City Research Online

City St George's, University of London

Citation: Palacios, M., García-Fanjul, J., Tuya, J. & Spanoudakis, G. (2015). Automatic test case generation for WS-Agreements using combinatorial testing. Computer Standards and Interfaces, 38, pp. 84-100. doi: 10.1016/j.csi.2014.10.003

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/5147/

Link to published version: https://doi.org/10.1016/j.csi.2014.10.003

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

Automatic test case generation for WS-Agreements using combinatorial testing

Marcos Palacios[a *], José García-Fanjul[a], Javier Tuya[a], George Spanoudakis[b]

[a] Department of Computer Science, University of Oviedo

{palaciosmarcos, jgfanjul, tuya}@ uniovi.es

[b] School of Informatics, City University London

G.E.Spanoudakis@ city.ac.uk

* Corresponding Author (Marcos Palacios)

Email address: palaciosmarcos@uniovi.es

Postal address: Department of Computer Science (1.S.25). Campus Universitario de Gijón. 33204. Gijón. Asturias. Spain.

Tel: +34 985 182 153. Fax: +34 985 181 986

Abstract

In the scope of the applications developed under the service-based paradigm, Service Level Agreements (SLAs) are a standard mechanism used to flexibly specify the Quality of Service (QoS) that must be delivered. These agreements contain the conditions negotiated between the service provider and consumers as well as the potential penalties derived from the violation of such conditions. In this context, it is important to assure that the service based application (SBA) behaves as expected in order to avoid potential consequences like penalties or dissatisfaction between the stakeholders that have negotiated and signed the SLA. In this article we address the testing of SLAs specified using the WS-Agreement standard by means of applying testing techniques such as the Classification Tree Method and Combinatorial Testing to generate test cases. From the content of the individual terms of the SLA, we identify situations that need to be tested. We also obtain a set of constraints based on the SLA specification and the behaviour of the SBA in order to guarantee the testability of the test cases. Furthermore, we define three different coverage strategies with the aim at grading the intensity of the tests. Finally, we have developed a tool named SLACT (SLA Combinatorial Testing) in order to automate the process and we have applied the whole approach to an eHealth case study.

Keywords: Software Testing, Service Based Applications, Service Level Agreements, WS-Agreement, Classification Tree Method, Combinatorial Testing.

1. Introduction

Service Oriented Architecture (SOA) has become a solid paradigm to develop interoperable, flexible and highly dynamic service based applications (SBAs) by means of integrating available services over the web. The main features of these services include loose coupling between them, coarse grained service interfaces, dynamic service discovery and binding, self-containment of services, service interoperability and protocol independence [1]. SBAs are implemented using different Internet-based standards [2] such as the Simple Object Access Protocol (SOAP) [3] for transmitting data, the Web Service Description Language (WSDL) [4] for defining services or the Business Process Executable Language for Web Services (BPEL4WS) [5] for orchestrating services.

In SBAs, it is necessary for the stakeholders involved to specify the conditions related to the provision and consumption of the services. These conditions are usually specified

in a contract or technical document, called Service Level Agreement (SLA), which is a standard mechanism that allows determining and regulating the trading between the service providers and the consumers. In the scope of SBAs, the WS-Agreement standard [6] is a language used to specify the conditions negotiated and agreed by these stakeholders. WS-Agreement supports the representation of information regarding the functional features of services, non-functional requirements related to the Quality of Service (QoS) level that should be achieved by the service provision, penalties derived from the violation of the terms and any other relevant information related to the agreement. It is therefore of utmost importance for both providers and consumers to develop suitable actions that allow avoiding or minimizing the consequences derived from SLA violations.

The management of SLAs [7] is an integral part of the applications developed under the rules of a standard SOA Governance framework [8] and has received considerable attention both in industry and academia (see, for example, the SLA@SOI FP7 European Project [9]). Many large companies, including Amazon, Microsoft, Google, AT&T and HP, 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 [10][11][12][13][14].

The management of SLAs involves different tasks including SLA negotiation [15], re-negotiation [16][17], specification [18][19], evaluation [20], testing [21], and monitoring [22]. Among these tasks, the testing of the SLAs has been identified as a challenge [23][24][25]. The testing of the SLAs involves designing and executing tests in the SBA by means of considering the specification of the SLA as the test basis. This requires that the specification of the SLAs needs to be somehow formalized in order to automate as much as possible the process of obtaining the tests.

Currently, there are different approaches that aim at detecting, forecasting or preventing SLA violations using testing techniques. Most of these works are reactive and use monitoring to observe the behaviour of the SBA at runtime in order to detect potential deviations from the guaranteed conditions specified in the SLA [22][40]. These approaches are useful in detecting problems in SBAs but they present an important drawback: problems are detected after their occurrence and, therefore, consequences derived from such problems cannot be avoided. On the other hand, different proactive approaches have been proposed to predict or anticipate the detection of problems associated to the violation of the SLA [21][36].

Considering the characteristics of both approaches, in a previous work we presented a conceptual method to test SLA-aware service based applications. In that work, our aim was to combine the advantages of both proactive and reactive approaches [26]. Later, we addressed the identification of situations to be tested from the information contained in the individual SLA terms [27]. These situations may be used to guide the design of an appropriate test suite that exercises such situations and also to derive a monitoring plan allowing checking the compliance of the SLA at runtime.

In this article, and as a further step, we define a method to generate test cases from an SLA specified in WS-Agreement standard language by integrating testing techniques that have been used broadly in the industry and standardized in ISO/IEC/IEEE 29119 [28]. We also provide a tool that automates the whole process. The main contributions of this article are the following:

1) We define how standard techniques for testing, namely the *Classification Tree Method* and *Combinatorial Testing* can be applied in the context of SLAs in order to obtain a set of test cases that are suitable for testing an SBA in which the conditions that need to be satisfied are specified in a single SLA.
2) We define how to automatically obtain specific constraints from the specification of the SLA and the behaviour of the SBA in order to avoid the generation of non-feasible test cases.

3) We define three different coverage strategies with the aim at grading the thoroughness of the resulting test suite.
4) We implement a tool called SLACT that automatically generates the test cases, making use of an existing testing tool [29].
5) We apply the approach to an eHealth scenario that was proposed by the EU F7 project PLASTIC [30] and has been previously used by other authors to test SLAs [31][32][33][34].

These contributions aim at taking into account some particular challenges that arise from the testing of service-based systems [23]. In our case, we have to deal with the controllability of the services and the infrastructure so, to mitigate this limitation, the services will be under our control and, consequently, the set of generated test cases will be executed in a controlled environment.

The rest of this article is structured as follows. Section 2 outlines related work. Section 3 provides a background about the two cornerstones of this research: Software Testing standards and WS-Agreement. It also summarizes the four-valued logic used to evaluate SLA Guarantee Terms, which was previously developed by the authors [27] and is again used in this article. Section 4 discusses the generation of test cases and the level of automation provided by SLACT. Section 5 presents the results derived from the application of our approach to the eHealth case study. Section 6 highlights the main limitations of this approach. Finally, Section 7 provides some concluding remarks and outlines plans for future work.

2. Related Work

In the scope of service-based applications, considerable effort has been spent in detecting SLA violations using different testing approaches. Typically, related strands of work may be categorized in two main groups: (i) the set of works which are aimed at anticipating problems and/or prevent them before such problems lead to undesired consequences for the stakeholders who have signed the agreement; and (ii) the set of works that are aimed at detecting SLA violations at runtime when the Software Under Test (SUT) is already deployed in the operational environment.

Few approaches have focused on the identification of tests from the specification of the SLAs in order to anticipate problems. Di Penta et al. [21] 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, as we do in our contribution, and bindings of the constituent services that may cause violations of the SLA. Palacios et al. [35] identify test requirements from the conditions included in an SLA specified in WS-Agreement using a well known testing technique, called *Category Partition Method*. Once such test requirements have been identified, they are combined in order to derive the test cases. Such combinations of the identified test requirements are not addressed in that work whereas in our contribution we apply combinatorial testing techniques in order to derive the test cases from the test requirements. Palacios et at. [36] also provides a coverage based criterion in order to test SLA-aware service-based application. In that work they focus on the logical relationships between the guarantee terms of a SLA specified in WS-Agreement whereas in this article we are focusing on the content of the individual guarantee terms. Furthermore, Bertolino et al. [31] have proposed the PUPPET framework, which allows generating stubs from the WS-Agreement, WSDL and BPEL specification of the services to test SLA-aware service compositions. This work is related to our work. However, instead of specifying the tests for the SBA as we do, they provide the necessary infrastructure to deploy and execute such tests. Thus, both works may be mutually complemented. Kotsokalis et al. [37] 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 as we do although they attempt to obtain the diagrams from this language (WS-Agreement). In their approach, they use two different values to evaluate the terms of the SLA. In our work, we show that four different values are necessary to consider all the potential situations derived from the evaluation of SLA Guarantee Terms. Finally, Muller et al. [38][39] propose to detect and explain conflicts within the specification of WS-Agreements by means of applying techniques based on Constraints Satisfaction Problems. This work focuses on checking whether the specification of the SLA guarantee terms is consistent so the detected problems are related to the SLA and not the services. In our work we focus on the detection of problems in the SUT by means of taking the specification of the SLA as the test basis.

Regarding the second group, several works have addressed the testing of SLAs using monitoring approaches in order to detect SLA violations. Mahbub and Spanoudakis [22] 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. [40] 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. [41] tackles the testing of SLA-aware SBA by monitoring the conditions specified in the SLA. This work was developed in the scope of the SLA@SOI European Project [9]. Beyond these works, there are other systems that have been developed to monitor whether service based applications violate SLAs including, for example, SALMon [42][43], SLAMonitor [44] HA-SLA [45] and CLAM [46].

Between these two groups, there is a set of works that make use of information gathered from monitoring techniques in order to prevent SLA violations. For example, Leitner et al. [47] 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 [48]. Ivanovic et al. [49] 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. Schmieders et al. [50] combined monitoring and prediction techniques in order to prevent SLA violations. Finally, Lorenzoli and Spanoudakis [51] presented a framework (EVEREST+) which supports the monitoring and prediction of potential violations of the QoS metrics specified in an SLA.

