contacted.	(s)he provides a response spending more than 8 seconds.	ev(GT8) = V ev(GT9) = I	CTR17
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	CTR22
The doctor finds the list of devices deployed in the patient's home.	This invocation in less than 2 seconds.	ev(GT12) =F	CTR29
The doctor gets the measure of device_1.	The measure is provided in more than 3 seconds.	ev(GT13) = V	
The doctor does not consult the measure of device_2.		ev(GT14) = I	
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 300 seconds.	ev(GT1) = F ev(GT2) = I	CTR5
In this scenario, the guarantee terms GT8 and GT13 have been violated so the corresponding penalties should be applied.			

Table 8.23: Test Case 2 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 6 seconds. The list contains only supervisors.	ev(GT3) = I ev(GT4) = F ev(GT5) = F ev(GT6) = I ev(GT7) = V	CTR8
A supervisor connected to the system through a wired device is contacted.	(s)he provides a response in more than 15 seconds.	ev(GT10) = I ev(GT11) = V	CTR21
As a supervisor has been contacted, no doctors are invoked.		ev(GT8) = N ev(GT9) = N	CTR16
As the alarm is managed by a supervisor, no medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	CTR28
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 600 seconds.	ev(GT1) = I ev(GT2) = F	CTR2
In this scenario, the guarantee terms GT7 and GT11 have been violated so the corresponding penalties should be applied.			

Table 8.24: Test Case 3 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals spending more than 6 seconds. The list contains only supervisors.	ev(GT3) = I ev(GT4) = V ev(GT5) = F ev(GT6) = I ev(GT7) = F	CTR10
A supervisor connected to the system through a mobile device is contacted.	(s)he provides a response in less than 20 seconds.	ev(GT10) = V ev(GT11) = I	CTR23
As a supervisor has been contacted, no doctors are invoked.		ev(GT8) = N ev(GT9) = N	CTR16
As the alarm is managed by a supervisor, no medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	CTR28
	After having carried out all of these tasks, the eHealth system provides a response to the patient spending more than 600 seconds.	ev(GT1) = I ev(GT2) = V	CTR4
In this scenario, the guarantee terms GT2, GT4 and GT10 have been violated so the corresponding penalties should be applied.			

Table 8.25: Test Case 4 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 3 seconds. The list contains only doctors.	ev(GT3) = F ev(GT4) = I ev(GT5) = F ev(GT6) = F ev(GT7) = I	CTR11
A doctor connected to the system through a mobile device is contacted.	(s)he provides a response in less than 8 seconds.	ev(GT8) = F ev(GT9) = I	CTR20
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	CTR22
The doctor finds the list of devices deployed in the patient's home.	This invocation in less than 2 seconds.	ev(GT12) =F	CTR30
The doctor does not consult the measure of device_1.		ev(GT13) = I	
The doctor gets the measure of device_2.	The measure is provided in more than 10 seconds.	ev(GT14) = F	
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 300 seconds.	ev(GT1) = F ev(GT2) = I	CTR5
In this scenario, none of the guarantee terms has been violated so no consequences are derived.			

Table 8.26: Test Case 5 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 3 seconds. The list contains both doctors and supervisors.	ev(GT3) = F ev(GT4) = I ev(GT5) = F ev(GT6) = V ev(GT7) = I	CTR13
A doctor connected to the system through a mobile device is contacted.	(s)he provides a response in less than 2 seconds..	ev(GT8) = I ev(GT9) = F	CTR19
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	CTR22
The doctor finds the list of devices deployed in the patient's home.	This invocation in less than 2 seconds.	ev(GT12) =F	
The doctor does not consult the measure of device_1.		ev(GT13) = N	CTR31
The doctor does not consult the measure of device_2.		ev(GT14) = N	
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 300 seconds.	ev(GT1) = F ev(GT2) = I	CTR5
In this scenario, the guarantee term GT6 has been violated so the corresponding penalty should be applied.			

Table 8.27: Test Case 6 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 6 seconds. The list contains both doctors and supervisors.	ev(GT3) = I ev(GT4) = F ev(GT5) = F ev(GT6) = I ev(GT7) = V	CTR14
A supervisor connected to the system through a wired device is contacted.	(s)he provides a response in less than 15 seconds.	ev(GT10) = I ev(GT11) = F	CTR25
As a supervisor has been contacted, no doctors are invoked.		ev(GT8) = N ev(GT9) = N	CTR16
As the alarm is managed by a supervisor, no medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	CTR28
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 600 seconds.	ev(GT1) = I ev(GT2) = F	CTR2
In this scenario, the guarantee term7 has been violated so the corresponding penalty should be applied.			

Table 8.28: Test Case 7 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 3 seconds. The list contains only doctors.	ev(GT3) = F ev(GT4) = I ev(GT5) = F ev(GT6) = F ev(GT7) = I	CTR11
A doctor connected to the system through a mobile device is contacted.	(s)he provides a response in less than 8 seconds.	ev(GT8) = F ev(GT9) = I	CTR20
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	CTR22
The doctor finds the list of devices deployed in the patient's home.	This invocation in less than 2 seconds.	ev(GT12) =F	CTR32
The doctor does not consult the measure of device_1.		ev(GT13) = I	
The doctor gets the measure of device_2.	The measure is provided in more than 10 seconds.	ev(GT14) = V	
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 300 seconds.	ev(GT1) = F ev(GT2) = I	CTR5
In this scenario, the guarantee term GT14 has been violated so the corresponding penalty should be applied.			

Table 8.29: Test Case 8 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
A *No Confirmation* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 6 seconds. The list contains only supervisors.	ev(GT3) = I ev(GT4) = F ev(GT5) = F ev(GT6) = I ev(GT7) = F	CTR8
A supervisor connected to the system through a mobile device is contacted.	(s)he provides a response in less than 20 seconds.	ev(GT10) = F ev(GT11) = I	CTR26
As a supervisor has been contacted, no doctors are invoked.		ev(GT8) = N ev(GT9) = N	CTR16
As the alarm is managed by a supervisor, no medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	CTR28
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 600 seconds..	ev(GT1) = I ev(GT2) = F	CTR2
In this scenario, none of the guarantee terms has been violated so no consequences are derived.			

Table 8.30: Test Case 9 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
An *Emergency* alarm arrives to the eHealth system.			
The registry is queried.	It provides a non-empty list of professionals in less than 3 seconds. The list contains only doctors.	ev(GT3) = F ev(GT4) = I ev(GT5) = F ev(GT6) = F ev(GT7) = I	CTR11
A doctor connected to the system through a mobile device is contacted.	(s)he provides a response in less than 8 seconds.	ev(GT8) = F ev(GT9) = I	CTR20
As a doctor has been contacted, no supervisors are invoked.		ev(GT10) = N ev(GT11) = N	CTR22
The doctor finds the list of devices deployed in the patient's home.	This invocation in less than 2 seconds.	ev(GT12) =F	CTR33
The doctor gets the measure of device_1.	The measure is provided in more than 3 seconds.	ev(GT13) = V	
The doctor does not consult the measure of device_2.		ev(GT14) = I	
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 300 seconds.	ev(GT1) = F ev(GT2) = I	CTR5
In this scenario, the guarantee term GT13 has been violated so the corresponding penalty should be applied.			

