SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **44**:201–234
Published online 18 October 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2161

A holistic approach to model-based testing of Web service compositions

Fevzi Belli[1], Andre Takeshi Endo[2,*,†], Michael Linschulte[1] and Adenilso Simao[2,‡]

[1]*Electrical Engineering and Mathematics, University of Paderborn, Germany*
[2]*Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (USP), Brazil*

SUMMARY

The behavior of composed Web services depends on the results of the invoked services; unexpected behavior of one of the invoked services can threat the correct execution of an entire composition. This paper proposes an event-based approach to black-box testing of Web service compositions based on event sequence graphs, which are extended by facilities to deal not only with service behavior under regular circumstances (i.e., where cooperating services are working as expected) but also with their behavior in undesirable situations (i.e., where cooperating services are *not* working as expected). Furthermore, the approach can be used independently of artifacts (e.g., Business Process Execution Language) or type of composition (orchestration/choreography). A large case study, based on a commercial Web application, demonstrates the feasibility of the approach and analyzes its characteristics. Test generation and execution are supported by dedicated tools. Especially, the use of an enterprise service bus for test execution is noteworthy and differs from other approaches. The results of the case study encourage to suggest that the new approach has the power to detect faults systematically, performing properly even with complex and large compositions. Copyright © 2012 John Wiley & Sons, Ltd.

Received 5 March 2012; Revised 24 September 2012; Accepted 26 September 2012

KEY WORDS: enterprise service bus; event sequence graphs; model-based testing; service composition testing; test case generation

1. INTRODUCTION

Enterprise applications have become more and more complex as they have to cope with strict requirements, such as of business processes and their dynamic evolution, and interaction among different companies. Accordingly, the architecture for integrated enterprise applications has to provide interoperability, scalability, and rapid development. The adoption of technologies that foster the interoperability between different applications has been a recurring solution. *Service-oriented architectures* (*SOAs*) and Web services have been used to enable loosely-coupled, distributed applications by using independent and self-contained services. These services can be combined in a workflow that characterizes a new, *composite* service. The resulting composite service is also called as *Web service composition* (*WSC*). Apart from the adoption of SOA and Web services, the use of a service bus to ease the integration process has been advocated for service-oriented applications [1–3]. The so-called *enterprise service bus* (*ESB*) controls, routes, and translates messages exchanged by the services involved [3].

*Correspondence to: Andre Takeshi Endo, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (USP), Brazil.
†E-mail: aendo@icmc.usp.br
‡The authors are alphabetically ordered.

To assure the delivery of high quality and robust service-oriented applications, SOA testing has received much attention [4]. In this context, WSC testing plays an important role [5–11] because the behavior of the composite services now depends not only on the WSC itself but also on the integrated services, complicating the testing process. The WSC can present complex communications among the integrated services in which missing or unexpected messages can lead to a failure. Furthermore, the composition may fail because of undesirable behavior of partner services, such as corrupted messages, unavailable servers, and long timeouts.

A common problem in testing any kind of application is to automatically generate meaningful test cases. The strategy of using models for test case generation is known as *model-based testing* (*MBT*). In MBT, a tester uses his or her knowledge of a given *system under consideration* (*SUC*)[§] to develop a model for generating test cases. MBT can be applied in initial development phases because the modeling and test generation do not require an executable system. The appropriate application of MBT in software projects brings several benefits, such as high fault detection rate, reduced cost and time for testing, requirement evolution, and high level of automation (see, e.g., a detailed evaluation of MBT in [12]).

Test models are more productive when specific features of the system can be described using an appropriate modeling technique. Event-based models have been used to support verification and testing [13–15] because events are essential for many different classes of systems, for example, Web applications or embedded systems. *Event sequence graphs* (*ESGs*), originally introduced for testing graphical user interfaces [14], were also used to model and test stateful Web services [16]. An advantage of ESGs with respect to other general purpose modeling languages is that ESGs are holistic; that is, they allow testing not only the desirable behavior (known as *positive testing*) but also the undesirable behavior (known as *negative testing*). Furthermore, they intensively use formal notions and algorithms known from graph theory and automata theory, which are also relevant to the approach introduced in this paper.

In this paper, a new approach, called *ESG for Web service compositions* (*ESG4WSC*) is introduced to generate cost-effective test cases for WSCs. An event-oriented approach is proposed for several reasons. First, message exchanges in a WSC can be viewed as events that follow an order. Second, ESG modeling can be learned in a short period [17], requires little manual work, and is supported by specific tools [14]. Furthermore, artifacts (e.g., standardized service descriptions) need not be available to create an ESG or any other model for a service composition. That is, an ESG can be constructed in an ad hoc way by the tester wherever no model is available. It is assumed that the tester can observe and modify the exchanged messages using an ESB (present in several SOAs); that is, the service composition is considered as a black box, but the tester has control over messages exchanged by the partner services.