3. Background

In this section we present the basic concepts about the two cornerstones of our research. On the one hand, we introduce some important standard definitions which are broadly used in the field of software testing. On the other hand, we focus on the structure of WS-Agreement standard language and how it is been used within the provisioning of applications developed under the paradigm of service oriented architectures. We also briefly present our previous research which is extended and improved in this article.

3.1 Software testing and standards

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”, according to the ISO/IEC 24765 (Software and Systems Engineering Vocabulary) [52].

The generation of test cases allows designing the conditions under which the SUT will be executed. This is important for the success of the tests as a good test design will allow detecting a higher number of faults. 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” [53]. 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 approaches that use the generation and execution of test cases are proactive in the sense that they are able to anticipate the detection of faults before the problems occur in a production environment. The generation of test cases tries to maximize the trade-off among different business criteria such as cost, benefit or risks. In some cases, it may be possible to design an in-depth and exhaustive test suite even if it involves a high cost in terms of money or effort. In other cases, however, there might be constraints hindering the definition and execution of exhaustive tests. When this happens, the tester is forced to select a less exhaustive testing technique. The generation of test cases is often a very tedious task so it is desirable to automate it as much of it as it is possible.

Currently, the fragmentation of the different standards is a common problem in this field. To fill this gap, the ISO/IEC/IEEE 29119 Systems and Software Engineering - Software Testing standard [28] is being developed with the aim of providing one definitive reference for software testing that defines vocabulary, processes, documentation and techniques. This standard comprises four parts: Definitions and Vocabulary (part 1), Test Process (part 2), Test Documentation (part 3) and Test Techniques (part 4).

In this article, we describe how two techniques which are used broadly in industry and academia [54] and are described in Part 4 of ISO/IEC/IEEE 29119, namely the Classification Tree Method and Combinatorial Testing, can be applied in testing SLAs.

The Classification Tree Method [55] 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 [56] 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 *All combinations*, *Pair-wise* or *Each choice* that will be later used in this article.

3.2 WS-Agreement Standard

Over the last decade, different languages have been proposed with the aim to support and standardize the specification of SLAs (e.g., WSLA [57], WS-Agreement [6], WSLO [58], SLANG [59][18], WS-Policy [60], the SLA Model [61] and WS-QoS [62]). In our work, we focus on WS-Agreement because it is a well-accepted standard in the SOA protocol stack for the management of the SLAs (Figure 1) and has been used in different approaches regarding the testing of SBAs.

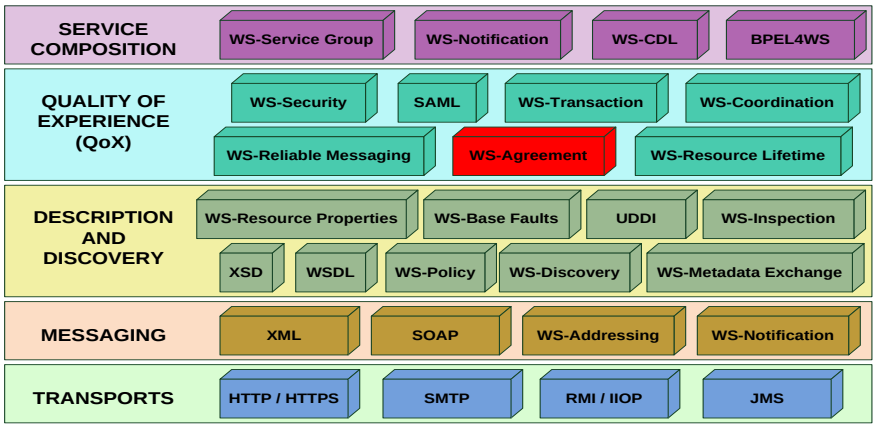

Figure 1: Web Service Protocol Stack (adapted from IBM Software Group [63])

WS-Agreement [6] was proposed by the Open Grid Forum working group (OGF). WS-Agreement is a standard that specifies a protocol for establishing agreements between two parties and a schema for the definition of SLAs. The specification of domain specific languages or extensions to express the conditions of the Guarantee Terms is out of the scope of WS-Agreement. The specification of an SLA in WS-Agreement is composed of three main parts (Figure 2):

- **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

The Guarantee Terms are the most important section of an SLA. A Guarantee Term contains: an internal element, called *Scope*, that specifies the list of services and, an optional *substructure* of the service that the terms applies to (for example, a particular method or endpoint); a *Qualifying Condition (QC)* which is an assertion or precondition determining whether the term is valid or not; and a *Service Level Objective (SLO)* that is the guarantee that must be met. 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.

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.

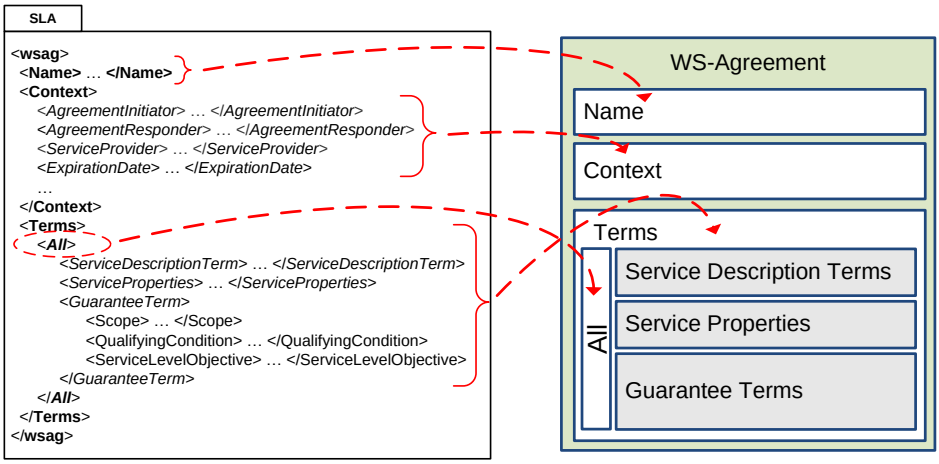

Figure 2: WS-Agreement structure

3.3 Evaluation of SLA Guarantee Terms

In previous research [27], we have developed a four-valued logic and defined a method to identify a set of test values from the information contained in the SLA Guarantee Terms.

The evaluation is one of the most important tasks within the management of SLAs in the context of SBAs. It requires analyzing the information gathered from the monitors, checking the specification of the guarantee terms and their internal elements and, finally, making a decision about the fulfilment of the conditions contained in such terms. Typically, the evaluation of an SLA may be depicted with a two-way traffic light indicator (green / red), which represents whether the agreement has been fulfilled or violated respectively.

Focusing on WS-Agreement standard language, a Guarantee Term is specified by means of the internal elements *Scope*, *Qualifying Condition (QC)* and *Service Level Objective (SLO)*. Considering the syntax of a Guarantee Term and the possible forms of analysis of the collected information from the service executions at runtime, there are four (as opposed to two) possible evaluation values for a Guarantee term, notably:

- *Fulfilled* – This value can be used 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* – This value can be used 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.
- *Not Determined* – This value can be used if and only if the methods of the services specified in the Scope have not been executed and the Qualifying Condition is met.
- *Inapplicable* – This value can be used if and only if the Qualifying Condition has not been satisfied.

The first three evaluation values are explicitly identified in the WS-Agreement standard as the three potential states in which the SLA can be so, from a testing point of view, we have added a four value (Inapplicable) in order to represent specific situations that are also interesting to be tested. In this case and in addition to the typical two evaluation values (i.e., Fulfilled / Violated), the utilization of *Not Determined* and *Inapplicable* leads to a four-valued logic where they are two similar interpretations of the treatment of the null value in the three-valued logic, broadly studied in the context of

Data Base Management Systems (DBMS) and applied in the scope of software testing [64][65][66][67].

Hence, a Guarantee Term denoted by *t* can be evaluated using a function *ev*, which can provide four different values as output:

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

At first glance and based on these four evaluation values, we could think that it is necessary to identify four different situations with the aim of achieving full coverage while evaluating the Guarantee Term. However, the internal syntactic structure and the semantics of a guarantee term specified in WS-Agreement standard language require a more exhaustive test suite to represent the whole set of situations that are interesting to observe or exercise from a testing point of view.

At this point, it is important to distinguish between the concepts of *evaluation values* and *test values*. An *evaluation value* is the outcome of the process of making a decision about the fulfilment of a Guarantee Term. If the behaviour of this mechanism is grounded in the proposed logic, there will be four possible evaluation values (i.e., Fulfilled, Violated, Not Determined and Inapplicable). On the other hand, a *test value* (also known as *test requirement* in our previous work) represents a situation of the SBA that must be covered (and satisfied) during testing [68]. A *test value* includes a set of conditions and steps that need to be checked through the execution of the SUT. And during this check, useful information gathered from monitors can be used by the evaluation mechanism in order to make a decision and provide the final *evaluation value* for the *test value*. In Figure 3, we show an example where a test value is identified from the content of a Guarantee Term specified in WS-Agreement. This test value exercises the situation where the method (service) specified in the Scope is executed, the Qualifying Condition is met and the objective is not satisfied.

Thus, according to the above logic the guarantee term is evaluated as Violated.