Table 8.31: Test Case 10 in the Compositor Testing Level.

Process	Expected Output	Eval. Values	CTR exercised
An alarm that is not an Emergency or Not Confirmation arrives to the eHealth system.			
As the type of the alarm is not allowed, the registry is not queried.		ev(GT3) = N ev(GT4) = N ev(GT5) = N ev(GT6) = N ev(GT7) = N	CTR9
No doctors are invoked.		ev(GT8) = N ev(GT9) = N	CTR16
No supervisors are invoked.		ev(GT10) = N ev(GT11) = N	CTR22
No medical devices are consulted.		ev(GT12) =N ev(GT13) = N ev(GT14) = N	CTR28
	After having carried out all of these tasks, the eHealth system provides a response to the patient in less than 300 seconds.	ev(GT1) = I ev(GT2) = I	CTR6
In this scenario, none of the guarantee terms has been violated so no consequences are derived.			

Table 8.32: Test Case 11 in the Compositor Testing Level.

The final set of Combined TRs contains 29 test requirements. The compositor that implies the generation of a higher number of Combined TRs is the *All* (3). For this compositor, seven Combined TRs were identified (CTR27-CTR33). In the final test suite, we have derived eleven cases to cover all the resultant Combined Test Requirements. This means that some of the Combined TRs are exercised more than once within the test suite, which is somehow necessary in order to compose complete scenarios to be tested.

8.5 Summary

In this chapter we have designed tests for an eHealth case study proposed in a European FP7 Project. Taking the eHealth system as our Software Under Test (SUT), we have applied the testing techniques developed in the two testing level described in Chapter 5 and Chapter 6. In each of these levels and making use of the tool support presented in Chapter 7, we have identified a set of test requirements and, later, we have generated the test cases that exercised such test requirements.

To be more specific, in the Guarantee Term Testing Levels we have obtained different test suites by means of selecting three different coverage strategies. The size of these test suites is 10 test cases for the least exhaustive strategy, 42 test cases for the most exhaustive and 32 test cases for the hybrid one, which is an intermediate strategy between the previous two.

In the Compositor Testing Level we have generated a final set of 11 test cases that exercise all the Combined Test Requirements identified by means of the application of the SLACDC criterion.

In the following and last chapter of this dissertation, we will state the conclusions of the work developed under this PhD and we will also outline potential directions of our future work.

Chapter 9

Conclusions

You have to know the past to understand the present

Carl Sagan, 1934-1996
American astronomer, cosmologist and science communicator

This chapter presents the conclusions of this dissertation. Firstly it highlights the main contributions of this work. After that, it discusses the main limitations and outlines potential research lines for future work.

9.1 Synthesis and Results

In the field of Service-Based Applications (SBAs), a Service Level Agreement (SLA) contains the conditions that must be fulfilled by the provider and the consumer during the executions of the services. In this context, many works have been proposed to detect whether the behaviour of the SBA has violated the SLA during its executions at runtime. When this occurs, different type of penalties may be applied in order to somehow compensate the consequences of the SLA violation. Few works have focused on the early detection of problems before the SBA is deployed in the operational environment. In this thesis, we have addressed the testing of SLAs by means of proposing a proactive approach that allows anticipating the detection of problems in the SBA so consequences derived from SLA violations can be avoided or minimized.

The first contribution of this dissertation is the design of **SLATF (SLA Testing Framework)**, which defines a testing process that takes the specification of the SLA as the test basis (Chapter 3). This framework involves the development of different activities with the objective of testing SLA-aware SBAs.

All the activities presented in SLATF need to take the evaluation of the SLA into account. This evaluation allows, on the one hand, determining whether the executions of the SBA fulfil the conditions agreed in the SLA. On the other hand, the evaluation is also used to identify potential situations that can cause problems in the SBA. In this context, the second contribution of this dissertation is **a four-valued logic that allows evaluating the SLA and its internal elements, including guarantee terms and compositors** (Chapter 4).

The testing process of SLATF requires identifying test requirements from the specification of the SLA and, later, deriving the test cases that cover such requirements. In this thesis, we have established **two different testing levels**, depending whether the tests are obtained from the content of the individual guarantee terms (Guarantee Term Testing Level) or from the logical relationships of such terms (Compositor Testing Level). In the first of these levels, we make use of standard **combinatorial testing techniques** in order to identify the test requirements and generate the final set of test cases (Chapter 5). In the second level, we devise **SLACDC (SLA Condition Decision**

Coverage), a coverage-based criterion that allows identifying test requirements that exercise interesting combinations of situations regarding the logical relationships between the terms of the SLA (Chapter 6).

The testing techniques proposed in the two aforementioned testing levels have been automated in order to reduce the cost and effort necessary to design the tests. Firstly, we have developed the **SLACT (SLA Combinatorial Testing)** tool, which allows us to automatically identify and combine the test requirements from the specification of an SLA using the WS-Agreement language (Chapter 7). Furthermore, we have also developed a **prototype** that automates the identification of new test requirements by using SLACDC.

Finally, we have evaluated the feasibility of our approaches using an **eHealth scenario** proposed in the context of a FP7 European Project (Chapter 8). The outcomes of this case study show that it is possible to obtain reduced set of test cases that exercise different situations regarding the SLA-aware service based application.

9.2 Discussion, limitations and extensions

In this section we identify some of the decisions we have made in this dissertation and we outline its main limitations. These considerations guide us to highlight motivating topics to be explored in our future work.

WS-Agreement as a cornerstone

In this thesis, we have used the WS-Agreement standard language in order to specify the SLAs that are taken as the test basis. In spite of the fact that many languages have been proposed to standardize the specification of SLAs [108], for example, WSLA, WSLO, SLANG, WS-QoS or WS-Policy, it has been WS-Agreement who has received more attention regarding the testing of SLAs, at least from the academic scope. As WS-Agreement presents a generic syntax, we envision that the outcomes derived from this dissertation could be extrapolated to any other existing SLA specification language.

Future work: generalize the proposed testing techniques and provide guidelines for the application of such techniques with other SLAs languages.

Proactive Approach

Since the beginning, this dissertation focuses on proposing a proactive approach in order to anticipate the detection of problems and, thus, avoid or mitigate the consequences derived from an SLA violation. To do this, one of the main tasks of this PhD involves the identification of test requirements, which represent the situations that are more interesting to be tested. We consider that these test requirements could also be used to guide the design of monitoring plans, helping to decide which aspects of the SBA need to be monitored.

Future work: find and explode the synergies between our proactive approach and the existing SLA monitoring-based approaches.

Simple conditions in the Qualifying Condition and the Service Level Objective

In our approach, we consider the content of the QC and the SLO as a whole, without analysing the internal conditions of both elements. Hence, we say that the QC (or the SLO) is satisfied or not but we do not take into account whether the QC (or the SLO) contains a more complex expression that needs to be analysed.

Future work: study how the complexity of the internal elements of a guarantee term affects the definition of new and more accurate tests.