The novelties and merits of this paper are summarized as follows:

- An event-based approach is proposed to support WSC testing by
  - extending the basic notions of ESG [14, 18], referred to as ESG4WSC, for testing the WSC behavior under regular circumstances (positive testing) and undesirable situations (negative testing) based on one model. The concept of 'sensitive' events is also introduced as test oracle for negative test cases;
  - introducing new scalable algorithms to generate positive and negative test cases from ESG4WSC;
  - enabling to model independently of artifacts like Business Process Execution Language (BPEL); that is, modeling in parallel to implementation is possible;
  - enabling to model independently of the type of composition, that is, orchestration or choreography.

[§]Notice that SUC is used in this paper instead of 'system under test (SUT)', which supposes that the system to be tested is already available. However, in the initial development phases, the system itself is not yet available but only a description and, in favorable cases, a model of the system. Using MBT techniques, test cases can be generated for this SUC but cannot be executed. Therefore, SUT will not be the appropriate term to call a system that can be analyzed by means of its model, but not executed and thus not yet tested. In a later stage, when the system has already been implemented and thus is available, it can be called SUT. Nevertheless, SUT is also an SUC.

DOI: 10.1002/spe

- A case study demonstrates the applicability of the approach and evaluates it by means of a practical and nontrivial Web application that is commercially available.
- Two tools are introduced to support automation:
  - *Test Suite Designer* (*TSD*) provides a graphical user interface, which allows to model the SUC and to generate test cases;
  - *Event Runner for Test Execution* (*ERunTE*) automates test execution by composing three modules: a Web service, a test runner, and an ESB component.

To our knowledge, there is no comparable work that performs testing of WSC by modeling and testing not only the published interface, which is available to the consumer, but also the internal communication with other services, which is usually hidden from the consumer. Furthermore, the ESG4WSC approach is *holistic* because it considers positive and negative testing at the same time [18].

Preliminary results of this work have been published in the paper [19], which is extended by (i) evolving the approach to a holistic view that includes the testing of unexpected behavior in the WSC (so-called negative testing), (ii) experimenting the approach in a case study engaging a large, complex commercial Web portal, and (iii) developing mechanisms to increase the automation level of the approach through dedicated, specific-purpose tools. In this paper, besides the positive testing previously described in [19], the negative testing and its algorithms for test case generation are proposed to complement the approach. The case study is conducted in a more complex and larger scenario to collect and analyze more information. The TSD tool is also improved to represent the ESG4WSC model; moreover, new algorithms for negative testing were implemented. Ideas that were presented before to support test execution were concretized in the ERunTE tool.

The remainder of this paper is organized as follows. Section 2 briefly presents concepts of SOA, Web services, and ESBs. Section 3 introduces the proposed holistic approach to test service compositions, along with an example. Section 4 presents a case study and discusses the results of experiments. Section 5 shows the tool support for the practical project work. Section 6 shows the related work. Finally, Section 7 concludes the paper and sketches the future work planned.

2. BACKGROUND

Information technology landscape of enterprises is mostly heterogeneous, complicating the integration of systems implemented by different technologies. SOA has been introduced to fill this gap and provide a *de facto* standard enabling communication among those systems. SOA is an emerging approach that aims to foster loose coupling among applications. It provides a standardized, distributed, and protocol-independent computing paradigm. Software resources are wrapped as 'services', which are well-defined and self-contained modules providing business functionality and being independent from other service states or contexts [3]. Usually, implementation details of a service are hidden, and only its interface is available; that is, a service can be viewed as a black box.

A SOA is based on three entities: provider, consumer, and registry. Using the Web services technology, practitioners realize a SOA through three main XML standards: Web Service Description Language (WSDL), Universal Description, Discovery and Integration (UDDI), and SOAP. WSDL is a W3C standard used to describe the service interface, including details like operations, data types, and adopted protocols. UDDI is an OASIS standard that defines a set of functionalities to support description and discovery of services. SOAP (originally for 'Simple Object Access Protocol' but not an acronym anymore [20]) is also a W3C protocol used to define the structure of messages exchanged among the services. When practitioners use SOAP, it is possible to define error messages by using *SOAP-Faults*. SOAP Faults are expected by the consumer if they are specified in the WSDL interface and used to map exceptions that happen within the service. However, the Web service frameworks also tend to launch SOAP Faults when internal exceptions are not handled correctly. In this case, it is said that the SOAP Fault is unexpected.