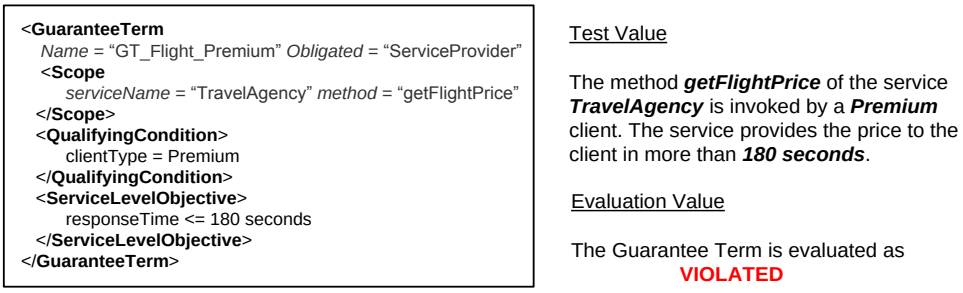

Figure 3: Relation Test Value – Evaluation Value

Keeping this in mind, a Guarantee Term may be evaluated with four different values but it is possible to identify, for each term, six test values, as it can be seen in Figure 4. More specifically, as shown in the figure, the internal elements of a Guarantee Term include Scope, Qualifying Condition and Service Level Objective. At the top of the figure we check whether the methods of the services specified in the Scope element have been invoked or not at the time of evaluating the SLA (the verification of this condition is performed using satisfied/unsatisfied as outputs). Furthermore, the content of the Qualifying Condition and the Service Level Objective represent conditions that may also be evaluated as satisfied/unsatisfied. Thus, we apply the multiple combinations of these three internal elements of a Guarantee Term. As there are three conditions with two truth-values for each condition, we would obtain eight different situations to test regarding the content of the Guarantee Term. However, due to the semantic meaning of

the internal elements of the term, there is a pair of combinations that do not make sense. These relate to cases where the methods of the services specified in the Scope have not been executed so it is useless to check whether the Service Level Objective have been satisfied or not (right branch of the Figure 4). Hence, we obtain six test values that are interesting to test from the specification of a Guarantee Term (identified by TV1-TV6).

At the bottom of the figure, we align the relation between the test values and the value provided by the evaluation mechanism when exercising such test value.

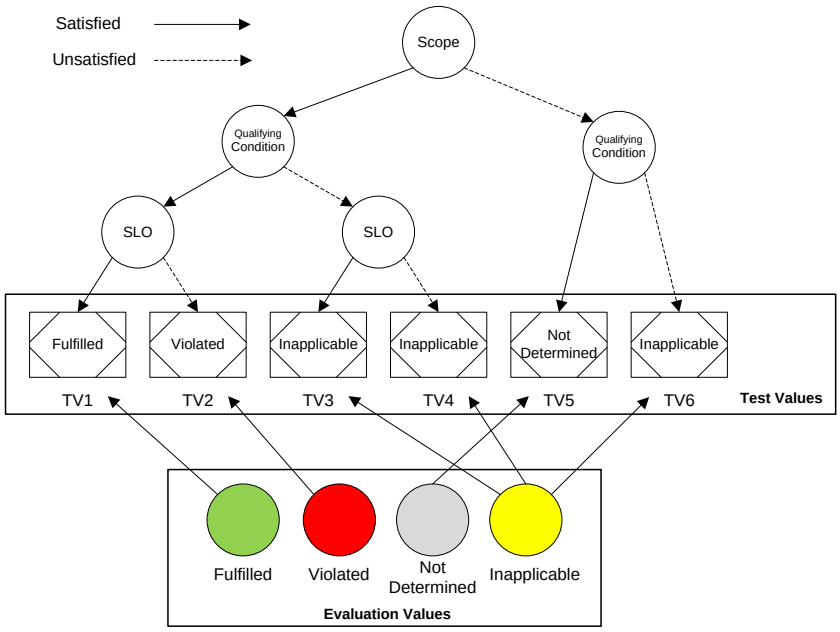

Figure 4: Identification of test values from individual Guarantee Terms

4. Test Method

An SLA specified in WS-Agreement has a hierarchical structure that logically combines the Guarantee Terms through the use of the specific elements named compositors (*All*, *OneOrMore* and *ExactlyOne*). Then, it is necessary to define how these terms are analyzed in order to identify the situations that need to be tested. In this section, we describe the method that allows deriving the test cases by means of elaborating the classification tree, specifying the constraints that guide the derivation of the test coverage items avoiding the generation of non feasible test cases and, finally, the tool that automates the whole process.

4.1 Construction of the Classification Tree

The first step of the method involves the construction of a suitable model to hierarchically represent the relevant information of the SLA specification in WS-Agreement, using the Classification Tree Method (CTM). This tree will later be used to derive the test coverage items and generate the test cases. To do so, we have to identify the classifications and classes that will formulate the tree.

The simplest approach could be to parse the structure of the SLA and construct a classification for each of the elements of the SLA. In this case the Service Level Objectives of the Guarantee Terms would represent the classifications at the lowest level whereas the evaluation of such SLOs would represent the classes of the tree. However, we need to ensure that the resulting tree represents all the test situations that may arise during the evaluation of the Guarantee Terms. Hence, we use the compositors of WS-Agreement to construct the first levels of the hierarchy in the tree and we raise the level

of abstraction of the most internal elements of the SLA. In particular, we take the structure of each Guarantee Term as a whole (including its Scope, Qualifying Condition and Service Level Objective) and we construct a classification for each of the SLA Guarantee Terms. Each of these classifications (Guarantee Terms) has six potential test values as we described in Section 3.3. Hence the leaves of the tree that represent the classes of each classification are constructed by representing the six test values for each Guarantee Term. With this approach, both the classifications and the classes fulfil the restriction of being disjoint partitions with respect to the SLA. Note that both classifications and classes represent different levels of detail of the situations to be tested. The lowest level (class) represents each situation that arises from each of the test values that have to be covered by the test cases.

It is worth mention that in order to be consistent with the notation of the testing techniques described in the ISO/IEC/IEEE 29119, in the rest of the article we will use the concept of *class* (CL) when we refer to the different situations that arise from the evaluation of a Guarantee Term.

At this stage, we have to deal with an important issue regarding the construction of the classes of the tree. 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 classes are identified. These two cases are described next.

Case 1 (C1): Guarantee Terms without Qualifying Condition

The first particular case (C1) arises when the Guarantee Term has no Qualifying Condition associated. The Qualifying Condition determines whether a term is valid 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 valid so only three classes (CL1, CL2 and CL5) are identified. Thus in the classification that represents such Guarantee Terms, only three classes are constructed in the tree (see Figure 5). The specification of the classes CL1 and CL2 in this tree represent that the methods of the services are invoked and the Service Level Objective is satisfied or unsatisfied, respectively. Class CL5 represents that the methods of the services have not been executed.

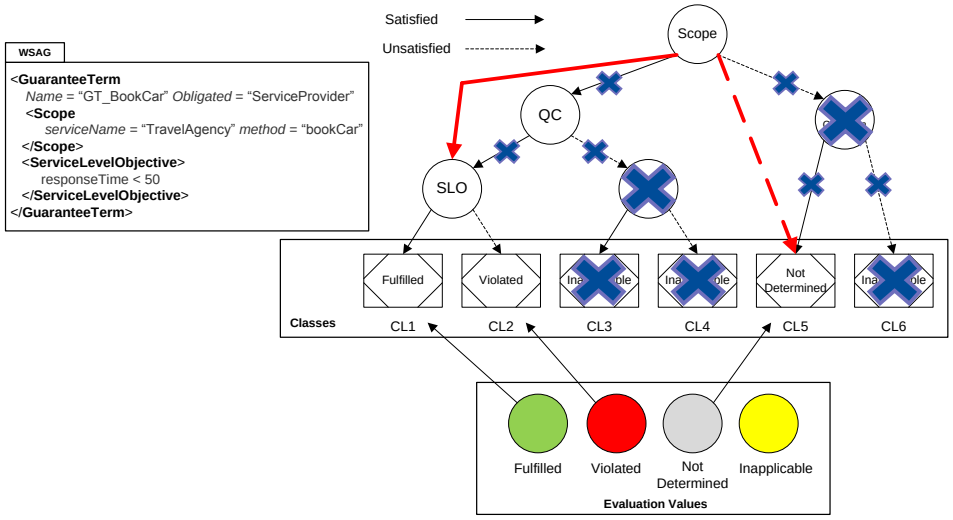

Figure 5: Application of Case 1 (C1)

Case 2 (C2): 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 (C2) 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 classes. In this case, it is impossible to check the fulfilment of the QC if the methods of the services have not been executed so the combinations performed in classes CL5 and CL6 do not make sense. In such case, classes CL5 and CL6 are joined in only one class representing that the methods of the services have not been executed (Figure 6). This means that we would construct five classes for the classification that represents such Guarantee Term instead of the six that are constructed in the general case.

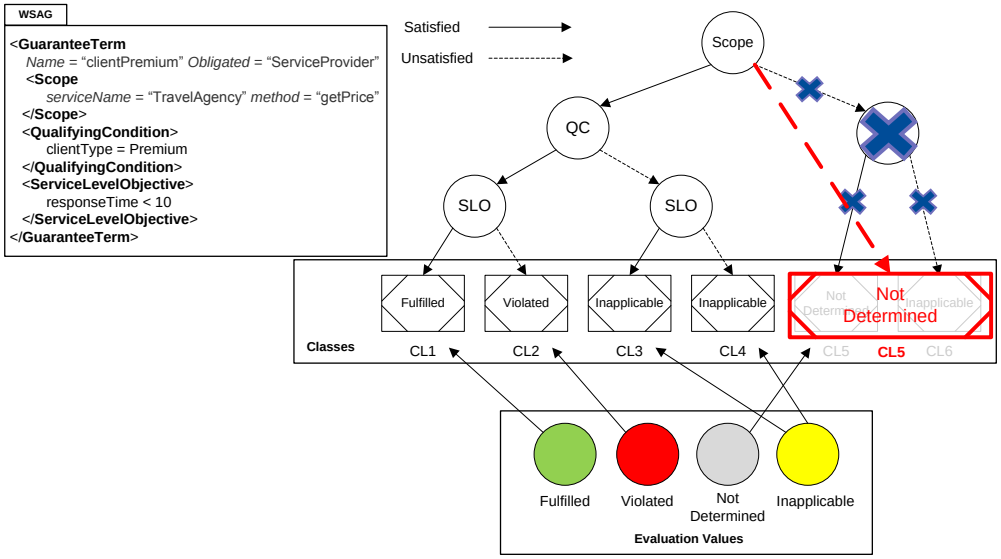

Figure 6: Application of Case 2 (C2)

After considering the application of both cases when constructing the classification tree, 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 depending on the particular cases applied. In Figure 7 we show an example of a tree constructed from the analysis of a WSAG-compliant agreement with three Guarantee Terms where no particular cases are applied to the first one, the particular case C1 is applied to the Guarantee Term GT2 and the particular case C2 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 when such class is exercised (green for Fulfilled, red for Violated, yellow for Inapplicable and grey for Not Determined).

