Coverage-based testing for Service Level Agreements

Marcos Palacios, José García-Fanjul, Javier Tuya, *Member, IEEE*, and George Spanoudakis, *Member, IEEE*

Abstract—Service Level Agreements (SLAs) are typically used to specify rules regarding the consumption of services that are agreed between the providers of the Service-Based Applications (SBAs) and their consumers. An SLA includes a list of terms that contain the guarantees that must be fulfilled during the provisioning and consumption of the services. Since the violation of such guarantees may lead to the application of potential penalties, it is important to assure that the SBA behaves as expected. In this article, we propose a proactive approach to test SLA-aware SBAs by means of identifying test requirements, which represent situations that are relevant to be tested. To address this issue, we define a four-valued logic that allows evaluating both the individual guarantee terms and their logical relationships. Grounded in this logic, we devise a test criterion based on the Modified Condition Decision Coverage (MCDC) in order to obtain a cost-effective set of test requirements from the structure of the SLA. Furthermore by analyzing the syntax and semantics of the agreement, we define specific rules to avoid non-feasible test requirements. The whole approach has been automated and applied over an eHealth case study.

Index Terms—Software Testing, Service Based Applications, Service Level Agreements, Coverage Criterion, MCDC.

1 Introduction

SERVICE Level Agreements (SLAs) are used in the scope of Service Oriented Architectures (SOA) as a standard formalism to specify the conditions that regulate the trading between the service providers and the consumers. These contracts represent a set of guarantee terms that contain the expected Quality of Service (QoS) that must be delivered during the provision and consumption of the services. Generally, the violation of a term of the SLA leads to negative consequences for the stakeholders, for example, the payment of penalty fees. Due to this, it is important that both providers and consumers try their utmost to avoid or minimize the SLA violations and their corresponding consequences.

The detection of these SLA violations is typically performed by observing the behavior of the Service Based Application (SBA) at runtime, recollecting data from the executions and making a decision about the evaluation of the SLA. To do this, different monitoring techniques have been proposed ([1], [2], [3]) and represent a good *post-mortem* solution in the sense that the problems are detected after they have occurred in the operational environment. Such problems may be therefore analyzed and solved so they are less likely to arise again. However, in specific scenarios where an SLA violation may lead to important consequences for the stakeholders it is not recommendable to wait until the problems appear at runtime. In these cases, the application of *ante morten* approaches allows the providers to reduce or avoid the number of SLA violations and, hence, minimize the penalties associated to such violations.

Among the fit-for-purpose tasks involved within the proactive detection of SLAs violations, testing has been identified as a challenge in the context of SOA-based research [4], [5], [6]. The objective of SLA-based testing is, on the one hand, to guarantee that the SBA satisfies the conditions specified in the SLA and, on the other hand, to assure that such SBA is able to behave properly even when some of its constituent services violate the SLA. For example, a service may not be able to fulfill the agreed response time (perhaps because such service is down). As the response time is a condition specified in the SLA, we aim at identifying tests that exercise the situations where the service does not answer or it spends too much time to give the response. With these tests, we check that the application provides appropriate mechanisms to handle the unexpected behavior of the aforementioned service. At this stage, the SLA-based testing aims at anticipating as much as possible the detection of problems in the SBA and thereby avoid the consequences derived when such problems arise at runtime in the operational environment.

In a previous work we addressed the identification of test requirements by analyzing the individual guarantee terms of the SLA [7]. These test requirements represent error-prone situations that are interesting to be tested. In this article we provide a further step by means of considering the whole logical structure of the SLA. As the number of test requirements may become unmanageable if the SLA is quite complex, we devise a coverage-based criterion with the aim at obtaining a reduced but effective set of test requirements.

- *MarcosPalacios is with the Department of Computer Science, University of Oviedo. Campus Universitario de Gijón. 33204. Asturias. Spain. E-mail: palaciosmarcos@uniovi.es.*
- *José García-Fanjul is with the Department of Computer Science, University of Oviedo. Campus Universitario de Gijón. 33204. Asturias. Spain. E-mail: jgfanjul@uniovi.es.*
- *Javier Tuya is with the Department of Computer Science, University of Oviedo. Campus Universitario de Gijón. 33204. Asturias. Spain. E-mail: tuya@uniovi.es.*
- *George Spanoudakis is with the Department of Computing, School of Informatics, City University London. EC1V OHB. London. E-mail: G.E.Spanoudakis@city.ac.uk*

The primary contributions of this article are:

1. Specification of a test criterion based on the MCDC coverage criterion [8] that allows the identification of test requirements by means of analyzing the information represented in both the guarantee terms and their logical combinations. This criterion makes use of a four-valued logic to evaluate SLAs, which is also defined.
2. Definition of specific rules that contribute to avoid the identification of non-feasible situations, considering the context of the SBA as well as the hierarchical structure of the SLA.
3. Automation of the process that identifies the test requirements using the aforementioned criterion.
4. Application of the criterion to a real and critical eHealth scenario that was proposed in the context of the PLASTIC European Framework [9] and used by other authors to test SLAs [10], [11]. This case study will also be used as a running example along the article.

The structure of the article is as follows. Section 2 provides a background about SLAs and the main concepts about software testing. Section 3 describes the logic to evaluate the elements of the SLA. Section 4 defines the criterion that allows obtaining test requirements from the logical associations of terms contained in the SLA. Section 5 presents the results obtained through the application of the approach in a real scenario. Section 6 outlines related work. Finally, Section 7 provides some conclusions and outlines the future work.

2 Background

In this section we describe the most relevant characteristics of SLAs, focusing on the WS-Agreement standard language and we introduce relevant concepts that are commonly used in the scope of software testing.

2.1 Service Level Agreements

Service Level Agreements (SLAs) are contracts that specify the rules for the trading between the consumers and the Service Based Applications (SBAs) providers. Typically, these rules specify which the constituent services of the SBA that will be regulated by the agreement are, and how these services should be offered. Currently, very important companies (including Google, Microsoft, AT&T, Amazon or HP) use SLAs as a guarantee for their clients to assure that their SBAs deliver the expected Quality of Service (QoS). Although the existing SLAs in the industrial domain [12] [13] seem to be quite simple nowadays, they could become more complex by means of establishing relationships between the terms or including information regarding the functional and non-functional features of the services as well as the penalties derived from the violations of the agreed guarantees. In this article we aim at testing service based applications when the SLA that regulates their behavior is not simple.

In addition to typical tasks involved within the management of the SLAs, including negotiation [14], [15], evaluation [16], optimization [17] [18] monitoring [1] or testing [19], [20], the specification of the SLAs has been widely studied over the last few years. In many occasions the SLAs are specified in documents without any kind of format or even using natural language. Unfortunately, this lack of formalism when creating an SLA hinders the automatic management of the agreement. In our case, the testing of the SLAs requires using such documents as the test basis so we need to somehow formalize the specification of the SLA in order to automate as much as possible the obtaining of tests.

Among the languages that have been published in order to standardize the content of the SLAs, such as, WSLA, WSLO, SLANG, WS-QoS or WS-Policy, it has been WS-Agreement the one that has received more attention regarding the SLA-based testing, at least from the academic scope. WS-Agreement presents a generic syntax that allows extrapolating its derived outcomes to any other existing SLA specification language. In fact, WS-Policy, which is gaining attraction from the industrial space, shares the same notation as WS-Agreement to represent the relationships between the guarantees. Thus, in this work, we focus on the syntax and semantics of WS-Agreement in order to test the conditions represented in the SLA.

WS-Agreement at a glance

WS-Agreement [21] is an XML based language proposed by the Open Grid Forum (OGF) that specifies a protocol for the establishment of agreements between two parties. This standard allows defining a hierarchical structure for the specification of an SLA. The specification of an SLA using the WS-Agreement standard language is composed of three main parts. These are:

- Name, i.e., the part specifying an optional name that can be given to the agreement.
- Context, i.e., the part defining the parties involved in the agreement and their role.
- Terms, i.e., the part expressing the negotiated and agreed obligations of each party. These obligations are specified using Service Description Terms (SDT), Service Properties (SP) and Guarantee Terms (GT).