A group of services can be assembled to create a new value-added service via composition. In a service composition, many services can be combined in a workflow to model and execute complex business processes. The services involved in a service composition are usually called

DOI: 10.1002/spe

partner services. Service compositions can be developed either as *orchestration* or as *choreography*. In service orchestration, there is a main entity that is responsible for coordinating the partner services. Currently, the most widespread language to implement a service orchestration is BPEL [21]. In service choreography, there is no control entity, and all partner services work cooperatively to achieve an agreed objective. There are several languages used to describe service choreography, for example, Web Services Choreography Description Language (WS-CDL) [22] and Web Service Choreography Interface (WSCI) [23]. The service composition is also a service (referred to as a *composite service*) and can be reused by other services. A service that is not a composition is usually called *atomic* service. The terms *service* and *Web service* are used as synonyms in this paper.

An ESB, which is an intermediate layer among the services, can also be included in a SOA. The ESB works as a backbone that uses Web services technology to support many communication patterns over different transport protocols and provides interesting capabilities for service-oriented applications, such as routing, provisioning, service management, integrity, and security [3]. The adoption of ESBs has been considered essential for companies to reach the full advantages provided by SOAs [2]. An ESB enables high interoperability and eases the distribution of business processes using different platforms and technologies [2]. Figure 1 presents the transition from a traditional SOA to an ESB-based architecture. According to Schmidt *et al.* [1], the ESB is an infrastructure that fully supports an integrated and flexible SOA. These features are reached by receiving, operating, or mediating on the service messages, as they flow through the bus. There are many possible uses for mediation, such as load balance, monitoring, and validation. Schmidt *et al.* [1] describe a set of mediation patterns that are useful in an ESB. In the context of this work, two patterns were used:

- *Monitor pattern* provides the feature for observing messages that pass through the ESB, not applying any type of change in the messages. This pattern can be applied to logging, audition, monitoring of service levels, measurement of client usages, and so on.
- *Aggregator pattern* provides the feature for monitoring messages from different services over a period of time and generating new messages or events. This pattern can be useful for realizing complex scenarios so that, for example, a set of events can be mapped to a single event.

The ESB can be provided by software that implements the concepts of those functionalities. Several ESB applications are available, from proprietary vendors to open-source solutions.[¶] In this work, the open-source version of Mule-ESB [24] is adopted, which is a lightweight Java-based ESB that includes much functionality to integrate existing systems.

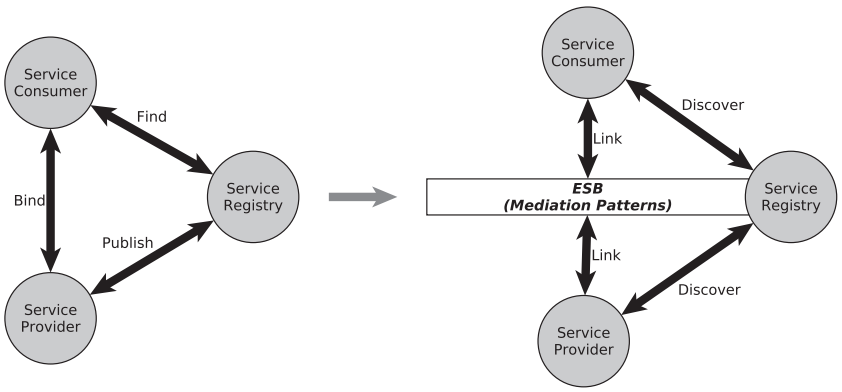

Figure 1. Enterprise service bus representation in a service-oriented architecture (adapted from [1]).

3. EVENT-BASED TESTING OF WEB SERVICE COMPOSITIONS

This section presents the holistic approach to testing service compositions using ESGs. Moreover, processes and algorithms are introduced to generate positive and negative test cases.

[¶]http://en.wikipedia.org/wiki/Enterprise_service_bus

DOI: 10.1002/spe

3.1. Running example and ESG4WSC model

To facilitate the description and understanding of the approach, a 'running example' is introduced first. After that, the ESG4WSC model is described in details.

3.1.1. Running example. The business process to grant loans called `xLoan`, proposed in [8], is the running example used to illustrate the approach. Note that this example is not the case study described in Section 4, which is a nontrivial, commercial Web application.

The example involves three services: `LoanService` (`LS`), `BankService` (`BS`), and `BlackListInformationService` (`BLIS`). `LoanService` represents the own business process `xLoan` whose workflow is implemented using BPEL. It contains three operations: request, cancel, and select. `BankService` represents the financial agency that approves (or not) loans, providing loan offers to its clients. The operations used in the example are approve, offer, confirm, and cancel. `BlackListInformationService` provides an operation checkBL to check if a client has debits with some financial organization.