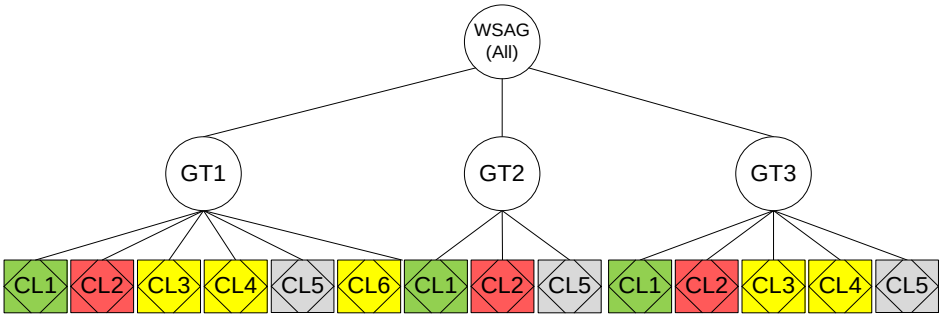

Figure 7: Example of a Classification Tree from an SLA

4.2 Generation of test cases with combinatorial testing techniques

Once we have constructed the classification tree, we can make a decision about the parts of the tree that represent the most critical situations and need to be covered with a higher intensity. To do this, we apply standard combinatorial testing techniques in order to derive the test coverage items and generate the test cases.

When deriving the test coverage items, not all the combinations of classes will be used because we have to deal with two potential problems:

- The first one is related to the number of derived test coverage items, which can be unmanageable if the SLA is complex.
- The second problem affects the testability of specific test coverage items 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 test cases. To address the second problem, we define specific constraints that the test suite has to satisfy to avoid generating non feasible test cases.

4.2.1 Combinatorial strategy

Once we have defined the classification tree and in order to derive the test coverage items, we use different combinatorial testing techniques. These techniques are defined in terms of parameters and values. When we use the constructed tree to test the SLA, the parameters are the classifications that represent the Guarantee Terms and the values are the classes that represent the test values.

After the identification of the parameters and their corresponding values, we derive the test coverage items by means of applying any of the testing techniques standardized in the ISO/IEC/IEEE 29119, which allow grading the intensity of the tests. These techniques are usually 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) represents a test coverage item and it must be exercised in at least one test case in the test suite. The most exhaustive coverage criterion is provided by *All Combinations testing*, which requires that every possible combination of classes 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 test coverage items and they must be included in at least one test case.

In addition to existing testing techniques, it is necessary to 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.

As a result of this process, we obtain the test coverage items that lead to the generation of test cases. Each test case contains a set of test coverage items where each classification is included in the test case just once (so each Guarantee Term will be evaluated once in each test case). The content of the classes combined in the test coverage items will determine the inputs of the test case. 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 classes. For example, different sources of information can be used such as UML State Transition Diagrams or Sequence Diagrams.

4.2.2 Definition of testability constraints

The derivation of the test coverage items 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 unfeasible combinations of test coverage items.

We distinguish between two types of constraints: implicit and explicit. The implicit constraints are based on the information that is represented in the terms of the SLA. The explicit constraints are identified through the analysis of the SUT. Both sets of constraints are always related to combinations of classes that represent feasible or non-feasible situations to be exercised, disregarding the characteristics of the SUT.

4.2.2.1 Implicit Constraints

Based on the syntax and semantics structure of WS-Agreement, we can identify a set of implicit constraints that can help avoiding non feasible combinations of classes used to derive the test cases. 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 4.1 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 $ev(GT_x) = CLy$, where GT_x is the classification that represents such Guarantee Term and CLy is the corresponding class.

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 test coverage item (the method/service specified in the Scope of GT1 is not executed), then one of the classes CL5-CL6 of the classification that represents GT2 must also be exercised. This constraint can be formally expressed as:

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

I2: Guarantee Terms that have the same Qualifying Conditions

If some Guarantee Terms share the same Qualifying Condition and this is not met, then all the classifications that represent these Guarantee Terms must take the values of the classes CL3, CL4 or CL6.

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

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

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.

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

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

4.2.2.2 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 specification of these constraints is manually done by the tester by using IF-THEN statements in which specific combinations of the GT evaluation values are allowed or forbidden.

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:

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

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.

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

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

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

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.

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

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.

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

E6: Additional constraints

Depending on the content of QCs or SLOs, the use of a specific test value of GT (GT_i) may require also the use of a specific test value 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 8) and a subset of the identified classes (right part of Figure 8). 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).

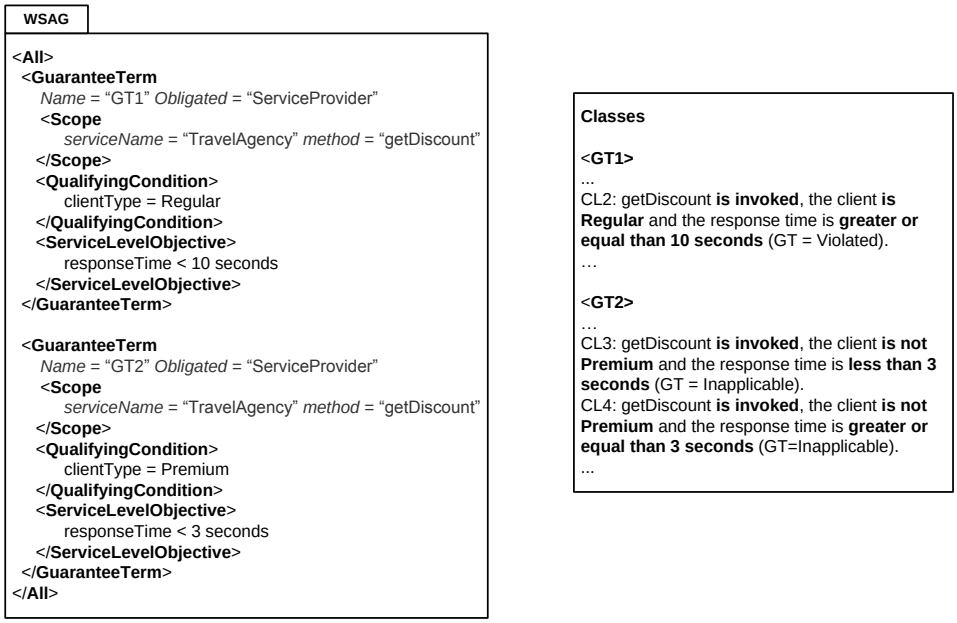

Figure 8: Excerpt of SLA Guarantee Terms and identified classes

4.3 Tool support: SLACT (SLA Combinatorial Testing)

To address the generation of test cases, we have developed a tool that is able to automatically generate test cases by means of deriving test coverage items through the combination of the classes identified from the SLA Guarantee Terms, called SLACT (SLA Combinatorial Testing). This tool builds upon an existing combinatorial testing tool [29].

SLACT has been implemented to automate: (1) the identification of classes and the definition of the implicit constraints, both processes from the specification of the WS-Agreement, (2) the application of the combinatorial testing technique according to the coverage strategy selected by the tester, and (3) the generation of a test suite that satisfies the expected coverage with the least number of test cases and the analysis of the coverage of the classes of each Guarantee Term.

SLACT has five components, as shown in Figure 9, namely the SLA Parser, Constraints Generator, Executor, Analyzer and, finally, the User Interface. The roles of these components are discussed below.

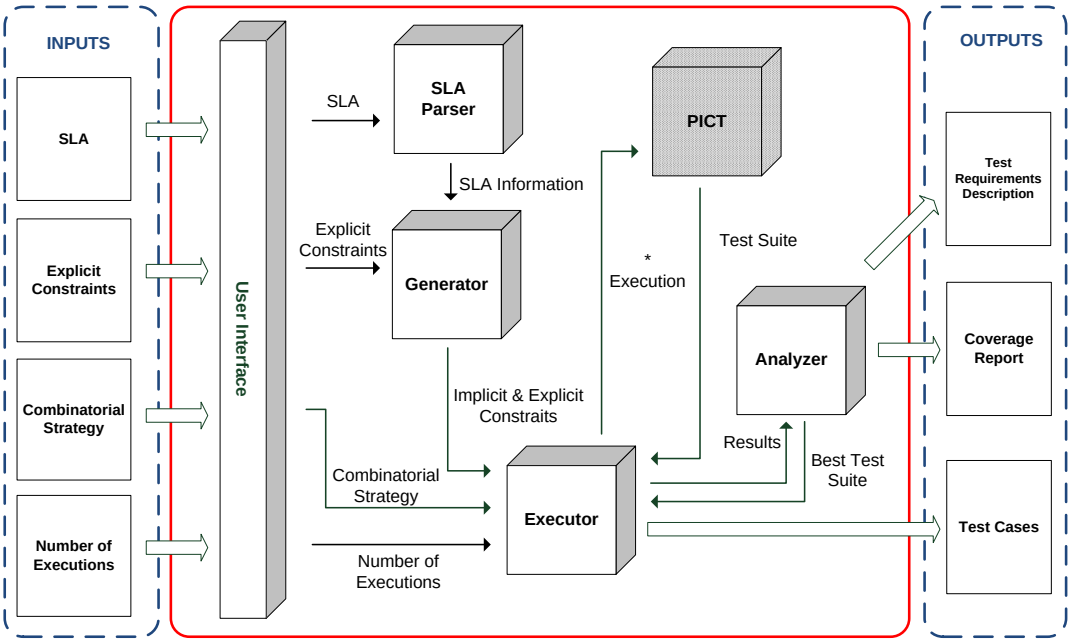

Figure 9: SLACT Architecture

SLA Parser

The first of the SLACT components is called *SLA Parser*. This component is in charge of parsing the XML document that contains the SLA specified in WS-Agreement language and extracting the relevant information of the individual Guarantee Terms.