The most important information of the SLA is represented by means of the Guarantee Terms, which describe the obligations that must be satisfied by a specific obligated party. A *Guarantee Term* (GT) contains the following internal elements: (1) the *Scope* specifies the list of services the term applies to, (2) the *Qualifying Condition* (QC) represents a precondition or assertion that determines whether the term is relevant and must be considered during the evaluation process, (3) the *Service Level Objective* (SLO) specifies the guarantee that must be met.

In WS-Agreement, the terms of the SLA can be logically and hierarchically grouped by means of Compositor elements. Specifically, there are three different types of compositor elements, namely: *All*, *OneOrMore* and *ExactlyOne*. These element types are equivalent to the logical AND, OR and XOR operators, respectively.

2.2 Software Testing

In the context of service computing, the testing of SLA-

aware SBAs has been posed as a challenging task [4], [5], [6]. Generally, testing is an activity in which the Software Under Test (SUT) is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system [22]. The execution of the SUT is usually performed through the design and execution of test cases.

A test case is a set of inputs, execution conditions and expected results developed for a particular objective [23]. The detection of faults is addressed by means of executing the SUT and comparing the observed results with the expected results, determining whether the behaviour of the software is correct or not. It is therefore imperative that a good design of test cases should allow detecting the highest possible number of faults. The generation and execution of test cases is considered a proactive or *ex ante* approach in the sense that it is able to detect problems in the SUT before such problems occur in the operational environment.

Monitoring is also a widely used testing technique that passively observes real time executions with the aim of detecting any deviation from the expected behaviour of the software during its operation [24]. Monitoring based approaches are considered reactive because problems are detected *ex post*, after they have occurred and, thus, potential further consequences cannot always be avoided.

Concerning these two main testing approaches, a test requirement represents a specific feature or situation of the SUT that must be satisfied or covered during testing [25]. Test requirements are typically identified following a specific test strategy, which might be based on different factors such as risks, models of the system, expert advice or heuristics. In this scope, the identification of test requirements from logical conditions is not trivial, as the number of combinations can be often huge. Testing all combinations may be unmanageable if not impossible altogether. Hence, it is necessary to decide the expected coverage and, based on this, define test criteria in order to provide a systematic way of selecting the best test requirements.

In this article the SUT is any SBA, typically a web service composition, in which the execution conditions of the constituent services are specified in a SLA using the WS-Agreement language. In our approach, we are dealing with the controllability of the services, which is a well known issue that limit the testability in SOA based system, by means of proposing a proactive approach. This means that the identification of test requirements allows guiding the generation of test cases and such test cases will be executed in a controlled environment so, in this case, the services are under our control. On the other hand, when the services are deployed in their operational environment and are consequently out of our control, we consider that the identified test requirements could also be used to define a monitoring plan in order to apply a reactive approach [26]. By observing the behavior at runtime and analyzing the exercised test requirements, it is possible to detect whether the SBA has evolved and new tests need to be designed or the SLA has become obsolete and need to be changed accordingly.

3 Evaluation of SLAs

The evaluation of an SLA requires analyzing the information gathered from the monitors and/or testers, checking the specification of the guarantee terms and, finally, making a decision about the fulfilment of such terms. We have outlined in Section 2 that an SLA specifies a set of terms that govern the execution of the constituent services of the SBA. Such guarantee terms can be hierarchically structured by means of using specific compositor elements (All, OneOrMore, ExactlyOne).

In this context, we identify two different levels regarding the evaluation of the SLAs.

- Level I: Individual Guarantee Terms.
- Level II: Composite Guarantee Terms defined by compositor elements.

The first level involves making a decision about the fulfilment of each individual guarantee term represented in the SLA [20]. The second level involves considering sets of Guarantee Terms logically grouped by the compositor elements and determining whether the composite terms are fulfilled or not.

In this section we propose a logic that allows evaluating both individual guarantee terms and composite terms, from a testing point of view, including all the potential situations derived from the task of assuring whether the SLA is being fulfilled or not.

3.1 Level I: Evaluating Individual Guarantee Terms

We firstly focus on each individual guarantee term in order to address the evaluation of the SLA. The evaluation of a guarantee term is usually performed in a dichotomic way, for example, depicting a two-way traffic light indicator (green/red) that indicates whether the term has been fulfilled or violated respectively. The use of these two classical values is really useful when the behaviour of the SBA is monitored at runtime in order to decide whether a problem has been detected, disregarding the situation that has caused such problem. However and from a testing point of view, we need to early identify the different potential situations that may derive in problems in the SBA.

Aligning this perspective with the syntax of WS-Agreement, a Guarantee Term is specified by means of the internal elements Scope, Qualifying Condition (QC) and Service Level Objective (SLO). After taking this syntax into account, a Guarantee Term can be evaluated as:

- **Fulfilled (F)** - if and only if the methods of the services specified in the Scope have been executed, the QC has been met and the SLO has been satisfied.
- **Violated (V)** - if and only if the methods of the services specified in the Scope have been executed, the QC has been met and the SLO has not been satisfied.

However, from a testing point of view, this two-valued logic may not be enough to evaluate the potential situations derived from the guarantee term. For example, situations where there are methods of the services associated to the scope of a guarantee term that have not been executed. Considering such situations introduces the need for an additional evaluation value, under which a Guarantee Term is evaluated as:

- **Not Determined (N)** - if and only if there are methods of the services specified in the Scope of the guarantee term, which have not been executed.

Furthermore, analyzing the internal elements of a Guarantee Term and its semantics according to WS-Agreement, we have to consider another case. This case arises when the Qualifying Condition of the term is not met during the execution of services. In this case, the Guarantee Term becomes irrelevant and it must not be taken into account for the purpose of the evaluation of the SLA so we say that a Guarantee Term is evaluated as:

- **Inapplicable (I)** - if and only if the methods of the services specified in the Scope have been executed but the Qualifying Condition has not been satisfied.

In other disciplines within the software engineering, it has been necessary to extend the typical binary logic (true / false) to deal with similar situations. For example, in the context of Database Management Systems (DBMS) the interpretation of the missing information is considered by means of a special third value (i.e., null), which has also been broadly used in the scope of database applications testing [27], [28], [29]. In our case, the use of these two additional evaluation values (Not Determined and Inapplicable) could represent an analogous interpretation of the treatment of the null value in DBMS and leads to a four-valued logic to evaluate SLAs.

Hence, a Guarantee Term denoted by t can be evaluated with four different evaluation values as output using a function ev(t):

$$ev(t) = \{\boldsymbol{F}ullfilled, \boldsymbol{V}iolated, \boldsymbol{N}ot\ Determined, \boldsymbol{I}napplicable\}$$

3.2 Level II: Evaluating Compositor Elements

After having provided a systematic way to evaluate the individual guarantee terms, now we focus on the logical combinations of the SLA Guarantee Terms. We have previously outlined that an SLA specified in WS-Agreement represents a hierarchical structure of guarantee terms, logically combined using the specific Compositor Elements All, OneOrMore and ExactlyOne. Thus, we need to complete the logic basis that allows evaluating the individual Guarantee Terms in order to unequivocally determine the evaluation value of these compositors. The use of a four valued logic in our approach allows us, on the one hand, to obtain the evaluation outcomes of the SLA and, on the other hand, to guide the identification of the test requirements by means of applying a coverage criterion.

According to the semantics of each compositor, herein we define the following logic to evaluate the SLA.

Evaluation of *All Compositor*

An All compositor element with multiple Guarantee Terms is evaluated as follows:

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

Fulfilled if
$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{F}) \wedge$
$(\forall j \in [1,n]\ and\ j \neq i : (ev(t_j) = \boldsymbol{F}\ \vee\ ev(t_j) = \boldsymbol{I}))$
Violated if
$\exists i \in [1,n] : ev(t_i) = \boldsymbol{V}$
Not Determined if
$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{N}) \wedge (\nexists j \in [1,n] : ev(t_j) = \boldsymbol{V})$

TABLE 1
COMPOSITOR ELEMENTS TRUTH TABLE

GT1	GT2	All	OneOrMore	ExactlyOne
F	F	F	F	V
F	V	V	F	F
F	N	N	F	N
F	I	F	F	F
V	F	V	F	F
V	V	V	V	V
V	N	V	N	N
V	I	V	V	V
N	F	N	F	N
N	V	V	N	N
N	N	N	N	N
N	I	N	N	N
I	F	F	F	F
I	V	V	V	V
I	N	N	N	N
I	I	I	I	I

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

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

It is worth mentioning that a WS-Agreement always specifies the content of the whole agreement under an All external compositor element so the evaluation of the SLA would be equivalent to the evaluation of such most external All element.