Simple hierarchy of the SLA

In our approach, we are considering that the SLA is composed of compositors that, likewise, contain guarantee terms. However, this hierarchy could be more complex if we add new levels of nesting, for example, dealing with compositors that contain both compositors and guarantee terms.

Future work: study how our approach should operate in case of several levels of imbrications.

Reuse of already designed tests

In both testing levels designed in this PhD, we use the SLA as the test basis to design the tests. From the content of the SLA, we use the developed tools to obtain the set of test requirements that will be later exercised through the test cases. However, a change in the specification of the SLA (even if it is a minor change) affects the identification of the tests so a new set of test requirements needs to be identified and, consequently, new test cases are generated without reusing the previous ones.

Future work: study how previous test designs could be reused when the specification of the SLA changes.

Automation of test cases

In this dissertation we obtain the specification of the test cases by means of identifying the Primitive TRs and the Combined TRs. From this specification, each test case needs to be manually prepared and executed.

Future work: study how to automate the test cases, probably using an activity-based or state-based model of the system.

Chapter 10

Conclusiones

There will come a time when you believe everything is finished.
That will be the beginning.

Louis Dearborn L'Amour, 1908-1988
American writer

Este capítulo presenta las conclusiones de esta tesis. En primer lugar se resumen las principales contribuciones del trabajo realizado. Después se discuten las limitaciones de los métodos de prueba propuestos y se esbozan potenciales líneas de investigación para el trabajo futuro.

10.1 Resumen y resultados

En el ámbito de las aplicaciones basadas en servicios (SBAs – Service-Based Applications), un Acuerdo de Nivel de Servicio (SLA – Service Level Agreement) contiene las condiciones que han de ser cumplidas tanto por el proveedor de los servicios como por el consumidor durante la ejecución de los mismos. En este contexto, mucho ha sido el trabajo dedicado a detectar si el comportamiento observador durante las ejecuciones de la aplicación viola las condiciones especificadas en el SLA. Cuando esto ocurre, diferentes tipos de penalización pueden ser aplicadas para compensar las consecuencias derivadas de dicha violación. Desafortunadamente, pocos trabajos se han orientado a anticipar la detección de las violaciones del SLA antes de que la aplicación haya sido desplegada en su entorno operacional. En esta tesis se ha investigado la prueba de los SLAs mediante un enfoque proactivo que permite anticipar la detección de problemas en la aplicación bajo prueba y, por tanto, se contribuye a evitar o mitigar las consecuencias derivadas de las violaciones del SLA.

La primera contribución de esta tesis es el diseño de **SLATF (SLA Testing Framework)**, un marco de trabajo que define un proceso de pruebas que toma la especificación del SLA como entrada (Chapter 3). Este marco de trabajo implica el desarrollo de diferentes actividades que tienen como objetivo la prueba de aplicación basadas en servicio que tienen asociado un acuerdo de nivel de servicio.

Todas las actividades presentes en SLATF necesitan tener en cuenta la evaluación del SLA. Esta evaluación permite, por un lado, determinar si las ejecuciones de la aplicación bajo prueba están cumpliendo las condiciones acordadas en el SLA. Por otra parte, la evaluación también es usada para identificar potenciales situaciones que pueden causar problemas en la aplicación. En este contexto, la segunda contribución de esta tesis es la presentación de **una lógica cuatrivaluada que permite evaluar el SLA y sus elementos internos**, incluyendo tanto los términos de garantía individuales como las combinaciones lógicas de dichos términos (Chapter 4).

El proceso de pruebas implementado por SLATF requiere la identificación de requisitos de prueba a partir de la especificación del SLA y, posteriormente, derivar los casos de prueba que ejercitan dichos requisitos. En esta tesis, hemos establecido **dos**

niveles de prueba para ello, dependiendo si las pruebas se obtienen a partir del contenido de cada término de garantía (Guarantee Term Testing Level) o de sus combinaciones lógicas (Compositor Testing Level). En el primero de dichos niveles usamos **diferentes técnicas de pruebas combinatorias** para identificar los requisitos y generar el conjunto final de casos de prueba (Chapter 5). En el segundo de los niveles desarrollamos **SLACDC (SLA Condition Decision Coverage)**, un criterio de pruebas basado en cobertura que permite identificar requisitos de prueba que ejercitan combinaciones interesantes de situaciones en relación a las relaciones lógicas entre los términos de garantía del SLA (Chapter 6).

Las técnicas de pruebas propuestas en los citados dos niveles han sido automatizadas para reducir el coste y esfuerzo necesario para diseñar las pruebas. En primer lugar, hemos desarrollado la herramienta **SLACT (SLA Combinatorial Testing)**, la cual automatiza la identificación y combinación de los requisitos de prueba a partir de un SLA especificado usando el lenguaje WS-Agreement (Chapter 7). Además, también hemos desarrollado un prototipo que automatiza la identificación de requisitos de prueba aplicando el criterio de cobertura SLACDC.

Finalmente hemos evaluado el enfoque desarrollado usando un **escenario de tele-asistencia médica** que ha sido propuesto en el contexto de un Proyecto Europeo del Séptimo Programa Marco (Chapter 8). Los resultados de este caso de estudio indican que es posible obtener un conjunto reducido de casos de prueba que ejercitan diferentes situaciones asociadas a la aplicación bajo prueba.

10.2 Discusión, limitaciones y trabajo futuro

En esta sección se discuten algunas de las decisiones que han sido tomadas en esta tesis y se indican sus principales limitaciones. Este análisis nos permite guiar las líneas de trabajo a seguir en el futuro.

WS-Agreement usado como lenguaje de especificación de SLAs

En esta tesis se ha usado WS-Agreement como el lenguaje estándar para especificar los SLAs que son tomados como entrada al proceso de pruebas. A pesas de que muchos otros lenguajes han sido propuestos para estandarizar la especificación del

SLA, como por ejemplo WSLA, WSLO, SLANG, WS-QoS o WS-Policy, ha sido WS-Agreement el que ha recibido más atención en el contexto de las pruebas del software, al menos desde un punto de vista académico. WS-Agreement presenta una sintaxis genérica para especificar un SLA y, por tanto, entendemos que los resultados obtenidos podrían ser extrapolados a otros lenguajes existentes de SLAs.

Trabajo Futuro: generalizar las pruebas de testing propuestas y proporcionar unas pautas para la aplicación de dichas técnicas con otros lenguajes de SLAs.

Enfoque Proactivo

Desde sus inicios, esta tesis se ha enfocado a proponer un enfoque proactivo con el objetivo de anticipar la detección de problemas y, por consiguiente, contribuir a evitar o mitigar las consecuencias derivadas de una violación del SLA. Para ello, una de las principales tareas de esta tesis es la identificación de requisitos de prueba, los cuales representan situaciones que son interesantes de ejercitar desde el punto de vista de las pruebas. Consideramos que estos requisitos de prueba podrían ser también usados para guiar el diseño y preparación de diferentes planes de monitorización, ayudando a decidir qué aspectos de la aplicación bajo prueba son más importantes y necesitan ser observador.