Constraints Generator

The second component of SLACT is called *Constraints Generator*. This component provides the following functions:

- Analyzing the information extracted from the parser to automatically obtain the implicit constraints.
- Allowing the definition of the explicit constraints.
- Allowing the selection of the combinatorial strategy.

The *Constraints Generator* is in charge of analyzing the information received from the SLA Parser in order to construct the classification tree by means of identifying the classifications and classes as it is described in Section 4.1. Also from the specification of the SLA, it automatically obtains the set of implicit constraints (taking the general case and the particular cases into account). Furthermore, through its User Interface (UI), SLACT allows the definition of the set of explicit constraints and the selection of the strategy that will guide the combinations to derive the test coverage items. Regarding this strategy, the tool allows the application of 1-wise, 2-wise or N-wise to all the classifications as well as the definition of a hybrid strategy that partially applies different combinatorial techniques to specific sets of classifications.

Executor

The third component of SLACT is called *Executor*. This component is in charge of executing the combinations of test cases with the appropriate parameters and values, considering the constraints provided by the Generator and the coverage strategy selected by the tester. Executor may be run in two different ways:

(i) It may perform multiple executions with the aim at obtaining different test suites (the number of executions can also be specified by the tester)

(ii) It may perform a single execution with a specific and previously identified input in order to obtain the test suite that contains the least number of test cases for the selected strategy.

This component makes use of the Pairwise Independent Combinatorial Tool (PICT) [29], which is a free tool developed by Microsoft that has been previously used in other testing approaches [69][70][71]. The core generation algorithm of PICT is based on a greedy heuristic optimized for speed. The output of Executor is the specification of the test suite that satisfies the selected coverage strategy with the least number of test cases.

Analyzer

The forth component of SLACT is called *Analyzer*. This component is in charge of two main tasks:

(i) It receives the results of the multiple executions and obtains the test suite that satisfies the expected coverage with the least number of test cases.

(ii) It also receives the specification of a test suite and performs an analysis of the test cases in order to assure that all of the classes are exercised and provides a report regarding the coverage of such classes.

User Interface

The last component of SLACT is the *User Interface* (Figure 10). This component allows:

- Selecting the XML document of the SLA.
- Representing the relevant information of the SLA as well as the implicit constraints automatically identified.
- Specifying the explicit constraints.
- Selecting the combinatorial testing strategy to be applied and the execution conditions.

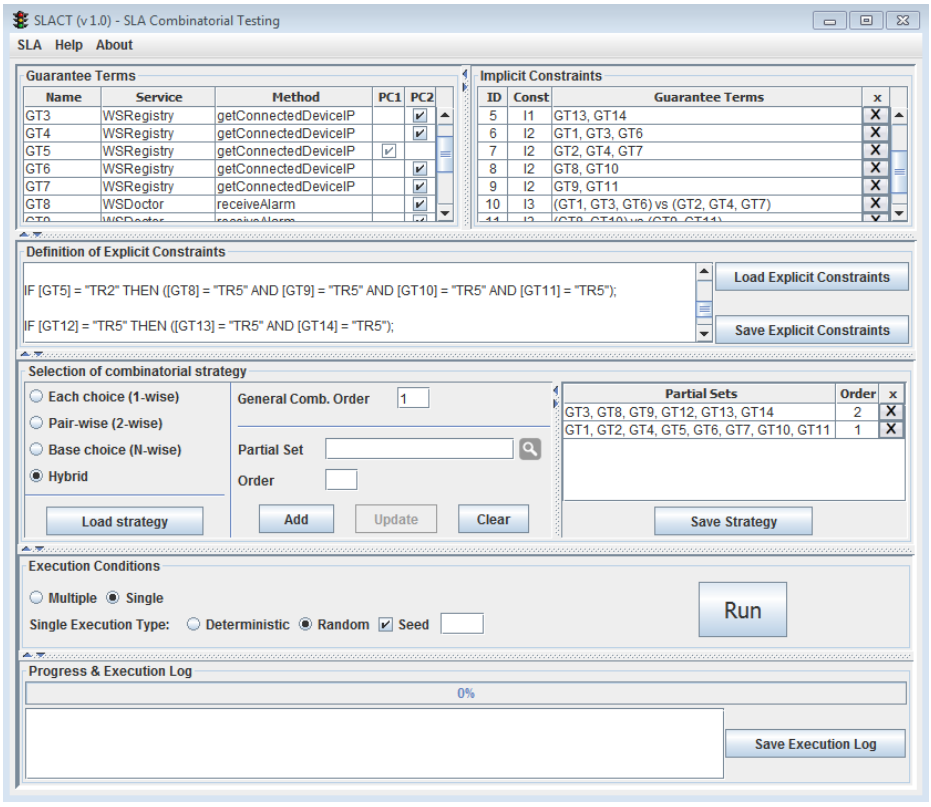

Figure 10: SLACT User Interface

5. Case Study: eHealth service-based Application

To illustrate the complete process of generating test cases for SLAs, we have carried out a case study based on an eHealth service-based application scenario that has been proposed by the European Project PLASTIC [30] and has been used in previous service-aware testing approaches [31][32][33][34]. This scenario represents the set of conditions that must be satisfied by the constituent services of an eHealth service based application as described in an SLA specified in WS-Agreement. This SLA can be downloaded from [72]. All the steps of the method described in Section 4 have been automatically performed using SLACT.

5.1 Description

The behaviour of the eHealth case study is represented in Figure 11. In summary, the application is deployed as a composite web service (WSHealth) that receives an alarm from patients and triggers appropriate actions to solve such alarms. 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. WSRegistry provides a list of IP addresses of the professionals who are available at that moment depending on the type of the alarm (*Emergency* or *Not Confirmation*). These professionals (WSDoctor or WSSupervisor) 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.

The conditions that have to be fulfilled by the constituent services of this eHealth system are specified in an SLA using WS-Agreement. The SLA contains 14 Guarantee Terms 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.

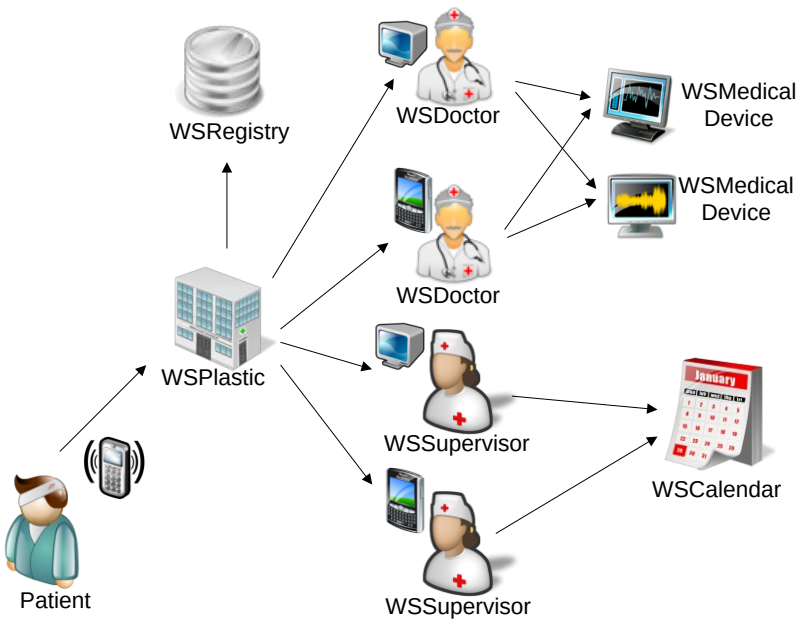

Figure 11: eHealth Scenario

5.2 Construction of the Classification Tree

We have applied the steps described in Section 4.1 in order to construct the tree by means of identifying the classifications and classes from the eHealth SLA. The results of this process are summarized in Table 1. The services and methods that constitute the case study are represented in the first column. The classifications at the lowest level of the tree are represented in the second column. The particular cases applied to identify the classes are outlined in the third 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 never 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 test values for each Guarantee Term. Hence, the number of identified classifications is 14 and the number of classes is 66. This model is the basis to generate the test cases by means of deriving the test coverage items. This task is performed by combining the classes using specific combinatorial criteria and specifying the rules that guide such combinations. In this case study, the derivation of test coverage items involves the combination of 14 classifications. Twelve of these classifications have 5 classes each. The other two classifications have 3 classes each.

Service.Method	Classification	Case/s	Classes
WSHealth.reportAlarm	GT1	C2	CL1, CL2, CL3, CL4, CL5
	GT2	C2	CL1, CL2, CL3, CL4, CL5
WS.Registry.getResidentialGateway	-	-	
WSRegistry.getConnectedDeviceIP	GT3	C2	CL1, CL2, CL3, CL4, CL5
	GT4	C2	CL1, CL2, CL3, CL4, CL5
	GT5	C1, C2	CL1, CL2, CL5
	GT6	C2	CL1, CL2, CL3, CL4, CL5
	GT7	C2	CL1, CL2, CL3, CL4, CL5
WSDoctor.receiveAlarm	GT8	C2	CL1, CL2, CL3, CL4, CL5
	GT9	C2	CL1, CL2, CL3, CL4, CL5
WSSupervisor.receiveAlarm	GT10	C2	CL1, CL2, CL3, CL4, CL5
	GT11	C2	CL1, CL2, CL3, CL4, CL5
WSMedicalDevice.getMedicalDevice	GT12	C1, C2	CL1, CL2, CL5
WSMedicalDevice.getMeasure	GT13	C2	CL1, CL2, CL3, CL4, CL5
	GT14	C2	CL1, CL2, CL3, CL4, CL5
WSCalendar.getAppointmentByMonth	-	-	-
WSCalendar.getAppointment	-	-	-
Total	14 Classifications		66 Classes

Table 1: Traceability between GTs and CLs

5.3 Generation of test cases

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 whereas the explicit constraints are specified through the User Interface of SLACT (see Figure 10). After analyzing the content of the SLA and relevant information regarding the behaviour of the SUT, 26 constraints have been identified in order to guide the generation of test cases. All these constraints are represented in Table 2. 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.