Evaluation of OneOrMore Compositor

Likewise, an OneOrMore compositor element with multiple Guarantee Terms is evaluated as follows:

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

Fulfilled if
$\exists i \in [1,n] : ev(t_i) = \boldsymbol{F}$
Violated if
$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{V}) \wedge$
$(\forall j \in [1,n] : (ev(t_j) = \boldsymbol{V}\ \vee\ ev(t_j) = \boldsymbol{I}))$
Not Determined if
$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{N}) \wedge (\nexists j \in [1,n] : ev(t_j) = \boldsymbol{F})$
Inapplicable if
$\forall i \in [1,n] : ev(t_i) = \boldsymbol{I}$

Evaluation of ExactlyOne Compositor

Finally, an ExactlyOne compositor element with multiple Guarantee Terms is evaluated as follows:

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

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

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

Not Determined if
$(\exists i \in [1,n] : ev(t_i) = \boldsymbol{N}) \wedge$
$(\nexists j,k \in [1,n], j \neq k : (ev(t_j) = ev(t_k) = \boldsymbol{F}))$

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

In order to illustrate the application of this logic, the truth table of these three compositor elements with two Guarantee Terms each one (GT1 and GT2) is represented in Table 1. The first two columns contain all the potential combinations of the evaluation values of both guarantee terms. Each cell of the last three columns specifies the final evaluation value of each compositor element when its internal guarantee terms are evaluated with the values represented in such row (for example, if GT1 is evaluated as Fulfilled and GT2 is evaluated as Inapplicable, the All compositor is evaluated as Fulfilled).

3.3 From Evaluation Values to Test Requirements

The aforementioned logic allows us to obtain the evaluation values of both Guarantee Terms and Compositors based on the information gathered from the execution of the services. So from now on:

Definition 1. *An evaluation value is the output provided by the mechanism in charge of making a decision about the fulfillment of a guarantee term or a compositor. There are four different evaluation values: (F) Fulfilled, (V) Violated, (I) Inapplicable and (N) Not Determined.*

These values are used as a cornerstone to identify the test requirements that will be later covered with the test cases. In a previous work we dealt with the identification of test requirements by means of analyzing the information contained in the individual guarantee terms [7]. In this article, we are providing a further step in the sense that we tackle the testing of the logical combinations of the terms represented through the compositors.

At this stage, we introduce the definition of a test requirement based on the evaluation values:

Definition 2. *A test requirement represents a situation exercised when executing the SUT in which each Guarantee Term has to take a predetermined and specific evaluation value.*

The relation between a test requirement and its constituent evaluation values is explained using an example from the eHealth scenario, depicted in Fig. 1. In the left part of the figure an All compositor element with two guarantee terms is represented in WS-Agreement. From the information contained in such compositor we try to identify potential error-prone situations that could be interesting to test. To address this task, we force the first guarantee term (GT_Device1) to take the Inapplicable evaluation value whereas the second guarantee term (GT_Device2) is evaluated as Violated, then the *All* compositor will be evaluated as Violated. From these evaluation values a test requirement is identified (bottom right part of the figure). To be more specific, we are testing the behaviour of the eHealth system when a doctor is managing an Emergency and one of the devices (*device1*) is not queried (GT_Device1 = Inapplicable) whereas the other service (device2) is not working properly according to the SLA (GT_Device2= Violated). Despite of the violation of

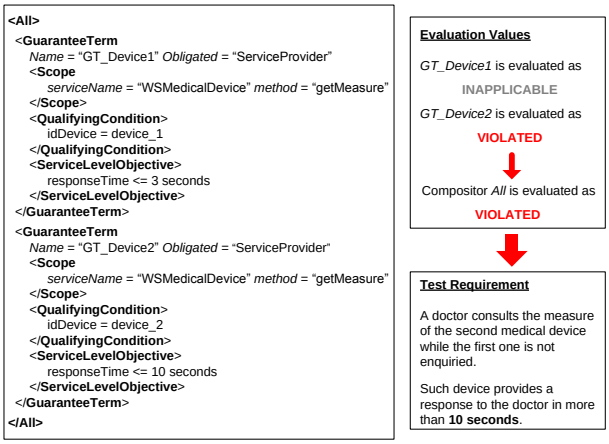

Fig. 1. Relation between Evaluation Values & Test Requirement

one guarantee term, the service composition must be able to continue its execution and provide a solution to the patient's incidence. If there was a problem when dealing with an unexpected behaviour of one medical device as described, the exercitation of the identified test requirement would allow us to uncover it before deploying the eHealth system in the production environment.

4 Coverage-Based Test Criterion

The process of testing SLA-aware SBAs can be improved by identifying test requirements from the specification of the SLAs using a criterion based on the principle of the Modified Condition / Decision Coverage (MCDC) [8] that allows obtaining a cost-effective set of test requirements, representing situations that are interesting to exercise regarding the SLA and the SBA. This set contains a reduced number of test requirements to be exercised in order to uncover problems in the SBA.

Typically, MCDC is applied to a specification of the SUT. In our case, the specification that says how the SUT must behave is the SLA. In such SLA there are guarantee terms that represent conditions that can be satisfied or not. Hence, it is necessary to design tests that exercise situations in which the guarantee terms are fulfilled and situations in which not. Within this approach, these situations are obtained by means of the application of our MCDC-based criterion.

4.1 Four-valued MCDC Test Criterion

Once the logic to evaluate the compositor elements of an SLA has been defined, we obtain the test requirements by combining the potential evaluation values of the terms included in the compositors. Considering that each term can be evaluated with four different values, the amount of test requirements would grow exponentially with the number of terms if we applied all the possible combinations. Hence, our objective is to avoid the exponential growth of test requirements in order to obtain a reduced but cost-effective set of test requirements and we achieve it by using MCDC.

Modified Condition Decision Coverage (MCDC), defined in the RTCA/DO-178B standard [8], is a broadly studied structural coverage criterion [31], [32]. It has also

been used for test suite reduction and prioritization [33] because it provides a linear increase in the number of test requirements [30]. MCDC is a criterion that falls between condition/decision and multiple condition coverage [32]. This criterion has been shown to represent a good balance of test-set size and fault detecting ability simultaneously [34], [35]. MCDC is defined as a conjunction of the following requirements:

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

Consider the decision d = (a AND b) where a, b are two boolean conditions. To satisfy MCDC criterion, we need to generate three test cases (0,1) (1,1) (1,0) as described in Fig. 2.

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

Definition 3. *A set of test requirements satisfies the SLACDC (SLA Condition / Decision Coverage) criterion for a combination of terms grouped within a compositor using the four-valued logic if and only if:*

1. *Every guarantee term has taken all possible evaluation values at least once.*
2. *The compositor has taken all possible evaluation values as outcome at least once.*
3. *For each possible evaluation value of a guarantee term, a variation from a specific evaluation value to a different value has been shown to independently affect the evaluation of the compositor (this is, when we switch the evaluation value of the guarantee term while holding fixed the evaluation values of the rest of terms, the outcome of the evaluation of the compositor varies).*

As an example, consider an All compositor element with two guarantee terms (GT1 and GT2) represented in Fig. 3. To address the identification of test requirements, we start from the situation where both guarantee terms are evaluated as Fulfilled and, thus, the All compositor is also evaluated as Fulfilled (row 1 in the figure). Then, we set the second row obtaining the first pair (a), which allows us to switch the evaluation value of GT1 from Fulfilled to Violated and this change affects the evaluation value of the compositor, which also changes from Fulfilled to Violated. After this, we set the third row obtaining a new pair (b), where the evaluation value of GT1 switches from Violated to Inapplicable and, consequently, the evaluation value of the All compositor changes from Violated to Fulfilled.

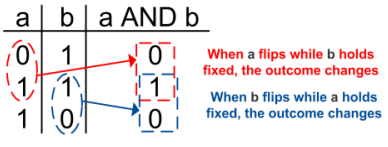

Fig. 2. Example of application of MCDC

This process continues until we obtain six different pairs (*a* to *f*) that fulfil the conditions (1) and (3) of SLACDC criterion (Definition 3). However condition (2) is not fulfilled because the All compositor has not been evaluated as Inapplicable yet. In order to satisfy condition (2) we identify a new test requirement (row 8) where both guarantee terms are evaluated as Inapplicable and, thus, the All compositor also takes the Inapplicable evaluation value. At this stage, a final set of 8 test requirements (TR1-TR8) is obtained (Fig. 3) that satisfy the criterion instead of the 16 test requirements that would be obtained using a complete combination.