The example is extended to add parallel flow (a common entity of WSCs) in the process by including a new service called `CommercialAssociationService` (`CAS`). Similar to `BlackListInformationService`, `CAS` provides operations to check whether a client has debits with some commercial organization. In the extension, both services are supposed to be called in parallel. If the client has debit according to one of them, the client needs the bank approval.

3.1.2. ESG4WSC model. This section introduces an event-based model, named ESG4WSC, that represents the request and response messages exchanged between services involved in a WSC. When a given event is refined by input parameters that determine the next events, decision tables (DTs) are associated to augment the representation. DTs are widely employed in information processing and are also traditionally used for testing, for example, in cause and effect graphs [25]. A DT logically links constraints (if) with events (then) that are to be triggered, depending on combinations of constraints (rules). DTs are powerful mechanisms for

- handling sequences of events that depend on constraints and
- refining data modeling of calls to invoked services [16].

Decision tables are formally defined as follows.

Definition 1
A (simple/binary) *decision table* $DT = \{C, E, R\}$ represents events that depend on certain constraints, where

- C is the nonempty finite set of constraints (conditions), which can be evaluated as either true or false,
- E is the nonempty finite set of events, and
- R is the nonempty finite set of rules each of which forms a Boolean expression connecting the truth/false configurations of constraints and determines the executable or awaited event.

Definition 2
Let R be a set of rules as in Definition 1. Then, a *rule* $R_i \in R$ is defined as $R_i = (C_{\text{True}}, C_{\text{False}}, E_x)$, where

- $C_{\text{True}}, C_{\text{False}} \subseteq C$ are the disjoint sets of constraints that have to be evaluated as **true** and **false**, respectively;
- $E_x \subseteq E$ is the set of events that should be executable if all constraints $t \in C_{\text{True}}$ are resolved to true and all constraints $f \in C_{\text{False}}$ are resolved to false. In this work, $|E_x| = 1$ for all rules to avoid nondeterminism.

Note that under regular circumstances, C_{True} and C_{False} partition C, that is, $C_{\text{True}} \cup C_{\text{False}} = C$ and $C_{\text{True}} \cap C_{\text{False}} = \emptyset$. In certain cases, it is inevitable to have constraints with a *don't care*

DOI: 10.1002/spe

Table I. A decision table for operation checkBL.

		Rules		
	checkBL(uniqueID)	R1	R2	R3
Constraints	**uniqueID is valid**	T	T	F
	uniqueID in Blacklist	T	F	–
	inBList < 200 ms	✓		
Events	notinBList < 100 ms		✓	
	SOAPFault—invalid identification			✓

(noted as '–' in a DT). In this case, such a constraint is not considered in a rule and is neither in C_{True} nor in C_{False}. Here, DTs are used to refine input parameter of invoked services.

Table I presents a DT that models the invoking process of operation checkBL of BLIS. It contains two constraints on input parameter uniqueID, three successor events (inBList, notInBList, and SOAPFault), and three rules (R1, R2, and R3). Constraints in bold model possible domains of input parameters and constraints in italics represent additional constraints to that input data. Rules are used to determine the allowed successor event of operation checkBL. For instance, R1 means that if uniqueID is valid and also in the blacklist (i.e., both constraints are true), then the next event (viz, event inBList standing for uniqueID being in the blacklist) should be completed in less than 200 ms. In R3, if uniqueID is not valid, then the other constraint does not matter (namely '–') and the next event is SOAPFault. As illustrated in Table I, the events can be extended by time constraints whenever a response is expected within a certain time range.

Decision tables are useful to describe constraints, but they are not appropriate for describing WSC interactions. Hence, the ESG notion is extended and combined with DTs to consider additional aspects, such as communication, parallel flow, and conditional activities.

Definition 3
An *ESG for Web service compositions* $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$ is a directed graph, where

- V is a nonempty finite set of vertices (representing events);
- $E \subseteq V \times V$ is a finite set of arcs (edges);
- M is a finite set of refining ESG4WSC models;
- $R \subseteq V \times M$ is a relation that specifies which ESG4WSCs are connected to a refined vertex;
- DT is a set of DTs that refine events according to function f;
- $f : V \to DT \cup \{\varepsilon\}$ is a function that maps a decision table $dt \in DT$ to a vertex $v \in V$. If $v \in V$ is not associated with a DT, then $f(v) = \varepsilon$;
- $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$ and $\gamma \in \Gamma$ called entry nodes and exit nodes, respectively, wherein for each $v \in V$ there exists at least one sequence of vertices $\langle \xi, v_0, \ldots, v_k \rangle$ from $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

Definitions 4 and 5 elaborate Definition 3, formalizing the set of vertices and the set of DTs, respectively.