Trabajo Futuro: analizar y explotar las sinergias entre nuestro enfoque proactivo y los enfoques de pruebas reactivos basados en la monitorización de los SLAs.

Condiciones simples tanto en la Qualifying Condition como en los Service Level Objectives.

En nuestro enfoque se tiene en cuenta el contenido de la Qualifying Condition (QC) y de los Service Level Objectives (SLOs) como un todo, sin analizar las condiciones internas de dichos elementos. De esta forma, decimos que la QC (o el SLO) se cumplen o no se cumplen, pero no tenemos en cuenta si ambos elementos contienen expresiones más complejas que necesitarían ser analizadas con mayor grado de detalle.

Trabajo Futuro: estudiar cómo la complejidad de los elementos internos de un término de garantía afecta a la definición de nuevas y más precisas pruebas.

Jerarquía simple de los SLAs

En esta tesis se considera que los SLAs están compuestos de compositores que, a su vez, contienen términos de garantía. Sin embargo, esta jerarquía podría ser más compleja si se incorporan nuevos niveles de anidamiento como, por ejemplo, incluyendo compositores dentro de otros compositores.

Trabajo Futuro: estudiar cómo nuestra aportación debería ser adaptada para contemplar la posibilidad de niveles de anidación complejos.

Reutilización de pruebas ya diseñadas.

En los dos niveles de pruebas propuestos se usa el SLA como entrada para diseñar las pruebas. A partir del contenido del SLA hacemos uso de las herramientas desarrolladas para obtener el conjunto de requisitos de prueba que serán ejercitados más tarde mediante la ejecución de los casos de prueba. Sin embargo, un cambio en la especificación de SLA (incluso siendo una modificación menor) afecta a la identificación de las pruebas por lo que un nuevo conjunto de requisitos de prueba necesita ser obtenido. Consecuentemente nuevos conjuntos de casos de prueba son generados sin tener en cuenta los anteriores.

Trabajo Futuro: estudiar cómo se pueden reutilizar los casos de prueba ya diseñados cuando la especificación del SLA cambia.

Automatización de los casos de prueba

En esta tesis se obtiene la especificación de los casos de prueba mediante la identificación de los Primitive Test Requirements y los Combined Test Requirements. Posteriormente y a partir de dicha especificación, cada caso de prueba necesita ser manualmente preparado para llevar a cabo su ejecución.

Trabajo Futuro: estudiar cómo dichos casos de prueba pueden ser automatizados, posiblemente usando un modelo del sistema basado en actividades o estados.

Institutional Acknowledgments

This dissertation has been partially funded by the Department of Science and Innovation (Spain) and ERDF funds within the National Program for Research, Development and Innovation, project **Test4SOA (TIN2007-67843-C06-01)**, project **Test4DBS (TIN2010-20057-C03-01)** and **FICYT** (Government of the Principality of Asturias) **Grant BP09-075.**

Appendix 1: eHealth SLA

This SLA can be publicly downloaded in [125].

```
<?xml version="1.0" encoding="UTF-8"?>

<wsag:AgreementOffer
xmlns:tns="http://www.w3.org/2005/08/addressing"
xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"
xmlns:wsrf-bf="http://docs.oasis-open.org/wsrf/bf-2.xsd"
xmlns:xml="http://www.w3.org/XML/1998/namespace"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SLATest="http://giis.uniovi.es/SLATest"
xsi:schemaLocation="http://schemas.ggf.org/graap/2007/03/ws-
agreement">

  <wsag:Name>eHealth Scenario</wsag:Name>

  <wsag:Context>
    <wsag:ServiceProvider>AgreementInitiator</wsag:ServiceProvider>
    <wsag:ExpirationTime>2013-12-31T00:00:00</wsag:ExpirationTime>
  </wsag:Context>

  <wsag:Terms>

    <wsag:All>

    <wsag:All>

        <wsag:GuaranteeTerm wsag:Name="GT1"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSHealth">
                <SLATest:Method>
                    <NameMethod>reportAlarm</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>alarmType</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>Emergency</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>lt</SLATest:operator>
                    <SLATest:constant>300</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
```

```
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>10</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT2"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSHealth">
                <SLATest:Method>
                    <NameMethod>reportAlarm</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>alarmType</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>No Confirmation</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>lt</SLATest:operator>
                    <SLATest:constant>600</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>10</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

    </wsag:All>

    <wsag:All>

        <wsag:GuaranteeTerm wsag:Name="GT3"
wsag:Obligated="ServiceProvider">
```

```
            <wsag:ServiceScope wsag:ServiceName="WSRegistry">
                <SLATest:Method>
                    <NameMethod>getConnectedDeviceIP</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>alarmType</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>Emergency</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>lt</SLATest:operator>
                    <SLATest:constant>3</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>3</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT4"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSRegistry">
                <SLATest:Method>
                    <NameMethod>getConnectedDeviceIP</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>alarmType</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>No Confirmation</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>lt</SLATest:operator>
                    <SLATest:constant>6</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>
```

```
            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>1</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT5"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSRegistry">
                <SLATest:Method>
                    <NameMethod>getConnectedDeviceIP</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>

    <SLATest:variable>count(list_of_professionals)</SLATest:variable>
                    <SLATest:operator>gt</SLATest:operator>
                    <SLATest:constant>0</SLATest:constant>
                    <SLATest:unit>professionals</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>5</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT6"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSRegistry">
                <SLATest:Method>
                    <NameMethod>getConnectedDeviceIP</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>alarmType</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>Emergency</SLATest:constant>
            </wsag:QualifyingCondition>
```

```
            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:expression>for all i
(list_of_professionals(i) = doctor) </SLATest:expression>

                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList/>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT7"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSRegistry">
                <SLATest:Method>
                    <NameMethod>getConnectedDeviceIP</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>alarmType</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>No Confirmation</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:expression>for all i
(list_of_professionals(i) = supervisor) </SLATest:expression>

                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList/>
        </wsag:GuaranteeTerm>

    </wsag:All>

    <wsag:ExactlyOne>

        <wsag:GuaranteeTerm wsag:Name="GT8"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSDoctor">
                <SLATest:Method>
                    <NameMethod>receiveAlarm</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>deployedOn</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>MobileNode</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
```

```
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>le</SLATest:operator>
                    <SLATest:constant>6</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>2</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT9"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSDoctor">
                <SLATest:Method>
                    <NameMethod>receiveAlarm</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>deployedOn</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>WiredServer</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>le</SLATest:operator>
                    <SLATest:constant>2</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>2</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

    </wsag:ExactlyOne>
```

```
    <wsag:ExactlyOne>

        <wsag:GuaranteeTerm wsag:Name="GT10"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSSupervisor">
                <SLATest:Method>
                    <NameMethod>receiveAlarm</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>deployedOn</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>MobileNode</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>le</SLATest:operator>
                    <SLATest:constant>20</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>1.5</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT11"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSDoctor">
                <SLATest:Method>
                    <NameMethod>receiveAlarm</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>deployedOn</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>WiredServer</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>le</SLATest:operator>
                    <SLATest:constant>15</SLATest:constant>
```

```
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>1.5</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

    </wsag:ExactlyOne>

    <wsag:All>

        <wsag:GuaranteeTerm wsag:Name="GT12"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSMedicalDevice">
                <SLATest:Method>
                    <NameMethod>getMedicalDevices</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>le</SLATest:operator>
                    <SLATest:constant>2</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>1</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT13"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSMedicalDevice">
                <SLATest:Method>
                    <NameMethod>getMeasure</NameMethod>
                </SLATest:Method>
```

```
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>idMedicalDevice</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>device_1</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>le</SLATest:operator>
                    <SLATest:constant>3</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
                    <wsag:ValueExpression>
                        <SLATest:constant>0.2</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BusinessValueList>
        </wsag:GuaranteeTerm>

        <wsag:GuaranteeTerm wsag:Name="GT14"
wsag:Obligated="ServiceProvider">

            <wsag:ServiceScope wsag:ServiceName="WSMedicalDevice">
                <SLATest:Method>
                    <NameMethod>getMeasure</NameMethod>
                </SLATest:Method>
            </wsag:ServiceScope>

            <wsag:QualifyingCondition>
                <SLATest:variable>idMedicalDevice</SLATest:variable>
                <SLATest:operator>eq</SLATest:operator>
                <SLATest:constant>device_2</SLATest:constant>
            </wsag:QualifyingCondition>

            <wsag:ServiceLevelObjective>
                <wsag:CustomServiceLevel>
                    <SLATest:variable>responseTime</SLATest:variable>
                    <SLATest:operator>le</SLATest:operator>
                    <SLATest:constant>10</SLATest:constant>
                    <SLATest:unit>seconds</SLATest:unit>
                </wsag:CustomServiceLevel>
            </wsag:ServiceLevelObjective>

            <wsag:BusinessValueList>
                <wsag:Penalty>
                    <wsag:AssessmentInterval>
                        <wsag:Count> 1 </wsag:Count>
                    </wsag:AssessmentInterval>
```

```
                    <wsag:ValueExpression>
                        <SLATest:constant>0.1</SLATest:constant>
                        <SLATest:unit>dollars</SLATest:unit>
                    </wsag:ValueExpression>
                </wsag:Penalty>
            </wsag:BbusinessValueList>
        </wsag:GuaranteeTerm>

    </wsag:All>

    </wsag:All>

  </wsag:Terms>

</wsag:AgreementOffer>
```