4.2 Identification of Test Requirements

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

All Compositor

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

1. ***Initialize the set with an initial test requirement (TR1) where all the guarantee terms are evaluated as Fulfilled.***
2. ***For each GT_i in the All_Compositor:***
   Add a new test requirement by means of switching the evaluation value of GT_i from Fulfilled (as it is in TR1) to (Violated, Inapplicable, Not Determined) while the evaluation of GT_j with $j \neq i$ remains fixed to Fulfilled.
3. ***Add a new test requirement where all the guarantee terms are evaluated as Inapplicable in order to get the Inapplicable evaluation value in the All_Compositor.***

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

Step1:

The set of test requirement is initialized with TR1 where all the terms are evaluated as Fulfilled.

Step2:

For each guarantee term we add three test requirements where the evaluation value of each guarantee term

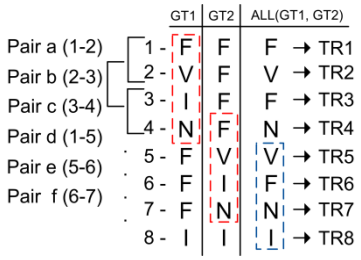

Fig. 3. MCDC for SLA GTs and Compositors with a four-valued logic

must be switched from Fulfilled to (Violated, Inapplicable, Not Determined) while holding the rest of terms fixed with Fulfilled. The set of test requirements identified in this step is represented in Table 2 (TR2-TR10).

Step3:

We identify a new test requirement where all the guarantee terms are evaluated with the Inapplicable value.

The final set of test requirements identified for this compositor is represented in Table 2. The first column labels each test requirement and the evaluations of the individual guarantee terms (GT) and the compositor are represented in the rest of the columns. The first row is remarked because it corresponds to the initial test requirement and the cells that represent the guarantee terms that switch their evaluation values are grey shaded.

The application of SLACDC criterion provides a linear number of combinations related to the number of conditions. In general, the number of combinations that satisfies MCDC for a binary logical decision is (n+1) where n is the number of conditions within the decision, there are two possible truth values (true/false) for each condition and the maximum number of combinations is 2*n [36]. In our case and dealing with a four-valued logic for the evaluation of the guarantee terms, the number of test requirements obtained with SLACDC criterion remains linear regarding the number of guarantee terms and evaluation values and can be obtained according to the following formula:

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

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

OneOrMore Compositor

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

1. ***Initialize the set with an initial test requirement (TR1) where all the guarantee terms are evaluated as Violated.***
2. ***For each GT_i in the OneOrMore_Compositor:***
   Add a new test requirement by means of switching the evaluation value of GT_i from Violated (as it is in TR1) to (Fulfilled, Inapplicable, Not Determined) while the evaluation of GT_j with $j \neq i$ remains fixed to Violated.
3. ***Add a new test requirement where all the guarantee terms are evaluated as Inapplicable in order to get the Inapplicable evaluation value in the OneOrMore_Compositor.***

TABLE 2
TEST REQUIREMENTS FOR AN ALL COMPOSITOR WITH 3 GUARANTEE TERMS

Test Req.	ev(GT_1)	ev(GT_2)	ev(GT_3)	ev(ALL)
TR1	F	F	F	F
TR2	V	F	F	V
TR3	I	F	F	F
TR4	N	F	F	N
TR5	F	V	F	V
TR6	F	I	F	F
TR7	F	N	F	N
TR8	F	F	V	V
TR9	F	F	I	F
TR10	F	F	N	N
TR11	I	I	I	I

We have omitted the explanation of the steps that perform the identification of test requirements for this compositor because the process is the same as for the All compositor. As an example, the test requirements identified for an OneOrMore compositor with 3 guarantee terms can be seen in the first rows of Table 3 (rows 1-11).

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

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

ExactlyOne Compositor

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

1. Test the combinations where the evaluation value of the compositor varies due to the flip from none term evaluated as fulfilled to only one term fulfilled.
2. Test the combinations where the evaluation value of the compositor varies to Violated because the flip involves the fulfilment of more than only one guarantee term.

The first scenario exercises the situation where all the guarantee terms are initially evaluated as Violated and we switch the evaluation value of each guarantee term to (Fulfilled, Inapplicable and Not Determined). Hence, it can be seen that this first scenario is exercised using the same set of test requirements that we have described for the OneOrMore compositor. This means that the algorithm (A1) to test this first scenario is the same and the test requirements obtained are represented in Table 3 (rows 1-11).

To exercise the second scenario, we have to obtain test requirements where there is already a unique guarantee term evaluated as Fulfilled and we flip the evaluation of another guarantee term between the four possible evaluation values. The algorithm (A2) for the identification of these test requirements is as follows:

1. ***Initialize an empty set of test requirements.***
2. ***For each GT_i in the ExactlyOne_Compositor:***
   a. ***Add an initial test requirement where one guarantee term GT_j with $j \neq i$ is evaluated as Fulfilled and the rest of guarantee terms are evaluated as Violated.***
   b. ***Add a new test requirement by means of switching the evaluation value of GT_i from Violated (as it is in the current initial test requirement) to (Fulfilled, Inapplicable, Not Determined) while the evaluation of GT_j with $j \neq i$ remains fixed to Fulfilled and the evaluation of the rest of terms remains fixed to Violated.***
3. ***Add a new test requirement where all the guarantee terms are evaluated as Inapplicable in order to get the Inapplicable evaluation value in the ExactlyOne_Compositor.***

The test requirements obtained with this algorithm

(A2) are represented in Table 3 (rows 12-24). The cells that contain the initial test requirement of step 2 for each guarantee term are remarked.

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

In Table 3 we have joined the set of test requirements obtained through the algorithm A1 and the algorithm A2 and we have marked the duplicated test requirements. In the first column we identify with a number all the test requirements obtained with both algorithms. In the second column we set an identifier to the final test requirement or a brief description about the reason for removing such test requirement. In the rest of column the evaluation values of the guarantee terms and compositor are represented. Furthermore, we have remarked the rows that represent the initial test requirements in each algorithm and those cells where the evaluation value of the guarantee term is switched (grey shaded).

After joining both sets and removing the duplicated test requirements, a final number of 19 test requirements are identified. This number is obtained through the formula:

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

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

TABLE 3
TEST REQUIREMENTS FOR AN EXACTLYONE COMPOSITOR WITH 3 GUARANTEE TERMS

Row	Test Req.	ev(GT_1)	ev(GT_2)	ev(GT_3)	ev(ExOne)
1	TR1	V	V	V	V
2	TR2	F	V	V	F
3	TR3	I	V	V	V
4	TR4	N	V	V	N
5	TR5	V	F	V	F
6	TR6	V	I	V	V
7	TR7	V	N	V	N
8	TR8	V	V	F	F
9	TR9	V	V	I	V
10	TR10	V	V	N	N
11	TR11	I	I	I	I
12	Duplicated (TR5)	V	F	V	F
13	TR12	F	F	V	V
14	TR13	I	F	V	F
15	TR14	N	F	V	N
16	Duplicated (TR2)	F	V	V	F
17	Duplicated (TR12)	F	F	V	V
18	TR15	F	I	V	F
19	TR16	F	N	V	N
20	Duplicated (TR2)	F	V	V	F
21	TR17	F	V	F	V
22	TR18	F	V	I	F
23	TR19	F	V	N	N
24	Duplicated (TR11)	I	I	I	I

4.3 Removing Non-feasible Test Requirements

The application of the aforementioned algorithms provides a set of test requirements that satisfies the SLACDC criterion to the logical combinations of terms expressed by means of the compositors. However some of the identified test requirements correspond to situations that may be non-feasible to exercise due to the semantic information contained in the guarantee terms. Hence, we have to deal with these specific situations in order to refine the tests previously obtained. To address this improvement we define a set of rules that allow modifying the test requirements that do not make sense and obtain other test requirements that represent feasible and interesting situations to be tested.