Definition 4
Let V be as in Definition 3. Then, the set of vertices V is partitioned into V_e, V_{refined}, V_{req}, and V_{resp}, that is, $V = V_e \cup V_{\text{refined}} \cup V_{\text{req}} \cup V_{\text{resp}}$ and V_e, V_{refined}, V_{req}, and V_{resp} are pairwise disjoint, where

- V_e is a set of generic events,
- $V_{\text{refined}} = \{v \in V \mid \exists m \in M \wedge (v, m) \in R\}$ is a set of vertices refined by one or more ESG4WSCs. A refinement with more than one ESG4WSC represents behavior running in parallel,

DOI: 10.1002/spe

- V_{req} is a set of vertices modeling a request to its own interface/operations (public) or an invoked service (private), and
- V_{resp} is a set of responses to a public or private request. Therefore, it is also remarked as public or private.

Definition 5
Let DT be defined as in Definition 3. Then, the set of decision tables DT is partitioned into DT_{seq} and DT_{input}, where

- DT_{seq} is the set of DTs that model the execution restrictions for following events and
- DT_{input} is the set of DTs that model constraints for input parameter of invoked operations.

Because WSCs always initiate with one or more request events, the set Ξ contains only vertices $v \in V_{\text{req}}$. To mark the entry and exit of an ESG4WSC, all $\xi \in \Xi$ are preceded by a pseudo-vertex $[\notin V$, and all $\gamma \in \Gamma$ are followed by another pseudo-vertex $] \notin V$.

For two events $v, v' \in V$, the event v' can follow the execution of v if and only if $(v, v') \in E$. In this case, v' is also called *successor* of v, and v is called *predecessor* of v'. If vertex $v \in V$ has more than one successor, then v is to be refined by a DT.

The semantics of an ESG4WSC is as follows: Any $v \in V$ represents an event, for example, a request or a response that occurs during the invocation of another service. In general, requests and responses can be divided into *public* and *private*. A public request is controlled by the tester; that is, it is an operation call to the WSC itself, which is supposed to be performed by a consumer or the tester. A public response is expected to be an answer of the WSC to a public request and therefore should be observable by the consumer/tester. The opposite is true for private requests and responses that represent partner services of the WSC. They are usually not observable by a consumer; however, it is assumed that they are observable by the tester. Private requests are to be observed by the tester, and the tester should control and (if necessary) send back the appropriate response.

Example 1
Figure 2 represents an ESG4WSC for `xLoan`. Requests are represented by gray vertices in circle shapes; responses are represented by gray vertices in ellipse shapes. Vertices with a bold line represent public requests and responses. Vertices refined by DTs are double-circled. Event check (dashed box) is a refined event with two refining ESG4WSCs, which represent operations checkBL and inDebtorsList that are to be executed concurrently. The corresponding ESG4WSC using the defined sets and functions looks like this:

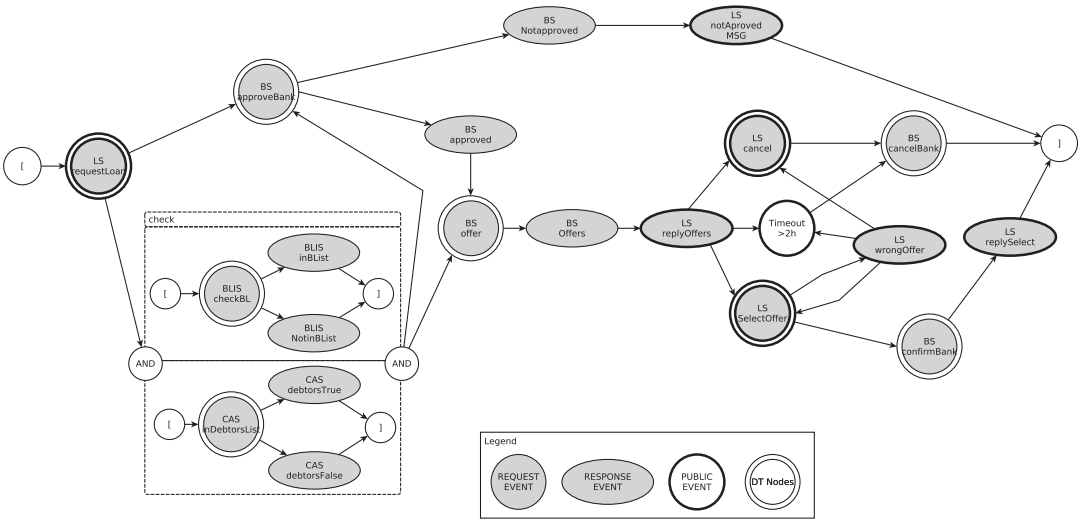

Figure 2. ESG4WSC for the `xLoan` example.