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 = Not 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 = Not 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)
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 Not 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 2: eHealth implicit and explicit constraints

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 test coverage items that will be used during testing. In general three different strategies can be applied in order to grade the level of intensity of the obtained test suites:

(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. Particularly, in the selected case study, 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 dispersed in the SLA and, thus, the classifications that represent such Guarantee Terms (GT1, GT3, GT8, GT9, GT12, GT13 and GT14 of Table 1) 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.

Bearing these considerations in mind, we have executed SLACT 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 test cases 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 test cases for the each choice, pair-wise and hybrid strategies respectively.

The results of these multiple executions are represented in Figure 12. The x-axis in the figure represents the size of the obtained test suites (the number of test cases generated in each test suite) provided by the tool. The y-axis represents the number of times each size is obtained. For example, for the *each choice* strategy, a test suite with 11 generated tests cases has been obtained more than 1200 times.

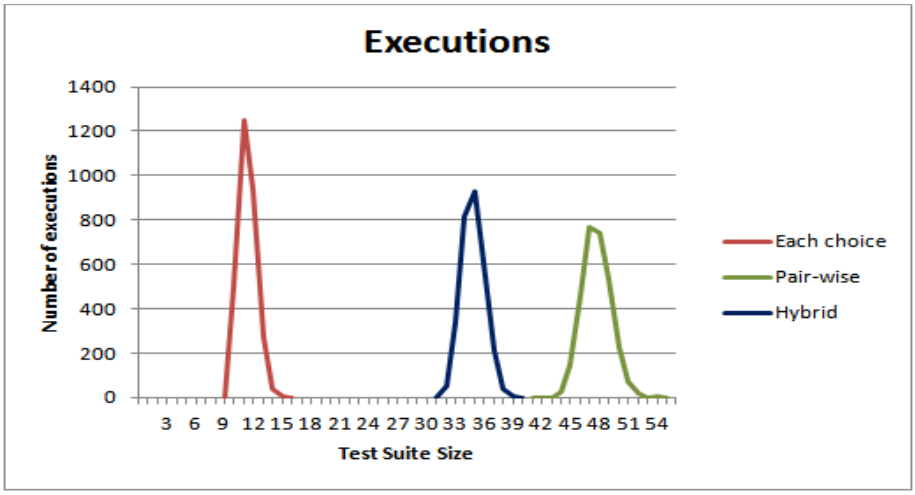

Figure 12: Multiple executions results

In the case of the *each choice* strategy, the results obtained for SLACT presents a *mean* (μ) of 11.37 (number of test cases) and a *standard deviation* (σ) of 0.91. To be more specific, 95% of the test suites contain approximately between 10 and 13 test cases. 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 test suite with a number of test cases between 45 and 50. Finally, the hybrid strategy is represented by μ = 34.84, σ = 1.22 and 95% of the test suites would contain between 33 and 37 test cases.

Analyzing the results for each applied coverage strategy and starting with the first one (*each choice*), we have obtained a test suite that contains 10 test cases (the output file is represented in Figure 13). In this file, the classifications of the tree (Guarantee Terms) are represented in columns and the test cases obtained through the combination of the classes are represented in rows. In order to describe how to derive test cases from the combinations of the classes to a test case, we consider, for example, test case number 6, outlined in the figure. The test case is generated by analyzing, according to the specification of the SLA, the meaning of each of the classes contained in such test case as well as the knowledge about the behaviour of the SUT. In this test case, we are exercising the situation when an alarm arrives to the eHealth system. The steps included in this test case and their corresponding exercised classes are represented in Table 3. To execute the test cases, we will have to sequentially exercise the steps described in such table.

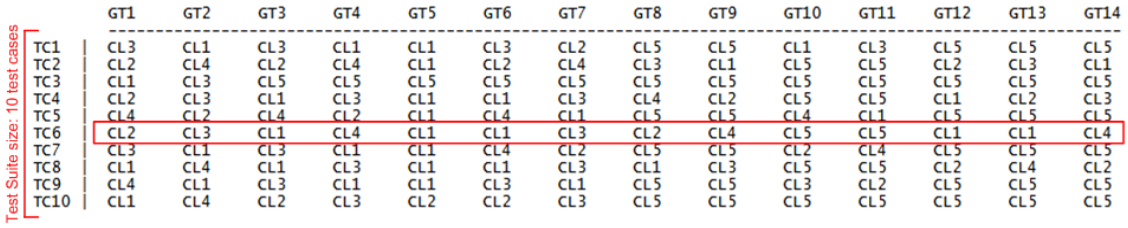

Figure 13: Content File of Each Choice Test Suite

Exercised Classes	Test Case Steps
	An alarm arrives to the eHealth system.
GT5 = CL1 GT6 = CL1 GT7 = CL3	The registry is queried and it provides a correct list of professionals.
GT3 = CL1 GT4 = CL4	The registry provides the answer in less than the specified time.
GT8 = CL2 GT9 = CL4	A doctor connected to the system through a mobile device is contacted but he does not provide an answer in the specified time.
GT12 = CL1	In spite of this fact, the doctor finds the list of devices deployed in the patient's home.
GT13 = CL1 GT14 = CL4	The doctor successfully gets the measure of device_1.
GT1 = CL2 GT2 = CL3	After having carried out all of these tasks, the eHealth system provides a response to the patient spending more time than the specified in the SLA
GT10 = CL5 GT11 = CL5	No supervisors are invoked in this test case.

Table 3: Specification of a test case

With these ten test cases, 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 2 and, thus, they are impossible to be exercised) and the coverage report provided by the Analyzer component of SLACT is represented in Table 4 (a). 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 test suite. For example, the class CL1 identified from the specification of GT5 is exercised in 8 test cases 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 4.

	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	6	16	13	0	3	1	2	26	0
GT3	3	2	3	1	1	13	14	5	6	4	11	16	1	3	1
GT4	3	1	3	2	1	6	5	14	13	4	2	2	26	1	1
GT5	8	1	-	-	1	34	4	-	-	4	30	1	-	-	1
GT6	3	2	2	2	1	14	13	5	6	4	26	1	1	3	1
GT7	2	2	4	1	1	5	6	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	6	5	7	17	5	7	6	7	7
GT10	1	1	1	1	6	3	2	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	9	-	-	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	4	5	5	23	4	4	5	6	13

Table 4: Classes coverage analysis: each choice (a), pair-wise (b), hybrid (c)

Regarding the second of the applied coverage strategies (*pair-wise*), the test suite with the least number of test cases that we obtained contained 42 test cases. The number of test cases 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 test case. The results provided by the Analyzer regarding the coverage for the classes of each Guarantee Term are also represented in Table 4 (b). As it can be seen in this table, all classes in the 2-wise strategy have been exercised more 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).

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 test suite we have obtained contains 32 test cases. The results provided by the Analyzer are represented in Table 4 (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 14. 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.

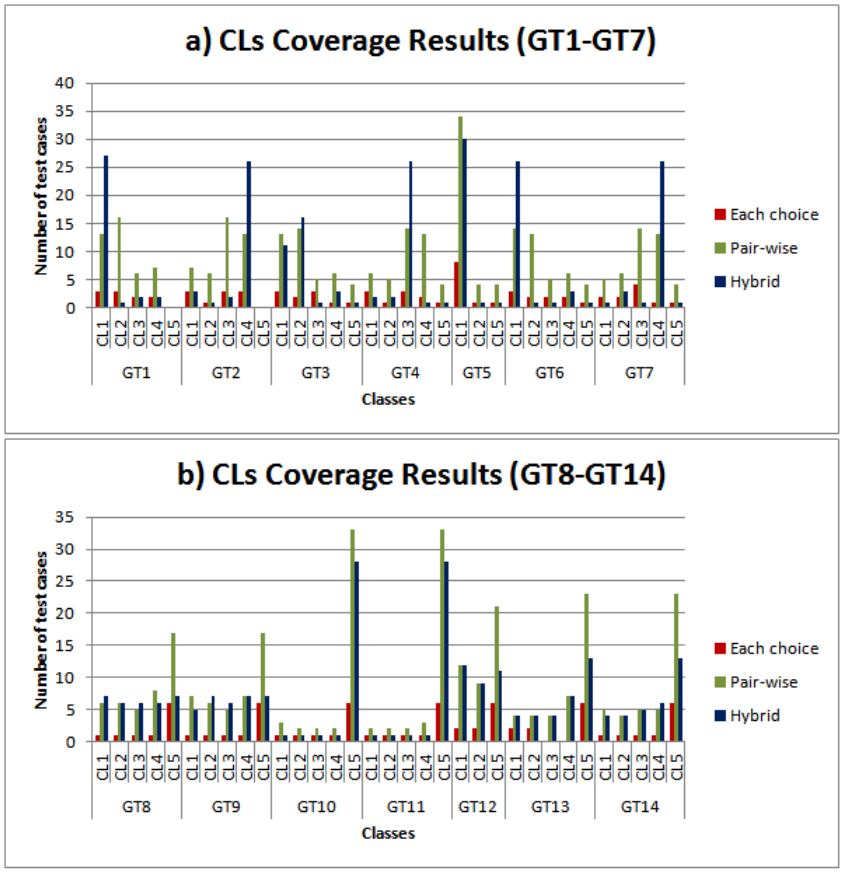

Figure 14: CLs Coverage Results

Analyzing the results obtained from the applied strategies, we hereafter highlight some considerations. First of all, it is the *each-choice* strategy the one that allows obtaining the smallest test suite whereas the *pairwise* strategy provides the largest set of test cases. Hence, the *each-choice* strategy is useful when, on the one hand, the criticity of the application is not high in the sense that an SLA violation does not lead to serious consequences for the users and, on the other hand, specific factors such as deadlines or budget hinder the design and execution of more detailed tests. Likewise, the application of *pairwise* testing is effective when the application needs to be exhaustively tested and we have fewer limitations that prevent from executing more thorough tests.