These rules are defined to keep fulfilled, as much as possible, the conditions (1) and (2) of the criterion (Definition 3) whereas the condition (3) needs to be relaxed. However, it cannot be assured that these conditions will finally be fulfilled in the resultant set of tests requirements due to the dependencies between the conditions specified in the SLA.

The application of the rules involves identifying the test requirements that are non-feasible in which certain evaluation values will be modified to obtain feasible test requirements. This process requires that more than one evaluation value is switched within the same test requirement so SLACDC criterion is based on a specific form of MCDC named Masking MCDC, investigated by Chilenski [36], which allows more than one condition to vary at once ensuring that only the condition of interest influences the outcome.

Rule 1: Guarantee Terms without Qualifying Condition

This first rule is applied when some of the guarantee terms included in the compositor does not have Qualifying Condition. In this case, the test requirements where such term is evaluated as Inapplicable must be removed. This means that:

Given a Compositor that contains $(\bigcup_{i=1}^{n} GT_i)$:

$$if\ (\exists i \in [1,\mathrm{n}] : \nexists QC(GT_i)\ \Rightarrow\ (ev(GT_i) \neq\ I)$$

In Fig. 4 an example of the application of this rule over the eHealth scenario is depicted. There is an All compositor with two internal guarantee terms. The first of them (GT1) does not have Qualifying Condition so the test requirements where this term is evaluated as Inapplicable are removed. In the right part of the figure all the guarantee terms obtained for the All compositor are represented. The test requirements where the current rule is applied are crossed out so we finally obtain a set of six test requirements instead of the original set of eight test requirements.

Rule 2: Guarantee Terms with same Scope

This second rule is applied when there are guarantee terms in a compositor that are related to the same method and service (Scope). In this case, the test requirements that include these terms contain coupled conditions (in MCDC conditions that cannot be varied independently are said to be coupled [32]) or, in SLACDC criterion, bet-

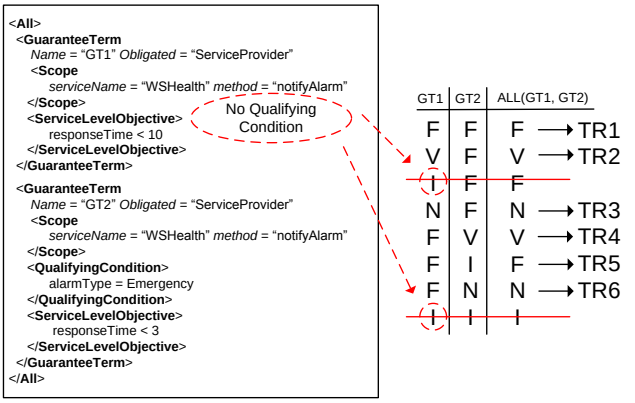

Fig. 4. Example of the Application of Rule 1.

ter named as coupled guarantee terms. This implies that if one of these terms is evaluated as Not Determined (the method/service is not invoked), then the other term must also be evaluated as Not Determined. This is:

Given a Compositor that contains $(\bigcup_{i=1}^{n} GT_i)$:

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

At this stage, if we have identified non-feasible test requirements due to dependencies between the scopes of a pair of involved guarantee terms, we have to modify the evaluation value of one of these guarantee terms. The procedure we follow to change this value aims at keep fulfilling as much as possible the condition (1) of Definition 3, bearing in mind that conditions (2) and (3) could be then relaxed. Then, the evaluation values Fulfilled / Violated / Inapplicable will be the candidates to be modified because, by construction, they are much more common than the other value Not Determined.

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

To illustrate the application of this rule, we consider an example of an All compositor with three guarantee terms (GT3, GT4, GT5), all of them affecting the same method / service (represented in the left part of Fig. 5.). The set of test requirements identified using the All compositor algorithm is represented in the first table within the top of the figure. From this requirements and applying this rule, we modify the specification of the test requirements 4, 7 and 8 in order to modify the non-feasible situations represented in such requirements. In the right part of the figure, we remark the involved guarantee terms in the modification, we underline the evaluation value that has been modified in each change and we cross out the removed test requirements for being duplicated. Finally, the resulting set of test requirements is represented in the bottom right part of the figure. Despite of having modified the evaluation values in some test requirements, it is remark-

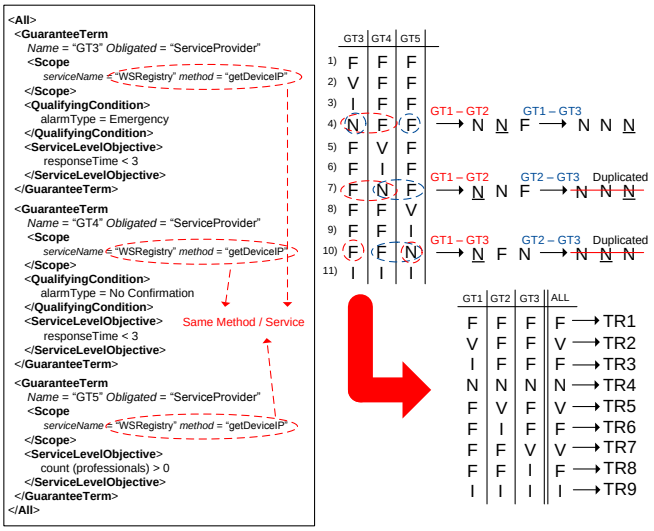

Fig. 5. Example of the Application of Rule 2.

able that, in this example, conditions (1) and (2) of the criterion are still being fulfilled whereas condition (3) has been relaxed for having switched more than one evaluation value in the same test requirement.

Rule 3: Guarantee Terms that have exactly the same QC

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

Given a Compositor that contains $(\bigcup_{i=1}^{n} GT_i)$:

$$if\ (\exists i,j \in [1,\mathrm{n}] : (QC(GT_i) = QC(GT_j)) \wedge$$
$$(ev(GT_i) = I)) \implies (ev(GT_j) \in \{I, N\})$$

As we specified for the previous rule, we have to modify the test requirements that contain these non-feasible combinations. Here again, we relax the condition (3) of the SLACDC criterion but trying to respect conditions (1) and (2) as much as possible.

To achieve this, we select the test requirements where this rule needs to be applied. As in the previous rule, the evaluation values Fulfilled and Violated are more usual than the Inapplicable so these are the values that will be modified to Inapplicable. Here again, this variation must be repeated for each pair of terms that contains the same Qualifying Condition within the compositor and resultant duplicated test requirements should be removed.

To illustrate the application of this rule, we use an example of an All compositor with three guarantee terms (represented in the left part of the Fig. 6) that affect different services. Two of these terms (GT4 and GT6) specify the same condition in the Qualifying Condition element. Once we have identified the set of test requirements by means of applying the algorithm for the All compositor,

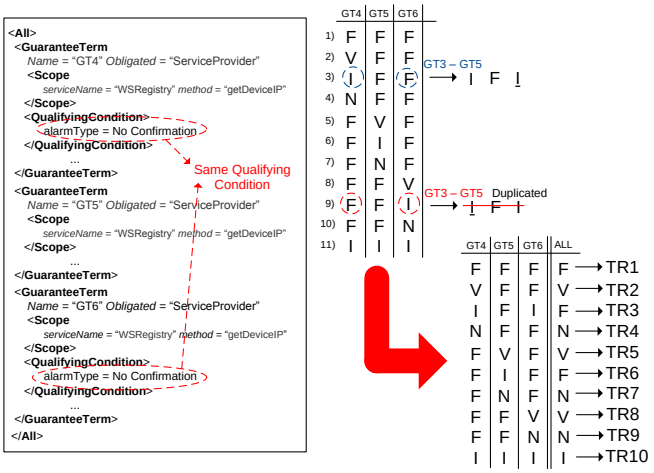

Fig. 6. Example of the Application of Rule 3.

we have to select and modify those requirements that contain any of the non-feasible aforementioned combinations (test requirements 3 and 9). In the right part of the figure, we perform the modifications, indicating the involved guarantee terms and crossing out the removed test requirement for being duplicated.