DOI: 10.1002/spe

$ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$ with

- $V_e = \{Timeout > 2h\}$,
- $V_{\text{refined}} = \{check\}$,
- $V_{\text{req}} = \{LS : requestLoan,\ BS : approveBank,\ BS : offer,\ LS : cancel,$
  $LS : SelectOffer,\ BS : cancelBank,\ BS : confirmBank\}$
- $V_{\text{resp}} = \{BS : approved,\ BS : Notapproved,\ LS : notAprovedMSG,\ BS : Offers,$
  $LS : replyOffers,\ LS : wrongOffer,\ LS : replySelect\}$
- $E = \{(LS : requestLoan,\ BS : approveBank), \ldots\}$,
- $M = \{M_{BLIS}, M_{CAS}\}$,
- $R = \{(check, M_{BLIS}), (check, M_{CAS})\}$,
- $DT = DT_{\text{seq}} \cup DT_{\text{input}} = \{dt_{check}\} \cup \{dt_{LS:requestLoan}, dt_{BS:approveBank}, \ldots\}$,
- $f(check) = dt_{check}$,
  $f(LS : requestLoan) = dt_{LS:requestLoan}$,
  $f(BS : approveBank) = dt_{BS:approveBank}, \ldots$
- $\Xi = \{LS : requestLoan\}$,
- $\Gamma = \{LS : notAprovedMSG,\ BS : cancelBank,\ LS : replySelect\}$

The ESG4WSC model can be seen as a simplified representation of possible and expected executions of a WSC. It aggregates sequences of events that describe walks of execution and message exchanges along the execution.

Definition 6
Let V and E be defined as in Definition 3. Then, any sequence of vertices $\langle v_0, \ldots, v_k \rangle$ is called an *event sequence* (*ES*) if $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$. The length l of an ES $\langle v_0, \ldots, v_k \rangle$ is defined as the number of vertices $|\langle v_0, \ldots, v_k \rangle|$; that is, $l(ES) = |\langle v_0, \ldots, v_k \rangle| = k + 1$. An $ES = \langle v_i, v_k \rangle$ of length 2 is called an *event pair* (*EP*). Furthermore, an ES is complete (or it is called a *complete event sequence, CES*), if $v_0 \in \Xi$ and $v_k \in \Gamma$. An ES is partial (or it is called a *partial event sequence, PES*), if $v_0 \in \Xi$.

Definition 7
Two events or ESs a and b that are to be executed in parallel are denoted as $a||b$. The operator $||$ is commutative, that is, $a||b = b||a$, and associative, that is, $(a||b)||c = a||(b||c)$.

Example 2
For the `xLoan` example given in Figure 2, services `CAS` and `BLIS` are to be executed in parallel; for example, following sequence might hold the following:

```
⟨BLIS:checkBL, BLIS:inBList⟩||
        ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩
```

With Definitions 6 and 7, sequences of events can be derived from the ESG4WSC and may represent parallel execution. A PES represents a sequence of events that starts with an entry node; a CES represents an event sequence that starts with an entry node and ends with an exit node.

3.2. Positive testing of Web service compositions

This section introduces the underlying fault model and test process for positive testing a WSC. Furthermore, it is explained how test cases are generated.

3.2.1. Fault model and test process. A CES (see Definition 6) describes a specific execution of a WSC that has to be enforced during testing. Thus, it is expected that exactly those events in the specified order are executed. According to this, the following faults might occur during the execution:

DOI: 10.1002/spe

- there are calls to services that are *not defined* in the CES,
- there are *missing* calls to services that are defined in the CES,
- the *sequence* of calls is different from the sequence given by the CES, and
- the *parameter* of calls to the invoked services does not correspond to the expected ones.