As intermediate solution we recommend to apply a hybrid testing strategy. By analyzing the application under test and identifying its more critical functionalities, we could design an appropriate strategy that allows obtaining a good balance between the number of generated tests and the intensity of such tests within the aforementioned parts of the application. In this case, we could decide to apply *pairwise* testing to the most critical functionalities whereas *each-choice* testing could be apply to the rest, as we have previously described.

Bearing these strategies in mind, the number of generated test cases depends on the strategy we choose when designing the tests. The more intensity we decide to test the SUT, the higher number of test cases will be obtained. However, even when the SUT has a complex SLA with many guarantee terms associated, the election of the each-choice testing allows obtaining a reduced number of test cases so the scalability does not represent a problem.

In addition to the choice of the testing strategy to be applied, the definition of the explicit constraints is another task that needs to be manually performed. As we have previously described, the use of the explicit constraints allows us to avoid the obtaining

of non-feasible combinations of test requirements concerning the behaviour of the SUT. If these explicit constraints are not defined and used, our approach allows identifying the test requirements and performs the combinations although the results would be less efficient. In this case, the tester would have to analyze each of the identified combinations of test requirements and, consequently, the test cases in order to detect situations that could not be exercised. Due to this, we would recommend to analyze the characteristics of the SUT and define the constraints before the obtaining of the tests because this is a task that is performed only once. After that, such constraints can be used to obtain different set of tests by applying the aforementioned testing strategies.

In our contribution, the most effort and time consuming tasks are the definition of the explicit constraints and the final derivation of the test cases from the identified combinations of test requirements because the identification of the test requirements, the obtaining of the implicit constraints and the combination of such requirements are fully automated. In this sense the scalability of the approach in terms of time-consuming is not a problem if we had to manage a SLA with a higher number of guarantee terms because the executions of SLACT are measured in the scale of a few seconds.

6. Limitations of this approach

In this section we discuss the main limitations of this approach.

First of all, we have used the WS-Agreement [6] 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, for example, WSLA [57], WSLO [58], SLANG [59][18], WS-QoS [62] or WS-Policy [60], the specification language that has received more attention regarding the testing of SLAs has been WS-Agreement, at least from the academic domain. As WS-Agreement presents a generic syntax, we envision that its derived outcomes could be extrapolated to any other existing SLA specification language.

In addition to this, in our work we are analyzing the content of the individual guarantee terms in order to generate the tests. However, an SLA may represent a hierarchy of terms that are logically combined using the specific *compositor* elements. As we state in our future work, we will improve the generation of tests by means of taking into account the logical structure of the agreement.

Likewise, in this work we are considering the content of the Qualifying Condition and the Service Level Objective elements 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 evaluated. We consider that a more detailed analysis of such elements could help to refine and improve the generated tests although the size of the test suite could grow and become unmanageable.

Finally, we have described that the SLA is our test basis and its analysis allows us to generate the test cases. From the content of the SLA, we use the developed tool, SLACT, to obtain the set of test requirements that will be later exercised through the generated test cases. However, any 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 one.

7. Conclusions and Future Work

In this article we have been concerned with the problem of testing service based applications (SBAs) regulated by Service Level Agreements (SLAs) that have been negotiated between the service provider and consumer. To address this problem, we propose a step-wise method that generates test cases from a specification based on WS-Agreement by means of defining how to apply existing testing techniques used in the

industry. We have also developed tool, called SLACT (SLA Combinatorial Testing), to automate the process.

The approach is based on identifying a set of classifications and classes from the content of the SLA in order to construct a hierarchical model using the Classification Tree Method. From this structure, we apply standard combinatorial testing techniques in order to derive the test coverage items through the combination of the classes represented in the leaves. We have applied three strategies that provide different levels of thoroughness in the resulting test suites.

The main benefit of our work is that supports the automatic generation of a set of test cases from the specification of SLAs, described in WS-Agreement. The execution of the test cases allows detecting problems in the SBA proactively, i.e., before such problems lead to undesired consequences for the stakeholders who have agreed the SLA. In addition to this, our approach can assure that the number of generated test cases will be manageable and such test cases are feasible to be executed in the SBA. Furthermore, it gives the tester the possibility to decide whether the SBA should be tested with more or less thoroughness or even determine which parts of the SBAs should be tested and with what degree of intensity.

The whole process has been automated by SLACT. SLACT receives the SLA and automatically identifies the classes and extracts the implicit constraints. The tool can be used to define explicit testing constraints and selecting a combinatorial strategy for testing. Based on these inputs, SLACT generates different test suites and provides one that, whilst satisfying the expected coverage, contains the least number of test cases.

The approach has been validated using an existing eHealth service based application where we applied the above three testing strategies obtaining three different test suites with 10, 42 and 32 test cases respectively. Thus we were able to obtain a reasonable and manageable number of tests for a critical scenario.

In future work, we will focus on improving the definition of tests using additional information contained in the SLA and taking both the logical and hierarchical structure of the agreement into account. We expect to be able to identify new cases for testing by applying existing standard coverage criteria as, for example, the Modified Condition Decision Coverage (MCDC) defined in the RTCA/DO-178B standard [73]. In relation to this, we will have to evaluate the testability of these test situations and decide which of them can be exercised through the generation of tests and which of them should be checked at runtime using monitoring techniques. Finally, we are planning to study the feasibility of improving SLACT with the aim of integrating these new test criteria.

Acknowledgment

This work 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 Test4DBS (TIN2010-20057-C03-01), project PERTEST (TIN2013-46928-C3-1-R) and FICYT (Government of the Principality of Asturias) Grant BP09-075.

References

1. E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, K. Pohl, A journey to highly dynamic, self-adaptive service-based applications, Automated Software Engineering, 15 (2008) 313–341.
2. M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-Oriented Computing: State of the Art and Research Challenges, IEEE Computer, 11, (2007).
3. Simple Object Access Protocol (SOAP). http://www.w3.org/TR/2007/REC-soap12-part0-20070427/ (Accessed August 2014).
4. Web Service Description Language (WSDL). http://www.w3.org/TR/wsdl20/ (Accessed August 2014).

5. OASIS: Web Services Business Process Execution Language (WSBPEL). http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (Accessed August 2014).
6. A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu, Web Services Agreement Specification (WS-Agreement), 2011.
7. O. Rana, W. Ziegler, Research challenges in managing and using Service Level Agreements, Grids, P2P and Services Computing, New York, NY: Springer, 2010, S. 187-200.
8. A. Bertolino, A. Polini, SOA Test Governance: enabling service integration testing across organization and technology borders, International Conference on Software Testing, Verification and Validation Workshops, 2009. ICSTW '09, 1-4 April 2009, pp.277-286.
9. SLA@SOI European Project. http://sla-at-soi.eu/ (Accessed in August 2014).
10. Amazon EC2 SLA: http://aws.amazon.com/ec2-sla/ (Accessed in August 2014)
11. Microsoft Windows Azure SLA: http://www.windowsazure.com/en-us/support/legal/sla (Accessed in August 2014)
12. Google Apps SLA: http://www.google.com/apps/intl/en/terms/sla.html (Accessed in August 2014)
13. AT&T services SLA: http://www.att.com/gen/general?pid=6622 (Accessed in August 2014)
14. HP Cloud SLA: https://www.hpcloud.com/SLA (Accessed in August 2014)
15. E. Di Nitto, M. Di Penta, A. Gambi, G. Ripa, M.L. Villani, Negotiation of Service Level Agreements: An architecture and a search-based approach, Fifth International Conference on Service-Oriented Computing - ICSOC 2007, Vienna, Austria, September 17-20, 2007, Proceedings. (2007) 295–306.
16. A. Rumpl, O. Wäldrich, W. Ziegler, Extending WS-Agreement with Multi-round Negotiation Capability, Grids and Service-Oriented Architectures for Service Level Agreements, CoreGRID series 13, Springer, 2010, 89-103.
17. S Sharaf, K Djemame, Enabling service-level agreement renegotiation through extending WS-Agreement specification, Service Oriented Computing and Applications, 2014.
18. J. Skene, F. Raimondi, W. Emmerich, Service-Level Agreements for Electronic Services, IEEE Transactions on Software Engineering, 36 (2), (March-April 2010) 288--304.
19. J. Trienekens, J. Bouman, M. VanDerZwan, Specification of service level agreements: Problems, principles and practices, Software Quality Journal, 12 (2004) 43– 57.
20. M. Palacios, L. Moreno, M.J. Escalona, M. Ruiz, Evaluating the Service Level Agreements of NDT under WS-Agreement. An empirical analysis, Proceedings of the 8th International Conference on Web Information Systems and Technologies, Porto, Portugal, 18-21 April 2012, pp. 246-250.
21. M. Di Penta, G. Canfora, G. Esposito, V. Mazza, M. Bruno, Search-based testing of service level agreements, Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO 07), London, ACM, New York, 2007, pp. 1090-1097.
22. K. Mahbub, G. Spanoudakis, Monitoring WS-Agreements: an event calculus based approach, Test and Analysis of Service Oriented Systems, Springer V., 2007, pp. 265-306.
23. G. Canfora, M. Di Penta, Testing services and service-centric systems: challenges and opportunities, IT Professional 8 (2) (2006) 9–17.
24. L. Baresi, N. Georgantas, K. Hamann, V. Issarny, W. Lamersdorf, A. Metzger, B. Pernici, Emerging Research Themes, Services-Oriented Systems, SRII Global Conference (SRII), 2012 Annual, 24-27 July 2012, pp.333-342.
25. M. Palacios, J. García-Fanjul, J. Tuya, Testing service oriented architectures with dynamic binding: a mapping study, Information and Software Technology, vol. 53 (3), (2011), 171-189.
26. M. Palacios, Defining an SLA-aware method to test service-oriented systems, Proceeding of the 9th International Conference on Service Oriented Computing (ICSOC), PhD