Rule 4: Guarantee Terms that have mutually disjoint QCs

This rule arises when, in a compositor, there are guarantee terms that contain Qualifying Conditions that are mutually disjoint. This means that, if the Qualifying Condition of one term is met then the Qualifying Condition of the other term must not be met. Regarding the non-feasible test requirements, if one of these terms is evaluated as Fulfilled or Violated in a test requirement then the other one term must be evaluated as Inapplicable or Not Determined. This is:

$Given\ a\ Compositor\ that\ contains\ (\cup_{i=1}^{n} GT_i)$:

$$if\ ((\exists i,j \in [1,n] : QC(GT_i) = !\,QC(GT_j)) \wedge (ev(GT_i) \in \{F,V\})) \implies ev(GT_j) \notin \{F,V\}$$

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

In order to avoid the appearance of the non-feasible combinations in the final test suite, we have to modify the test requirements that contain such combinations. The procedure is similar to the one performed in the previous pair of rules (Rule2 and Rule3). In fact, this rule is practically the opposite as Rule3. Here again, we will change from the most common Fulfilled or Violated evaluation values to another appropriate value. This could be both Inapplicable and Not Determined although we decide to switch to Inapplicable because the Not Determined value also affects the application of Rule2 and, in that case, we would have to apply again such rule so this may become an ineffective loop.

According to this principle, we search the involved test requirements. By construction, in each test requirement there is a guarantee term whose evaluation value varied (named *pivot GT*) while the evaluation values of the other

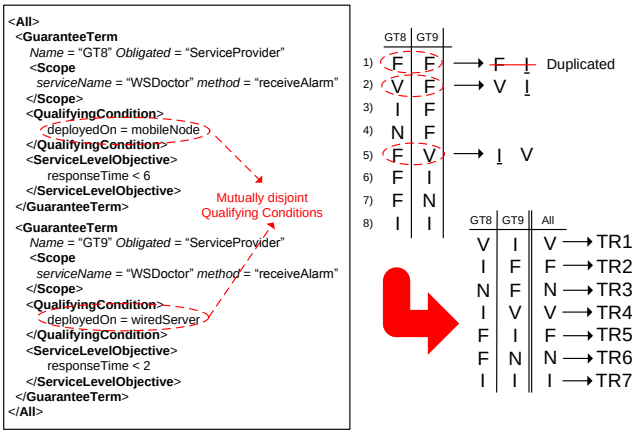

Fig. 7. Example of the Application of Rule 4.

terms remained fixed. If the pair of terms that have mutually disjoint QC includes the pivot GT, then we always modify the evaluation value of the other term from Fulfilled or Violated to Inapplicable. On the other hand, if the pair of terms does not include the pivot GT, then we could modify the evaluation value of any of the two terms. As always, we have to repeat the process for each pair of terms that appear in the test requirement and remove the test requirements that become duplicated.

In Fig. 7 we show the application of this rule for an All compositor with two internal guarantee terms that present two mutually disjoint Qualifying Conditions in their specifications. In the example, the first four test requirements were obtained by holding fixed the value Fulfilled in the second guarantee term while switching the value of the first guarantee term (so the pivot GT is GT8). Hence, in test requirements 1 and 2 we change the value of the second guarantee term from Fulfilled to Inapplicable as explained before. In test requirements 5 the pivot GT is GT9 so we modify the evaluation value Fulfilled of GT8 to Inapplicable.

4.4 Derivation of Test Cases

A test case specifies a set of steps that involve different executions of the constituent services of the application. These steps are defined by using the information represented in the test requirements that are exercised in such test case. In each of these steps the guarantee terms and compositors need to be evaluated so the expected output regarding the evaluation of the SLA is automatically obtained by applying the four-valued logic to the elements that composed the test requirement.

The objective of the generation of test cases is to maximize the trade-off among different factors such as cost, benefit or risks by means of obtaining a reasonable number of test cases that achieve detecting as many defects as possible. In this article, this process is based on the four-valued logic to evaluate SLAs and the proposed criterion to identify the test requirements. Once the set of test requirements that represent the situations to be tested is identified, we have to decide how to combine such test requirements in order to achieve an expected degree of coverage in the resulting test suite. To do this, it is also necessary to have some knowledge about the behaviour

of the SBA in order to properly combine the test requirements and obtain an effective set of test cases.

At this stage, it is the tester who has to combine the test requirements in each test case, bearing in mind that the more test requirements that can be covered in a single test case, the fewer test cases are needed to cover all the test requirements. In each test requirement we have assured that there are not non-feasible situations to test by applying the proposed specific rules but this does not mean that any combination of test requirements in a test case makes sense. Typically, there will be test requirements that are incompatible to be combined within the same test case due to the specification of the SBA so this task of generating the test cases by combining the test requirement is not definitely trivial and must be carefully performed.

4.5 Tool Support

We have implemented a proof-of-concept tool that automates the identification of test requirements from the specification of a SLA in the WS-Agreement language.

This tool receives the xml file of the SLA as input, it parses the content of the agreement and implements the algorithms for each compositor contained in the SLA. Once the initial set of test requirements is obtained, it accordingly modifies or removes the non-feasible combinations by means of applying the aforementioned rules.

The output of the tool is the specification of the final set of feasible test requirements, including the evaluation value of each involved Guarantee Term.

5 Case Study

In this section we illustrate the identification of test requirements from the logical conditions of a SLA associated to an eHealth service-based application. This scenario has been proposed in the context of the EU FP6 PLASTIC Project [9] and has also been used as case study in previous approaches that tackle the testing of SLAs [10], [11]. The SLA that contains the conditions that must be fulfilled by the stakeholders in this scenario is specified in WS-Agreement standard language and can be publicly downloaded [37]. The whole process has been performed automatically using the tool we have implemented.

5.1 Description

The behaviour of the service-based application that is used as case study in this article is as follows. Basically, the eHealth system is deployed as a composite service (WSHealth), which manages the alarms received from the patients. This service finds the list of professionals to solve the alarm in a registry (WSRegistry). There are two different types of alarms (Emergencies and Not Confirmation) and two types of professionals to handle the incident (doctors and supervisors), which are connected to the system through wired or mobile devices. If a doctor is contacted, he gets measures from the medical devices (WSMedicalDevice) deployed in the patient's location. If it is a supervisor who is contacted, he arranges an appointment for the patient in the calendar (WSCalendar).
This scenario has a SLA associated, specified in the WS-Agreement standard language. This SLA contains 14 Guarantee Terms, which are logically grouped using 5 compositors under the most external and mandatory All compositor. In Table 4 we represent the distribution of the guarantee terms in each of these compositors.

5.2 Identification of Test Requirements

The algorithms described in Section 4.2 have been applied in order to obtain the initial set of test requirements. As we have previously stated, many of these test requirements may be non-feasible so the rules defined in Section 4.3 have also been applied. As a result, Table 5 displays the compositors specified in the SLA (first column), the number of test requirements initially identified using the aforementioned algorithms (second column) and the number of test requirements that have been modified (M) and removed (R) after applying each rule (middle columns). Lastly, the last column outlines the final number of test requirements for each compositor.

Initially, a set of 62 test requirements are identified by applying the SLACDC criterion .These test requirements fulfil the conditions specified in such criterion, which assures that every Guarantee Term and every Compositor take the four potential evaluation values and the variation of any value affects the output of the evaluation. After that, we apply the rules we have defined in Section 4.3in order to avoid the obtaining of non-feasible requirements.

The final set contains a total number of 33 test requirements, which are represented in Fig. 8. This number is significantly lower than the number of test requirements we had obtained if we had applied a complete combination using the four-valued logic in each compositor. In that case, we had initially obtained a set of 1136 test requirements (4^n for each compositor, where n is the number of involved GTs).

5.3 Derivation of Test Cases

The identified test requirements are the basis to derive the test cases that will be executed in the SBA. In this section we provide an example about how different test requirements may be combined in order to generate a complex

TABLE 4
Structure of the eHealth SLA

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

TABLE 5
Test Requirements Identification

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

test case, which covers one test requirement from each of the SLA compositors (all of them are remarked in Fig. 8).

The specification of this test case is represented in Table 6. In the first column the exercised test requirement is represented. In the second column the description of the situations related to such test requirement is provided.

In this test case, the use of the two additional evaluation values (Inapplicable and Not Determined) contributes to identify a specific scenario in which no supervisors are invoked to manage the incidence (GT10 and GT11 = Not Determined) and only one medical device is queried whereas the other is not (GT14 = Inapplicable).