To cause and control a specific CES of the WSC, it is often inevitable to take control of partner services because they communicate with the SUC and the flow of the WSC might depend on a returned response. The modeled constraints of DTs enable to validate the data passed to the service operations. If the passed data values do not fit to the constraints, we have an irregular behavior. In this case, before generating an error message, input data for the initial WSC call have to be selected. Therefore, an initial DT is to be built consisting of modeled constraints that are associated with the initial input data. However, selecting associated constraints requires the knowledge on the intended behavior of the SUC and especially the expected responses of operation calls. Thus, CESs that describe an execution of the SUC have to be generated first. The overall test process consists of the following steps:

1. Generate CESs based on the given ESG4WSC-model.
2. Create a DT for the public WSC-requests of each CES.
3. Generate sets of input data for public WSC-requests.
4. Execute the SUC with the different input data generated. During the execution,
   (a) observe calls to invoked services during execution and
   (b) send back the expected response (if necessary) for the invoked service according to the CES.

A simple example for sending back the expected response in Step 4(b) would be a search operation of a partner service. If the test sequence wants to test a path where no result is returned by the partner service, it is required to send back a response to the WSC with no result even if the invoked service returns a result. However, testing the WSC where a result is returned of the partner service will be part of another test sequence. Thus, taking control of the partner service does not influence the fault uncovering capabilities.

3.2.2. Test case generation. To detect faults in the WSC (due to the fault model described previously), a test suite is generated that covers at least all EPs. Covering EPs is related to the often cited criterion of edge coverage as in white-box testing [26] or transition coverage as in finite state machine testing [27]. Moreover, this choice of coverage has been adopted to service integration testing approaches based on message choreography modeling [9] and finite state machines [28]. It has also been shown that most of the faults are found by covering EPs [29, 30]. However, also sequences of higher length can be covered if desired [31].

For covering EPs, the cost should be minimal; that is, CESs generated to cover all EPs should have a minimal total length. The problem of generating minimal CESs is related to the *Chinese postman problem* (*CPP*), as in [31]. The algorithm for deriving CESs from an ESG4WSC is described in following steps:

1. Generate CESs for the refined vertices first (recursive call);
2. Add multiple edges (representing EPs) to the ESG4WSC:
   (a) If a refined vertex has a DT restricting the ongoing execution,
      i. identify the valid successor for each CES with respect to the DT and
      ii. add an edge from the refined vertex to the allowed successor.
   (b) If a refined vertex has not a DT,
      i. add an edge from the refined vertex to the successor (there should be only one) for each CES.
3. Generate CESs according to the CPP algorithm (i.e., cover all EPs by CESs of minimal total length);
4. Replace refined vertices in the resulting CES set of Step 3 with the CESs derived in Step 1 with respect to their allowed successors.

DOI: 10.1002/spe

Note that Step 2 adds multiple edges to the underlying ESG4WSC; that is, every edge represents a CES of the refined vertex and its valid successor. The benefit of this approach is that the resulting CES set derived in Step 3 contains the refined vertex and its corresponding successor as much as needed so that the CESs of Step 1 can be combined completely with the CESs of Step 3 (recall that every EP/edge is to be covered in Step 3). After generating CESs, a DT for the initial WSC call is to be defined and evaluated. Therefore, constraints that are associated to the initial input data are selected and added to the initial WSC call. If a constraint set cannot be fulfilled, CESs can be deleted, for example, when two contradicting constraints are to be satisfied. Algorithm 1 along with Table VII (in the Appendix) gives a detailed description of the CES generation process. Example 3 shows the test generation process for the running example.

Example 3
According to Figure 2, the test generation process looks as follows:

Step 1: Generate CESs for refined vertices.
In this step, the CPP algorithm is applied to the two refining ESG4WSCs in event check (Figure 2). The event sequences of each ESG4WSC are combined with operator ||. The following sequences for refined vertex check have been generated:

```
S1: ⟨⟨BLIS:checkBL, BLIS:inBList⟩ ||
    ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩⟩
S2: ⟨⟨BLIS:checkBL, BLIS:inBList⟩ ||
    ⟨CAS:inDebtorsList,CAS:debtorsFalse⟩⟩
S3: ⟨⟨BLIS:checkBL,BLIS:NotinBList⟩ ||
    ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩⟩
S4: ⟨⟨BLIS:checkBL,BLIS:NotinBList⟩ ||
    ⟨CAS:inDebtorsList,CAS:debtorsFalse⟩⟩
```

Step 2: Add edges.
Event check has a DT that restricts the execution of next events BS:offer and BS:approveBank, in Table II. To cover all rules in this table, event pair `(check, BS:approveBank)` needs to be covered three times (R1, R2, and R3) and event pair `(check,BS:offer)` once (R4). Thus, the algorithm adds the following edges according to Table II:

- Three edges `(check, BS:approveBank)` for sequences S1 to S3;
- One edge `(check, BS:offer)` for sequence S4.

An intermediate ESG4WSC is produced (with extra edges added), as illustrated in Figure 3.