Acronyms

BVL	Business Value List (refers to an element of a guarantee term in WSAG)
CL	Class (refers to an element of the Classification Tree Method)
CPM	Category Partition Method
CTM	Classification Tree Method
CTR	Combined Test Requirement
GT	Guarantee Term (refers to an element of a WSAG)
JCR	Journal Citation Reports
MCDC	Modified Condition / Decision Coverage
NDT	Navigational Development Techniques methodology
PhD	Doctor of Philosophy
PTR	Primitive Test Requirement
QC	Qualifying Condition (refers to an element of a guarantee term in WSAG)
QoS	Quality of Service
SBA	Service-Based Application
SDT	Service Description Term (refers to an element of a guarantee term in WSAG)
SLA	Service Level Agreement
SLACDC	SLA Condition / Decision Coverage
SLACT	SLA Combinatorial Testing
SLATF	SLA Testing Framework
SLO	Service Level Objective (refers to an element of a guarantee term in WSAG)
SOA	Service Oriented Architecture
SUT	Software Under Test
TC	Test Case
WSAG	WS-Agreement standard language

Bibliography

[1] Amazon EC2 SLA. http://aws.amazon.com/ec2-sla/

[2] D. Ameller, and X. Franch. Service level agreement monitor (SALMon). In *Seventh International Conference on Composition-Based Software Systems (ICCBSS)*, pp. 224-227, 2008.

[3] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). 2011. http://ogf.org/documents/GFD.192.pdf

[4] AT&T services SLA. http://www.att.com/gen/general?pid=6622

[5] M. Autili, P.D. Benedetto, and P. Inverardi. Context-aware adaptive services: The plastic approach. In *12th International Conference on Fundamental Approaches to Software Engineering, FASE 2009*, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Lecture Notes in Computer Science, vol. 5503. Springer, pages 124–139.

[6] L. Baresi, N. Georgantas, K. Hamann, V. Issarny, W. Lamersdorf, A. Metzger, and B. Pernici. Emerging research themes, Services-Oriented Systems. In *2012 Annual SRII Global Conference (SRII)*, pages 333-342, 24-27 July 2012.

[7] B. Beizer. Software testing techniques (2nd ed.) Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[8] N.D. Belnap. A useful four-valued logic. In J.M. Dunn, G. Epstein (eds.), Modern Uses of Multiple-Valued Logic, Dordrecht: Reidel, pages 8-37, 1977.

[9] A. Bertolino and E. Marchetti. A Brief Essay on Software Testing, Chapter of Software Engineering. Volume 1: Development process, Third Edition, IEEE Computer Society/Wiley Interscience, 2005, pp 393-411.

[10] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini. Model-based generation of testbeds for web services. In *Proc of Testcom/FATES*, Lecture Notes In Computer Science, vol. 5047. Springer-Verlag, Berlin, Heidelberg, 2008, pages 266-282.

[11] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. Automatic test data generation for XML schema-based partition testing. In *Proc. of the Second International Workshop on Automation of Software Test*, May 2007, International Conference on Software Engineering, IEEE Computer Society, Washington, DC, page 4.

[12] A. Bertolino, and A. Polini. SOA Test Governance: enabling service integration testing across organization and technology borders. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW '09)*, pages 277-286, 1-4 April 2009.

[13] A. Bertolino, G. De Angelis, A. Di Marco, P. Inverardi, A. Sabetta, and M. Tivoli. A framework for analyzing and testing the performance of software services. In *Proceedings of the 3rd ISoLA*. CCIS, vol. 17, Springer, Heidelberg, 2008.

[14] A. Bertolino. Software testing research: Achievement, challenges and dreams. In *FOSE'07: Future of Software Engineering*, pages 85-103, 2007.

[15] A. Bertolino, G. de Angelis, and A. Polini. A QoS test-bed generator for web services. In *ICWE: Proceedings of International Conference on Web Engineering*, pages 16-20, 2007.

[16] BOE. (2011, 01/11/2012). Real Decreto 99/2011, January 28. Available. http://www.boe.es/buscar/doc.php?id=BOE-A-2011-2541

[17] M. Bozkurt, M. Harman, Y. Hassoun. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability (STVR)* 23 (4), 261-313, June 2013.

[18] K. Bratanis, D. Dranidis, and A. J. H. Simons. SLAs for cross-layer adaptation and monitoring of service-based applications: a case study. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications (QASBA)*, 2011.

[19] A. Bucchiarone, H. Melgratti, and F. Severoni. Testing service composition. In *Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07)*, 2007.

[20] D. Budgen, M. Turner, P. Brereton, B.A. Kitchenham, Using mapping studies in software engineering, in: Proceedings of PPIG, Lancaster University, 2008, pp. 195–204.

[21] G. Canfora and M. Di Penta. Testing and self-checking. In *Proc. of International Workshop on Web Services – Modeling and Testing (WS-MATE)*, pages 3-12, Palermo, Italy, June 2006.