5.4 Discussion

One of the main benefits of this work is that a reasonable and manageable number of test requirements and test cases are systematically and automatically obtained by applying SLACDC. To be more specific, in this case study only 10 test cases are needed to cover all the TRs.

In addition to this, is worth mentioning that some design decisions have been taken when generating the test cases. Once the test requirements are obtained from each compositor, in this case study we have applied *each-choice testing* to combine one test requirement from each compositor to derive a test case. We have used this testing technique with the aim at obtaining a reduced number of test cases. However, we could address the derivation of test cases by means of applying other more exhaustive testing techniques such as *Pairwise* or *All Combinations*.

On the other hand, all the test requirements have been automatically identified with the aim at fulfilling the conditions specified in Definition 3. Due to this, the identified test requirements represent interesting combinations of situations that a tester would identify when performing a manual test in the eHealth system, including the arrival of different types of alarm, the invocation of both doctors and supervisors, the use of different types of medical devices and so on.

TABLE 6
TEST CASE SPECIFICATION

TR	Description
TR5	An emergency arrives to the system. The eHealth system must provide a response to the patient in less than the specified threshold time (GT1 is Fulfilled).
TR7	The registry is invoked in order to provide the list of available professionals to manage the incidence. In such situation, the registry spends more time than the expected to give the response (GT3 is Violated and GT4 is Inapplicable). Despite of this, the fulfillment of GT5 and GT6 means that the provided list contains a group of doctors that are ready to solve the emergency
TR20	Once this list is received, a doctor connected to the system through a mobile device is contacted, who accepts the incidence in less than the required time due to the fulfillment of GT8.
TR22	Consequently, no supervisors are contacted (GT10 and GT11 are Not Determined)
TR29	The doctor successfully obtains the list of medical device deployed in the patient's home (GT12 is Fulfilled) and enquiries to receive the measure from the first medical device, which spends too much response time (GT13 is Violated and GT14 is Inapplicable).

All (1)	GT1	GT2
TR1	V	I
TR2	I	F
TR3	N	N
TR4	I	V
TR5	F	I
TR6	I	I

All (2)	GT3	GT4	GT5	GT6	GT7
TR7	V	I	F	F	I
TR8	I	F	F	I	F
TR9	N	N	N	N	N
TR10	I	V	F	I	F
TR11	F	I	F	F	I
TR12	F	I	V	F	I
TR13	F	I	F	V	I
TR14	I	F	F	I	V

ExOne (1)	GT8	GT9
TR15	I	V
TR16	N	N
TR17	V	I
TR18	I	I
TR19	I	F
TR20	F	I

ExOne (2)	GT10	GT11
TR21	I	V
TR22	N	N
TR23	V	I
TR24	I	I
TR25	I	F
TR26	F	I

All (3)	GT12	GT13	GT14
TR27	V	F	I
TR28	N	F	I
TR29	F	V	I
TR30	F	I	F
TR31	F	N	N
TR32	F	I	V
TR33	F	F	I

Fig. 8. Final Set of Test Requirements.

6 RELATED WORK

During recent years, many works have been proposed with the final objective of detecting SLA violations. Most of these works can be classified according to two main dimensions: (1) proactive approaches that aim at preventing or anticipating the detection of problems in the SBA and (2) reactive approaches that detect the problems at runtime by observing the behaviour of the SBA. Below we briefly describe the main characteristics of each work and we state the points in common with our approach.

Regarding the first group, few works have addressed the early identification of tests from the specification of SLAs. Di Penta et al. [19] propose a black and white-box approach to detect SLA violations in atomic and web service compositions by means of using Genetic Algorithms. Their objective is the generation of inputs that causes violations of the SLA whereas we focus on the identification of situations that implies evaluating the SLA with all the potential evaluation values, not only violations. Palacios et al. [38] use the Category Partition Method (CPM) testing technique in order to identify tests from WS-Agreements. This work states the problem of the exponential growth of tests when the SLA becomes complex, which is an issue that we address in our work by means of applying coverage-based testing. In previous works, we provide a general framework to test SLAs [20] and we focus on identifying test requirements from the individual guarantee terms of WS-Agreements [7]. In this article we extend the test basis by considering not only the guarantee terms in isolation but the logical composition of such terms. Bertolino et al. [11] propose PUPPET framework, which generates test beds from the WSDL and BPEL specification of service compositions, considering the information contained in a WS-Agreement. This work can be complemented with our work in the sense that they provide the necessary infrastructure to deploy and execute the tests we identify in this article. Muller et al. [39] propose static testing by detecting and explaining inconsistencies between the terms of WS-Agreements using a Constraint Satisfaction Problem based approach.

Regarding the second group, there are more works

that use monitoring techniques rather than testing to detect SLA violations. Raimondi et al [3] propose an automatic SLA monitoring system that verifies the traces of services executions by translating timeliness constraints into timed automata. Mahbub and Spanoudakis [1] present an Event Calculus (EC) based approach to model and monitor the conditions specified in a WS-Agreement. Comuzzi et al. [40] address both the establishment and monitoring of SLA in the context of the SLA@SOI European Project [41]. Beyond these works, other systems such as SALMonADA [2] or SLAMonitor [42], have been proposed to monitor the behavior of the SBA and detect the SLA violations. These approaches may be complemented with our work by means of configuring the monitoring systems in order to observe whether the test requirements have been exercised or not at runtime.

In the borderline between these two groups, there are other works that use monitoring techniques not to detect but prevent SLA violations. Lorenzoli and Spanoudakis [43] present EVEREST+ framework, which allows monitoring and predicting potential violations of the QoS metrics described in the SLA. Leitner et al. [44] propose another framework to predict SLA violations by using machine learning techniques. Finally, Ivanovic et al. [45] detect SLA violations by means of monitoring and analyzing the QoS metrics using a constraint-based approach.

7 Conclusions and Future Work

In this article we have addressed the evaluation and testing of the logical composition of guarantee terms in a Service Level Agreement (SLA). We have defined a four-valued logic that allows evaluating both individual guarantee terms and compositor elements. This logic is the basis for the SLACDC (SLA Condition / Decision Coverage) criterion we have devised in order to identify a set of test requirements that combine different evaluation values of the terms involved in a compositor. This criterion is based on MCDC criterion and it provides a linear growth of test requirements regarding the number of guarantee terms included in the compositor.

In addition to this identification of test requirements, we have to deal with non-feasible situations due to the semantics of the SLA terms. To address this issue, we have define a set of rules, which are automatically applied, that allow removing the non-feasible test requirements or, if possible, the modification of such requirements in order to obtain feasible situations.

The whole approach has been automated and validated over an eHealth case study proposed in the context of a European FP7 Project. The automation of the approach allows reducing the tester's effort required to design and specify aligned with the SLA specification. Furthermore, the analysis and exercitation of the test requirements also allow detecting wrong SLA specifications regarding the relationships between the guarantee terms.

In future work, we will focus on evaluating the potential use of the aforementioned test requirements in other testing domains. For example, these requirements may contribute to derive monitoring plans that provide guidelines about which situations are more interesting to observe at runtime, when the SBA is deployed and executed in the operational environment. Furthermore, we expect to address the design of tests when the specification of the SLA contains different levels of nesting between the guarantee terms and the compositors.

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) and FICYT (Government of the Principality of Asturias) Grant BP09-075.

References

[1] K. Mahbub and G. Spanoudakis, "Monitoring WS-Agreements: an Event Calculus Based Approach," *Test and Analysis of Service Oriented Systems*, Springer V., 2007, pp. 265-306.

[2] C. Muller, M. Oriol, M. Rodriguez, X. Franch, J. Marco, M. Resinas, and A. Ruiz-Cortés, "SALMonADA: A Platform for Monitoring and Explaining Violations of WS-agreement-Compliant Documents" *ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS)*, pp. 43-49, 2012.

[3] F. Raimondi, J. Skene, and W. Emmerich, "Efficient Online Monitoring of Web-Service SLAs," *Proc. 16th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE-16), 2008.*

[4] L. Baresi, N. Georgantas, K. Hamann, V. Issarny, W. Lamersdorf, A. Metzger, and B. Pernici, "Emerging Research Themes in Services-Oriented Systems," *SRII Global Conference (SRII), 2012 Annual* , vol., no., pp.333-342, 24-27 July 2012.