Symposium, G. Pallis et al. (Eds.): ICSOC 2011, LNCS 7221, pp. 164-170. Springer, Heidelberg (2012).
27. M. Palacios, J. García-Fanjul, J. Tuya, G. Spanoudakis, Identifying test requirements by analyzing SLA Guarantee Terms, Proceedings of the 19th International Conference on Web Services, Application and Experience Track, Honolulu, Hawaii, USA, 24-29 June 2012, pp- 351-358.
28. ISO/IEC/IEEE 29119 - Software and Systems Engineering - Software Testing. http://www.softwaretestingstandard.org/ (Accessed in August 2014)
29. J. Czerwonka, Pairwise testing in real world, Pacific Northwest Software Quality Conference, October 2006, pp. 419–430.
30. PLASTIC European Project. http://www.ist-plastic.org/ (Accessed in August 2014).
31. A. Bertolino, G. De Angelis, L. Frantzen, A. Polini, Model-based generation of testbeds for web services, Testing of communicating systems and formal approaches to software testing - TESTCOM/FATES. LNCS, vol. 5047, 2008, pp. 266-282. Springer.
32. L. Frantzen, M.N. Huerta, Z.G. Kiss, T. Wallet, On-The-Fly model-based testing of web services with Jambition, 5th International Workshop on Web Services and Formal Methods (WS-FM 2008), ser. LNCS, no. 5387. Springer, 2009, pp. 143-157.
33. A. Bertolino, G. De Angelis, A. Di Marco, P. Inverardi, A. Sabetta, M. Tivoli, A framework for analyzing and testing the performance of software services, Proceedings of the 3rd ISoLA. CCIS, vol. 17, Springer, Heidelberg, 2008.
34. M. Autili, P.D. Benedetto, P. Inverardi, Context-aware adaptive services: The plastic approach, 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. Proceedings. Lecture Notes in Computer Science, vol. 5503. Springer, 124–139.
35. 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.
36. M. Palacios, J. García-Fanjul, J. Tuya, G. Spanoudakis, Coverage-based Testing for Service Level Agreements, IEEE Transactions on Service Computing, February 2014 (in press).
37. C. Kotsokalis, R. Yahyapour, M.A. Rojas Gonzalez, Modeling Service Level Agreements with Binary Decision Diagrams, International Conference on Service-Oriented Computing (ICSOC), 2009. LNCS. Vol. 5900, 190-204.
38. C. Muller, M. Resinas, A. Ruiz-Cortes, Automated Analysis of Conflicts in WS-Agreement, IEEE Transactions on Services Computing, February 2013 (in press).
39. C. Muller, M. Oriol, X. Franch, J. Marco, Comprohensive Explanations of SLA Violations at Runtime, IEEE Transactions on Service Computing, vol. 7 (2) (2014), 168-183.
40. F. Raimondi, J. Skene, W. Emmerich, Efficient online monitoring of web-service SLAs, Proceedings of the 16th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16), 2008.
41. M. Comuzzi, C. Kotsokalis, G. Spanoudakis, R. Yahyapour, Establishing and Monitoring SLAs in Complex Service Based Systems, IEEE International Conference on Web Services (ICWS), 2009, Los Angeles, CA, pp. 783-790.
42. C. Muller, M. Oriol, M. Rodriguez, X. Franch, J. Marco, M. Resinas, A. Ruiz-Cortes, SALMonADA: A platform for monitoring and explaining violations of WS-agreement-compliant documents, Workshop on Principles of Engineering Service Oriented Systems (PESOS), 2012 ICSE, 4 June 2012, pp.43-49.
43. M. Oriol, J. Marco, X. Franch, D. Ameller, Monitoring adaptable SOA system using SALMon, Workshop of Service Monitoring, Adaptation and Beyond (MONA+), ServiceWave Conference, 2008.
44. N. Goel, .V.N. Kumar, R.K. Shyamasundar, SLA Monitor: A System for Dynamic Monitoring of Adaptive Web Services, 9th IEEE European Conference on Web Services (ECOWS), 2011, pp.109-116.

45. 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 (2013), 114-117.
46. K. Bratanis, D. Dranidis, A. J. H. Simons, SLAs for cross-layer adaptation and monitoring of service-based applications: a case study, International Workshop on Quality Assurance for Service-Based Applications (QASBA), 2011.
47. P. Leitner, A. Michlmayr, F. Rosenberg, S. Dustdar, Monitoring, prediction and prevention of SLA violations in composite services, IEEE International Conference on Web Services (ICWS), 2010, pp. 369-376.
48. P. Leitner, S. Dustdar, B. Wetzstein, 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.
49. D. Ivanovic, M. Carro, M. Hermenegildo, Constraint-based runtime prediction of SLA violations in service orchestrations, Proceedings of the International Conference on Service Oriented Computing (ICSOC), 2011, pp. 62-76.
50. E. Schmieders, A. Micsik, M. Oriol, K. Mahbub, R. Kazhamiakin, Combining SLA prediction and cross layer adaptation for preventing SLA violations, 2nd Workshop on Software Services: Cloud Computing and Applications based on Software Services, 2011, Timisoara, Romania.
51. D. Lorenzoli, G. Spanoudakis, Runtime Prediction of Software Service Availability, International Conference on Software Engineering Research and Practice (SERP'11), 2011, pp. 239-245.
52. ISO/IEC 24765, Software and Systems Engineering Vocabulary, 2006.
53. IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology, http://standards.ieee.org/findstds/standard/610.12-1990.html (Accessed in August 2014).
54. C. Nie, H. Leung, A survey of combinatorial testing, ACM Computing Surveys (CSUR), Volume 43 Issue 2, January 2011.
55. M. Grochtmann, K. Grimm, Classification trees for partition testing, Software Testing, Verification and Reliability, vol. 3 (2) (June 1993), 63–82.
56. M. Grindal, J. Offut, S.F. Andler, Combination testing strategies – a survey, Software Testing, Verification and Reliability Volume 15, Issue 3, (September 2005), 167–199.
57. A. Keller, H. Ludwig, The WSLA Framework: Specifying and Monitoring of Service Level Agreements for Web Services, IBM research report RC22456, 2002.
58. V. Tosic, B. Pagurek, K. Patel, B. Esfandiari, W. Ma, Management applications of the Web Service Offerings Language (WSOL), 15th International Conference on Advanced Information Systems Engineering(CAiSE'03), Velden, Austria, June 2003.
59. D.D. Lamanna, J. Skene, W. Emmerich, SLAng: A language for defining Service Level Agreements, 9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03), San Juan, Puerto Rico, 2003.
60. A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, M. Yalinalp, "Web services policy 1.5 — framework," http://www.w3.org/TR/ws-policy, 04 September 2007.
61. K.T. Kearney, F. Torelli, The SLA Model, Service Level Agreements for Cloud Computing, Springer New York, pp. 43-77, 2011.
62. M. Tian, A. Gramm, T. Naumowicz, H. Ritter, J. Schiller, A concept for QoS integration in Web Services, 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.
63. D. Sabbah, Bringing Grid & Web Service Together, Opening Keynote Globus World 2004, Vice President of Strategy and Technology, IBM Software Group.
64. N.D. Belnap, A useful four-valued logic. In: J.M. Dunn, G. Epstein (eds.), Modern Uses of Multiple-Valued Logic, Dordrecht: Reidel, (1977) 8-37.
65. E.F. Codd, The Relational Model for Database Management - Version 2. Addison-Wesley, Reading, MA, (1990).

66. G. Gessert, Four Valued Logic for Relational Database Systems, Sigmod Rec. 19 (1), (1990) 29- 35.
67. J. Tuya, M.J. Suárez-Cabal, C. de la Riva, Full predicate coverage for testing SQL database queries, Software Testing, Verification and Reliability, 20 (3) (September 2010) 237-288.
68. J. Offut, L. Nan, P. Ammann, X. Wuzhi, Using abstraction and Web applications to teach criteria-based test design, 24th IEEECS Conference on Software Engineering Education and Training (CSEE&T), 2011, pp.227-236.
69. M.B. Cohen, M.B. Dwyer, J. Shi, Interaction Testing of Highly-Configurable Systems in the Presence of Constraints, Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA), 2007, ACM, New York, NY, USA, pp. 129-139.
70. T. Nanba, T. Tsuchiya, T. Kikuno, Constructing test sets for pairwise testing: A SAT-based approach, Second International Conference on Networking and Computing (ICNC), 2011, Nov. 30 2011-Dec. 2 2011, pp.271-274.
71. J.D. McCaffrey, An empirical study of pairwise test set generation using a genetic algorithm, Seventh International Conference on Information Technology: New Generations (ITNG), 2010, 12-14 April 2010, pp.992-997.
72. Software Engineering Research Group (GIIS) SLA downloads: http://giis.uniovi.es/testing/downloads/sla/?lang=en (Accessed in August 2014).
73. RCTA Inc. DO-178-B: Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA), 1992.

Vitae

Marcos Palacios is currently a Teaching Assistant at University of Oviedo, Spain, and a member of the Software Engineering Research Group of that University. He received his PhD in Computing in 2014 and his M. Sc. in Computer Science in 2008, both from the University of Oviedo. He has collaborated with City University London (London, UK) as visiting researcher. His research interests include software engineering, software testing and service-based applications.

José García-Fanjul is currently Professor at University of Oviedo, Spain, and a member of the Software Engineering Research Group of that University. He received his PhD and M.Sc. in Computing from the University of Oviedo. His research interests include software engineering, software testing and service-based applications, and he has authored several research papers published on journals and international conferences.

Javier Tuya is Professor at University of Oviedo, Spain, where is the research leader of the Software Engineering Research Group. He received his PhD in Engineering from the University of Oviedo. He is Director of the Indra-Uniovi Chair, member of the ISO/IEC JTC1/SC7/WG26 working group for the ISO/IEC/IEEE 29119 Software Testing standard and convener of the corresponding AENOR National

Body working group His research interests include software engineering, process improvement, verification & validation and software testing.

George Spanoudakis is Professor of Computing and Associate Dean for Research in the School of Informatics at City University London. His research is in software engineering with a focus on service oriented computing and software systems security where he has published more than 120 peer-reviewed papers. His research has attracted more than €4.8m of funding and has been the principal investigator of several R&D projects. He has served in the committees of several international conferences, and the editorial boards of several journals including the Int. J. of Software Engineering and Knowledge Engineering and Int. J. of Advances in Security.