[22] G. Canfora and M. Di Penta. Service-oriented architectures testing: a survey. In *Software Engineering: International Summer Schools, ISSSE 2006-2008*, Salerno, Italy, Lecture Notes In Computer Science, vol. 5413, 2009, Springer-Verlag, Berlin, pages 78-105.

[23] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Professional* 8 (2), 9–17, 2006.

[24] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. QoS-Aware replanning of composite web services. In *Proc. of the International Conference on Web Services (ICWS 2005)*, Orlando, FL, USA, July 2005.

[25] R. Casado. Testing advanced transactions in Service-based Software Systems. PhD Dissertation. University of Oviedo (Spain), 2013.

[26] V. Casola, A. Mazzeo, N. Mazzocca, and M. Rak. A SLA evaluation methodology in Service Oriented Architectures. In *Quality of Protection* (pp. 119-130), 2006, Springer US.

[27] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, vol 9 (5), pages 193-229, 1994.

[28] J.J. Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, U.S. Department of Transportation, Federal Aviation Administration, April 2001.

[29] E.F. Codd. The Relational Model for Database Management - Version 2. Addison-Wesley, Reading, MA, (1990).

[30] M.B. Cohen, M.B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA)*, ACM, pages 129-139, New York, NY, USA, 2007.

[31] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour. Establishing and monitoring SLAs in complex service based systems. In *Proc. IEEE International Conference on Web Services (ICWS)*, 2009, Los Angeles, CA.

[32] L. Copeland. A practitioner's guide to software test design. Artech House, Inc., Norwood, MA, USA, 2003.

[33] J. Czerwonka. Pairwise testing in real world. Pacific Northwest Software Quality Conference, pages 419–430, October 2006.

[34] C. De la Riva, J. Garcia-Fanjul, and J. Tuya. A partition-based approach for XPath testing. In *Proc. of the International Conference on Software Engineering Advances (ICSEA 06)*, Oct-Nov. 2006, IEEE Computer Society, Washington, DC, 17.

[35] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15, 313–34, 2008.

[36] E. Di Nitto, M. Di Penta, A. Gambi, G. Ripa, and M.L. Villani. Negotiation of Service Level Agreements: An architecture and a search-based approach. In *5th*

International Conference on Service-Oriented Computing - ICSOC 2007, pages 295–306, Vienna, Austria, September 17-20, 2007.

[37] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno. Search-based testing of service level agreements. In *Proc. of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO 07)*, London, July 2007, ACM, New York, pages 1090-1097.

[38] A. Dupuy and N. Leveson. An Empirical Evaluation of the MCDC Coverage Criterion on the HETE 2 Satellite Software. In *Proc. 19th Digital Avionics System Conference (DASC)*, 2000.

[39] A.T. Endo, A. da Simao, S. Souza, and P. Souza. Web services composition testing: a strategy based on structural testing of parallel programs. In *Testing: Academic and Industrial Conference - Practice And Research Techniques (TAIC PART)*, 2008.

[40] A.T. Endo and A.S. Simao. A systematic review on formal testing approaches for web services. In *4th Brazilian Workshop on Systematic and Automated Software Testing (SAST 2010)* - in conjunction with the 22nd IFIP International Conference on Testing Software and Systems (ICTSS'10), 2010.

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

[42] M.J. Escalona and G. Aragón. NDT a model-driven approach for web requirements. *IEEE Transactions on Software Engineering*, vol. 34 (3), pp 377-390, 2008.

[43] K. Fakhfakh, T. Chaari, S. Tazi, K. Drira, and M. Jmaiel, M. A comprehensive ontology-based approach for SLA obligations monitoring. In the *Second International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP'08)*, pp. 217-222, 2008

[44] L. Frantzen, M. N. Huerta, Z. G. Kiss, T. Wallet. On-the-fly model-based testing of web services with Jambition. In *Int. Workshop on Web Services and Formal Methods (WS-FM 2008)*, ser. LNCS, no. 5387. Springer, 2009, pages 143-157.

[45] J. García-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 (TAIC PART 2006)*, pp. 127-130, 2006.

[46] J. García-Fanjul, J. Tuya, and C. De La Riva. Generating test cases specifications for BPEL compositions of web services using SPIN. In *International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, pp. 83-94, 2006

[47] J. García-Fanjul, M. Palacios, J. Tuya, and C. de la Riva. Pruebas de composiciones de servicios web. *Novática Journal*, number 200, pages 61-64, July-August 2009.

[48] J. García-Fanjul, M. Palacios, J. Tuya, and C. de la Riva. Methods for testing web service compositions. *The European Journal for the Informatics Professional*, vol. 10(5), pages 62-66, 2009.

[49] G. Gessert. Four valued logic for relational database systems. Sigmod Rec. 19 (1), (1990) 29- 35.

[50] N. Goel, N.V.N. Kumar, and R.K. Shyamasundar. SLA Monitor: a system for dynamic monitoring of adaptive web services. In *Proc. 9th IEEE European Conference on Web Services (ECOWS)*, pages 109-116, 2011.

[51] Google Apps SLA. http://www.google.com/apps/intl/en/terms/sla.html

[52] M. Grindal, J. Offut, and S.F. Andler. Combination testing strategies – a survey. *Software Testing, Verification and Reliability*, Volume 15, Issue 3, 167–199, September 2005.

[53] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, vol. 3 (2), 63–82, June 1993.

[54] W.C. Hetzel and B. Hetzel. The complete guide to software testing. John Wiley & Sons, Inc. 1991.

[55] J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In *Proc. of the 1st European Conference on Towards A Service-Based Internet*, Spain, Dec. 2008, Lecture Notes In Computer Science, vol. 5377, Springer-Verlag, pages 122-133.

[56] Z. Hong and Z. Yufeng. Collaborative testing of web services. *IEEE Transactions on Services Computing*, vol. 5, pp. 116-130, 2012.

[57] HP Cloud SLA: https://www.hpcloud.com/SLA

[58] IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology, http://standards.ieee.org/findstds/standard/610.12-1990.html.

[59] ISO/IEC 24765, Software and Systems Engineering Vocabulary, 2006.

[60] ISO/IEC/IEEE 29119 - Software and Systems Engineering - Software Testing Standard.

[61] ISTQB – International Software Testing Qualifying Boards. http://www.istqb.org

[62] D. Ivanovic, M. Carro, and M. Hermenegildo. Constraint-based runtime prediction of SLA violations in service orchestrations. In *Proc. International Conference on Service Oriented Computing (ICSOC)*, pages 62-76, Paphos, Cyprus, 2011.

[63] J.A. Jones and M.J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, vol 29 (3), 195-209, 2003.

[64] J. Kapoor and J.P. Bowen. A formal analysis of MCDC and RCDC test criteria. *Software Testing, Verification and Reliability*, vol 15 (1), 21-40, 2005.

[65] J. Kapoor and J.P.Bowen. Experimental evaluation of the tolerance for control-flow test criteria. *Software Testing, Verification and Reliability*, 14 (3), 167-187, 2004.