[5] M. Palacios, J. García-Fanjul, and J. Tuya, "Testing Service Oriented Architectures with Dynamic Binding: a Mapping Study," *Information and Software Technology*, vol. 53 (3), pp. 171-189, 2011.

[6] G. Canfora and M. Di Penta, "Testing Services and Service-Centric Systems: Challenges and Opportunities," *IT Professional* 8 (2), pp. 9–17, 2006.

[7] M. Palacios, J. García-Fanjul, and J. Tuya, "Identifying Test Requirements by Analyzing SLA Guarantee Terms," *Proc. 19th Int. Conf. on Web Services*, pp. 351-358, 2012.

[8] RCTA Inc. DO-178-B: Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA), 1992.

[9] PLASTIC European Project. http://www.ist-plastic.org

[10] M. Autili, P.D. Benedetto, and P. Inverardi, "Context-Aware Adaptive Services: The Plastic Approach," *Proc. 12th Int. Conf. In Fundamental Approaches to Software Engineering (FASE)*, York, UK, March 22-29, 2009. Proc. LNCS, vol. 5503, pp. 124–139.

[11] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini, "Model-Based Generation of Testbeds for Web Services," *Proc. Testing of TESTCOM/FATES*, LNCS, vol. 5047, 2008, pp. 266-282.

[12] Google Apps SLA: http://www.google.com/apps/intl/en/terms/sla.html

[13] Amazon EC2 SLA: http://aws.amazon.com/ec2-sla/

[14] E. Di Nitto, M. Di Penta, A. Gambi, G. Ripa, and M.L. Villani, "Negotiation of Service Level Agreements: an Architecture and a Search-Based Approach," *Proc. 5th Int. Conf. Service-Oriented Computing (ICSOC)*, September 17-20, pp. 295–306, 2007.

[15] F.H. Zulkernine and P. Martin, "An Adaptive and Intelligent SLA Negotiation System for Web Services," *IEEE Trans. Services Computing*, vol. 4, no. 1, pp. 31-43, Jan.-March 2011.

[16] M. Palacios, L. Moreno, M.J. Escalona, and M. Ruiz, "Evaluating the Service Level Agreements of NDT under WS-Agreement. An Empirical Analysis," *Proc. 8th Int. Conf. on Web Information Systems and Technologies*, Porto, Portugal, April 2012.

[17] D.M. Quan and L.T. Yang, "Parallel Mapping with Time Optimization for SLA-Aware Compositional Services in the Business Grid," *IEEE Trans. Services Computing*, vol. 4, no. 3, pp. 196-206, July-Sept. 2011. doi: 10.1109/TSC.2011.27
[18] H. Wada, J. Suzuki, Y. Yamano, and K. Oba, "E3: A Multiobjective Optimization Framework for SLA-Aware Service Composition," *IEEE Trans. Services Computing*, vol. 5, no. 3, pp. 358-372, Third Quarter 2012. doi: 10.1109/TSC.2011.6
[19] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno, "Search-Based Testing of Service Level Agreements," *Proc. Annual Conference on Genetic and Evolutionary Computation (GECCO 07)*, London, ACM, New York, 2007, pp. 1090-1097.
[20] M. Palacios, "Defining an SLA-aware Method to Test Service-Oriented Systems," *Proc. 9th Int. Conf. on Service Oriented Computing (ICSOC)*, PhD Symposium, G. Pallis et al. (Eds.): ICSOC 2011, LNCS 7221, pp. 164--170. Springer, Heidelberg 2012.
[21] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web services agreement specification (WS-Agreement)," 2010.
[22] ISO/IEC 24765, Software and Systems Eng. Vocabulary, 2006.
[23] IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology.
[24] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A Journey to Highly Dynamic, Self-Adaptive Service-Based Applications," *Automated Soft. Eng.*, 15, pp. 313–341, 2008.
[25] J. Offut, L. Nan, P. Ammann, and X. Wuzhi, "Using Abstraction and Web Applications to Teach Criteria-Based Test Design," *Proc. 24th IEEECS Conference on Software Engineering Education and Training (CSEE&T)*, 2011, pp.227-236.
[26] Q. Wang; J. Shao; F. Deng; Y. Liu; M. Li, J. Han, and M. Hong, "An Online Monitoring Approach for Web Service Requirements,", *IEEE Trans. Services Computing*, vol. 2, no.4, pp. 338-351, Oct.-Dec. 2009. doi: 10.1109/TSC.2009.22
[27] N.D. Belnap, "A Useful Four-valued Logic," In: J.M. Dunn, G. Epstein (eds.), Modern Uses of Multiple-Valued Logic, Dordrecht: Reidel, pp. 8—37, 1977.
[28] E.F. Codd, "The Relational Model for Database Management," - Version 2. Addison-Wesley, Reading, MA, 1990.
[29] G. Gessert, "Four Valued Logic for Relational Database Systems," Sigmod Rec. 19 (1), pp. 29—35, 1990.
[30] A. Dupuy and N. Leveson, "An Empirical Evaluation of the MCDC Coverage Criterion on the HETE 2 Satellite Software," *Proc. 19th Digital Avionics System Conference (DASC)*, 2000.
[31] M.R. Woodward and M.A. Hennell, "On the Relationship Between Two Control-Flow Coverage Criteria: all JJpaths and MCDC," *Information and Software Technology*, vol. 48 (7), pp. 433-440, 2006.
[32] J.J. Chilenski and S.P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol 9 (5), pp. 193-229, 1994.
[33] J.A. Jones and M.J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Transactions Software Engineering*, vol 29 (3), pp. 195-209, 2003.
[34] J. Kapoor and J.P.Bowen, "Experimental Evaluation of the Tolerance for Control-Flow Test Criteria," *Software Testing, Verification and Reliability*, vol. 14 (3), pp. 167-187, 2004.
[35] T.K. Yu and M.F. Lau, "A Comparison of MC/DC, MUMCUT and Several Other Coverage Criteria for Logical Decisions," *Journal of Systems and Software*, vol. 79 (5), pp. 577-590, 2005.
[36] J.J. Chilenski, "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion," Technical Report DOT/FAA/AR-01/18, U.S. Department of Transportation, Federal Aviation Administration, April 2001.
[37] Software Engineering Research Group (GIIS) downloads: http://giis.uniovi.es/testing/downloads/?lang=en.
[38] M. Palacios, J. García-Fanjul, J. Tuya, and C. de la Riva. "A Proactive Approach to Test Service Level Agreements," Proc *5th Int. Conf. Software Eng. Advances (ICSEA)*, pp. 453-458, 2010.
[39] C. Muller, M. Resinas, and A. Ruiz-Cortes, "Automated Analysis of Conflicts in WS-Agreement," *IEEE Trans. Services Computing*, 25 Feb. 2013. IEEE computer Society Digital Library.
[40] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour, "Establishing and Monitoring SLAs in Complex Service Based Systems," *Int. Conf. on Web Services (ICWS)*, pp. 783-790, 2009.
[41] SLA@SOI European Project
[42] N. Goel, .V.N. Kumar, R.K. Shyamasundar, "SLA Monitor: A System for Dynamic Monitoring of Adaptive Web Services," *Proc 9th IEEE European Conf. on Web Services*, pp. 109-116, 2011.
[43] D. Lorenzoli and G. Spanoudakis, "EVEREST+: Run-Time SLA Violations Prediction," *Proc. of the 5th International Workshop on Middleware for Service Oriented Computing (MW4SOC '10)*, pp. 13-18. ACM, New York, NY, USA, 2010.
[44] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "Monitoring, Prediction and Prevention of SLA Violations in Composite Services," *IEEE Int. Conf. on Web Services (ICWS)*, pp. 369-376, 2010.
[45] D. Ivanovic, M. Carro, and M. Hermenegildo, "Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations," *Proc. of the International Conference on Service Oriented Computing (ICSOC)*, 2011, pp. 62-76.

Marcos Palacios is currently PhD student and Teaching Assistant at University of Oviedo, Spain, and a member of the Software Engineering Research Group of that University. He received his B. Sc. degree in Computer Science in 2006 and his M. Sc. in Computer Science in 2008 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 in software engineering include verification & validation and software testing for database applications and services. He is a member of the IEEE, IEEE Computer Society, ACM and the Association for Software Testing (AST).

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. He is a member of the IEEE.