[66] A. Keller and H. Ludwig. The WSLA Framework: specifying and monitoring of service level agreements for web services. IBM research report RC22456, 2002.

[67] B.A. Kitchenham. Procedures for performing systematic reviews. Keele University Technical Report TR/SE-0401 and NICTA Technical Report 0400011T.1, 2004

[68] B.A. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report, EBSE-2007-001, UK, July 2007.

[69] B.A. Kitchenham, D. Budgen, P. Brereton, The value of mapping studies – a participant-observer case study, in: EASE'10: Proceedings of Evaluation and Assessment in Software Engineering, 2010

[70] C. Kotsokalis, R. Yahyapour, and M.A. Rojas Gonzalez. Modeling service level agreements with binary decision diagrams. In *Proc. International Conference on Service-Oriented Computing (ICSOC)*, pages 190-204, 2009.

[71] D.D. Lamanna, J. Skene, and W. Emmerich. A language for defining service level agreements. In *9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, San Juan, Puerto Rico, 2003.

[72] P. Leitner, S. Dustdar, B. Wetzstein, and F. Leymann. Cost-based prevention of violations of service level agreements in composed services using self-adaptation. *Workshop on European Software Services and Systems Research-Results and Challenges (S-Cube)*, 2012, pp. 34-35.

[73] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. Monitoring, prediction and prevention of SLA violations in composite services. In *Proc. IEEE International Conference on Web Services (ICWS) Industry and Applications Track*, 2010, pages 369-376.

[74] 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 IEEE International Symposium on Service-Oriented System Engineering*, pp. 135-141, 2008

[75] D. Lorenzoli and G. Spanoudakis. EVEREST+: Runtime SLA violations prediction. In *5th Middleware for Service-oriented Computing Workshop*, in conjunction with the 11th ACM/IFIP/USENIX International Middleware Conference, 2010.

[76] D. Lorenzoli and G. Spanoudakis. Runtime prediction of software service availability. In *Int. Conference on Software Engineering Research and Practice (SERP'11)*, July 18-21, 2011, USA

[77] K. Mahbub and G. Spanoudakis. Monitoring WS-Agreements: an event calculus based approach. Test and Analysis of Service Oriented Systems, Springer Verlang, 2007, pp. 265-306.

[78] J.D. McCaffrey. An empirical study of pairwise test set generation using a genetic algorithm. In *7th International Conference on Information Technology: New Generations (ITNG)*, 2010, 12-14 April 2010, pages 992-997.

[79] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, Volume 37 Issue 4, 316-344, December 2005.

[80] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol*, 14 (1), 1–41, 2005.

[81] Microsoft. (2012). Microsoft Academic Research.

[82] Microsoft PICT (Pairwise Independent Combinatorial Testing). http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi

[83] Microsoft Windows Azure SLA. http://www.windowsazure.com/en-us/support/legal/sla

[84] A. Mosallanejad, R. Atan. HA-SLA: a hierarchical autonomic SLA model for SLA monitoring in cloud computing. *Journal of Software Engineering and Applications*, vol. 6, 114-117, 2013.

[85] C. Muller, M. Oriol, M. Rodriguez, X. Franch, J. Marco, M. Resinas, and A. Ruiz-Cortes. SALMonADA: A platform for monitoring and explaining violations of WS-agreement-compliant documents. In *Workshop on Principles of Engineering Service Oriented Systems (PESOS)*, 2012 ICSE, pages 43-49, 4 June 2012.

[86] C. Muller, M. Resinas, and A. Ruiz-Cortes. Automated analysis of conflicts in WS-Agreement. *IEEE Transactions on Services Computing*. 2013. IEEE computer Society Digital Library. IEEE Computer Society, http://dx.doi.org/10.1109/TSC.2013.9.

[87] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Comprehensive QoS monitoring of web services and event-based sla violation detection. In *Proceedings of the 4th international workshop on middleware for service oriented computing* (pp. 1-6). ACM, 2009

[88] G.J. Myers. The Art of Software Testing. John Wiley & Sons, 2004.

[89] T. Nanba, T. Tsuchiya, and T. Kikuno. Constructing test sets for pairwise testing: A SAT-based approach. In *2nd International Conference on Networking and Computing (ICNC)*, 2011, Nov. 30 2011-Dec. 2 2011, pp.271-274.

[90] NDTQ-Frawework. www.iwt2.org/iwt2/ndt-qframework.php

[91] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, Volume 43 Issue 2, January 2011.

[92] J. Offut, L. Nan, P. Ammann, and X. Wuzhi. Using abstraction and Web applications to teach criteria-based test design. In *24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, pages 227-236, 2011.

[93] M. Oriol, J. Marco, X. Franch, and D. Ameller. Monitoring adaptable SOA system using SALMon. In *Workshop of Service Monitoring, Adaptation and Beyond (MONA+)*, ServiceWave Conf., 2008.

[94] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications ACM* 31 (6), 676-686, Jun. 1988.

[95] M. Palacios, J. García-Fanjul, J. Tuya, and C. de la Riva. Estado del arte en la investigación de métodos y herramientas de pruebas para procesos de negocio BPEL. In *Actas de las IV Jornadas Científico-Técnicas en Servicios Web y SOA (JSWEB-08)*, pages 132–137, Sevilla, Spain, October 2008.

[96] M. Palacios, J. García-Fanjul, J. Tuya, and C. de la Riva. A proactive approach to test service level agreements. In *5th International Conference on Software Engineering Advances (ICSEA 2010)*, pages 453-458, Nice, France, 2010.

[97] M. Palacios, J. García-Fanjul, and J. Tuya. Protocolo para la revisión sistemática de estudios sobre pruebas en SOAs con enlace dinámico. In *V Taller sobre Pruebas en Ingeniería del Software (PRIS 2010)*, Valencia, Spain, September 2010. Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos, Vol. 4, No. 5, pages 17-24, 2010.

[98] M. Palacios, J. García-Fanjul, and J. Tuya. Testing in service oriented architectures with dynamic binding: a mapping study. *Information and Software Technology*, 53 (3), 171-189, March 2011.

[99] M. Palacios. Defining an SLA-aware method to test service oriented systems. In *9th International Conference on Service-Oriented Computing (ICSOC)*, PhD Symposium, Paphos, Cyprus, December 2011. In G. Pallis et al. (Eds.): ICSOC 2011, LNCS 7221, Springer 2012, pages 164–170, 2012.

[100] M. Palacios, J. García-Fanjul, and J. Tuya. Definición de una estrategia de pruebas basada en acuerdos de nivel de servicio. In *XVI Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2011)*, pages 519-524, La Coruña, Spain, September 2011.

[101] M. Palacios, J. García-Fanjul, J. Tuya, and G. Spanoudakis. Identifying test requirements by analyzing SLA guarantee terms. In *19th International Conference on Web Services (ICWS)*, pages 351-358, Honolulu, Hawaii, USA, June 2012.

[102] M. Palacios, L. Moreno, M. J. Escalona, and M. Ruiz. Evaluating the service level agreements of NDT under WS-Agreement. An empirical analysis. In *8th International Conference on Web Information Systems and Technologies (WEBIST 2012)*, pages 246-250, Oporto, Portugal, 2012.

[103] M. Palacios, J. García-Fanjul, and J. Tuya. Testing in service oriented architectures with dynamic binding: a mapping study. In *XVII Jornadas en Ingeniería del Software (JISBD 2012)* (Already published articles), pages 383-384, Almería, Spain, September 2012.

[104] M. Palacios, P. Robles, J. García-Fanjul, J. Tuya. SLACT: a test case generation tool for service level agreements. In *XVIII Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2013)*. Madrid, 2013

[105] M. Palacios, J. García-Fanjul, J. Tuya, and G. Spanoudakis. Coverage-based testing for service level agreements. *IEEE Transactions on Services Computing* 2014. DOI 10.1109/TSC.2014.2300486

[106] M. Palacios, J. García-Fanjul, J. Tuya, and G. Spanoudakis. Automatic test case generation for WS-Agreements using combinatorial testing. *Computer Standards & Interfaces* (Submitted in 2013).

[107] M. Palacios, J. García-Fanjul, and J. Tuya. Design and implementation of a tool to test service level agreements. *IEEE Latin America Transactions*. Vol. 12, Issue 2, March 2014, pp. 256-261.

[108] L. Paliulionienè. On description of contracts and agreements in the context of SOA. *Computational Science and Techniques*, Vol. 1 Number 2, 183-195, 2013.

[109] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: state of the art and research challenges. *IEEE Computer* 40 (11), 38-45, November 2007.

[110] M.P. Papazoglou and W.J.A.M. van den Heuvel, Service oriented architectures: approaches, technologies and research issues, *Very Large Database Journal* 16 (3), 389–415, 2007.

[111] PLASTIC European Project homepage. http://www.ist-plastic.org/

[112] A. Pichot, O. Waldrich, W. Ziegler, and P. Wieder. Towards dynamic service level agreement negotiation: an approach based on WS-Agreement. In *Proceeding of the Fourth International Conference on Web Information Systems and Technologies*, 2008.

[113] D.M. Quan and L.T. Yang. Parallel mapping with time optimization for SLA-aware compositional services in the business grid. *IEEE Transactions on Services Computing*, vol. 4, no. 3, 196-206, July-Sept. 2011.

[114] F. Raimondi, J. Skene, and W. Emmerich. Efficient online monitoring of web-service SLAs. In *Proceedings of the 16th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16)*, 2008.

[115] O. Rana and W. Ziegler. Research challenges in managing and using service level agreements. In *Grids, P2P and Services Computing*, New York, NY: Springer, 2010, pages 187-200.

[116] RCTA Inc. DO-178-B: Software considerations in airborne systems and equipment certification. Radio Technical Commission for Aeronautics (RTCA). 1992.

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

[118] D. Sabbah. Bringing grid & web service together. In Opening Keynote Globus World 2004, Vice President of Strategy and Technology, IBM Software Group.

[119] A. Sahai, V. Machiraju, M. Sayal, A. Van Moorsel, and F. Casati. Automated SLA monitoring for web services. In *Management Technologies for E-Commerce and E-Business Applications* (pp. 28-41). Springer Berlin Heidelberg, 2002.

[120] E. Schmieders, A. Micsik, M. Oriol, K. Mahbub, and R. Kazhamiakin. Combining SLA prediction and cross layer adaptation for preventing SLA violations. In *Proc. 2nd Workshop on Software Services: Cloud Computing and Applications based on Software Services*, 2011, Timisoara, Romania.

[121] J. Skene, F. Raimondi, and W. Emmerich. Service-level agreements for electronic services. *IEEE Transactions on Software Engineering*, 36 (2), 288-304, March-April 2010.

[122] SLA@SOI European Project. http://sla-at-soi.eu

[123] SLACT (SLA Combinatorial Testing). http://in2test.lsi.uniovi.es/tools/slact/

[124] SOAP - Simple Object Access Protocol. http://www.w3.org/TR/2007/REC-soap12-part0-20070427

[125] Software Engineering Research Group (GIIS) sample SLAs. http://giis.uniovi.es/testing/downloads/sla/?lang=en

[126] G. Spanoudakis and K. Mahbub. Non-intrusive monitoring of service-based systems. *International Journal of Cooperative Information Systems*, Vol. 15 (03), 325-358, 2006.

[127] C.A. Sun, Y. Shang, Y. Zhao, and T.Y. Chen. Scenario-oriented testing for web service compositions using BPEL. In *12th International Conference on Quality Software (QSIC)*, pp. 171- 174, 2012.

[128] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller. A concept for QoS integration in web services. In *1st Web Services Quality Workshop (WQW 2003)*, in conjunction with IEEE Computer Society 4th International Conference on Web Information Systems Engineering (WISE 2003), Rome, Italy, December 2003.

[129] V. Tosic, B. Pagurek, K. Patel, B. Esfandiari, and W. Ma. Management applications of the Web Service Offerings Language (WSOL). In *15th International Conference on Advanced Information Systems Engineering (CAiSE'03)*, Velden, Austria, June 2003.

[130] J. Trienekens, J. Bouman, and M. VanDerZwan. Specification of service level agreements: problems, principles and practices. *Software Quality Journal*, 12, 43– 57, 2004.

[131] J. Tuya, M.J. Suárez-Cabal, and C. de la Riva. Full predicate coverage for testing SQL database queries. *Software Testing, Verification and Reliability*, 20 (3), 237-288, September 2010.

[132] UDDI - Universal Description, Discovery and Integration. http://www.uddi.org/pubs/uddi_v3.htm

[133] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices* 35, 6, 26–36, June 2000.

[134] A.S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and U. Yalçinalp. Web services policy 1.5-framework. W3C Recommendation, 4, 1-41, 2007.

[135] H. Wada, J. Suzuki, Y. Yamano, and K. Oba. E3: a multiobjective optimization framework for SLA-aware service composition. *IEEE Transactions on Services Computing*, vol. 5, no. 3, 358-372, Third Quarter 2012.

[136] Q. Wang; J. Shao; F. Deng; Y. Liu; M. Li, J. Han, and M. Hong. An online monitoring approach for web service requirements. *IEEE Transactions on Services Computing*, vol. 2, no. 4, 338-351, Oct.-Dec. 2009.

[137] M.R. Woodward and M.A. Hennell. On the relationship between two control-flow coverage criteria: all JJpaths and MCDC. *Information and Software Technology*, vol 48 (7), 433-440, 2006.

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

[139] WSDL - Web Service Description Language. http://www.w3.org/TR/wsdl20

[140] T.K. Yu and M.F. Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software*, vol. 79 (5), 577-590, 2005.

[141] Z. Zakaria, R.Atan, A.A.A. Ghani, N.F.M. Sani. Unit testing approaches for BPEL: a systematic review. In *APSEC: Proceedings of the Asia-Pacific Software Engineering Conference*, pages 316-322, 2009.

[142] F.H. Zulkernine and P. Martin. An adaptive and intelligent SLA negotiation system for web services. *IEEE Transactions on Services Computing*, vol. 4, no. 1, 31-43, Jan.-March 2011.