Step 3: Generate CESs.
The CPP algorithm is applied on the intermediate ESG4WSC (produced in Step 2) to produce CESs. In this step, the refined events (e.g., `check`) are considered as simple vertices (Figure 3). The following CESs have been generated (refined vertices and their successor are emphasized with a bold font):

Table II. Decision table for vertex check of Figure 2.

dt_{check}	R1	R2	R3	R4
event: BLIS:inBList happens	T	T	F	F
event: BLIS:NotInBList happens	F	F	T	T
event: CAS:DebtorsTrue happens	T	F	T	F
event: CAS:DebtorsFalse happens	F	T	F	T
BS:offer				✓
BS:approveBank	✓	✓	✓	

DOI: 10.1002/spe

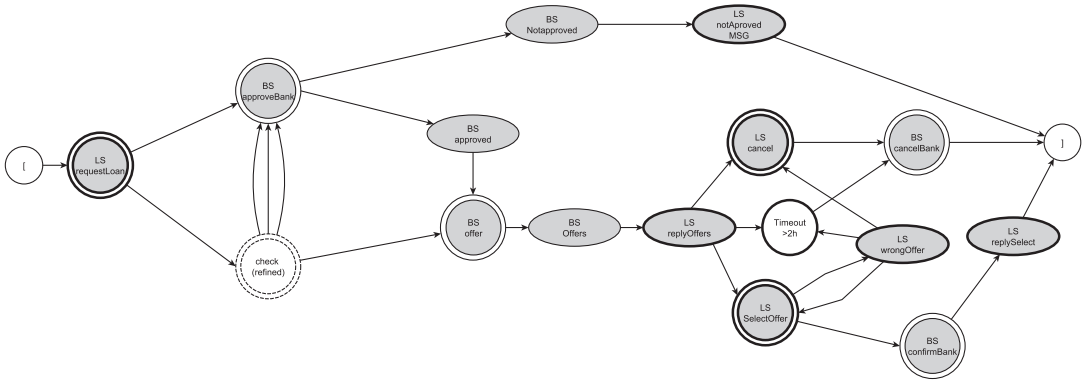

Figure 3. ESG4WSC for the xLoan example extended by additional edges.

CES1: ⟨LS:requestLoan, BS:approveBank, BS:Notapproved, LS:notApprovedMSG⟩

CES2: ⟨LS:requestLoan, **check**, **BS:approveBank**, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:cancel, BS:cancelBank⟩

CES3: ⟨LS:requestLoan, **check**, **BS:approveBank**, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:cancel, BS:cancelBank⟩

CES4: ⟨LS:requestLoan, **check**, **BS:approveBank**, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:SelectOffer, LS:wrongOffer,Timeout > 2h, BS:cancelBank⟩

CES5: ⟨LS:requestLoan, **check**, **BS:offer**, BS:Offers, LS:replyOffers, LS:SelectOffer, BS:confirmBank, LS:replySelect⟩

CES6: ⟨LS:requestLoan, **check**, **BS:offer**, BS:Offers, LS:replyOffers, Timeout > 2h, BS:cancelBank⟩

Step 4: Replace refined vertices by sequences of Step 1.

Refined events are searched in the CES set generated in Step 3 and are replaced by the corresponding CESs derived from the refining ESG4WSCs in Step 1. In the final test suite, event pair (check, BS:approveBank) is covered exactly three times in CES2, CES3, and CES4 using S1, S2, and S3, respectively, to replace check. Event pair (check,BS:offer) is covered twice in CES5 and CES6; S4 is used in both CESs to replace check.

CES1: ⟨LS:requestLoan, BS:approveBank, BS:Notapproved, LS:notApprovedMSG⟩

CES2: ⟨LS:requestLoan, ⟨⟨**BLIS:checkBL, BLIS:inBList**⟩ || ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:cancel, BS:cancelBank⟩

CES3: ⟨LS:requestLoan, ⟨⟨**BLIS:checkBL, BLIS:inBList**⟩ || ⟨CAS:inDebtorsList,CAS:debtorsFalse⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:cancel, BS:cancelBank⟩

CES4: ⟨LS:requestLoan, ⟨⟨**BLIS:checkBL,BLIS:NotinBList**⟩ || ⟨CAS:inDebtorsList,CAS:debtorsTrue⟩⟩, BS:approveBank, BS:approved, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, LS:wrongOffer, LS:SelectOffer, LS:wrongOffer, Timeout > 2h, BS:cancelBank⟩

DOI: 10.1002/spe

```
CES5: ⟨LS:requestLoan, ⟨⟨BLIS:checkBL,BLIS:NotinBList⟩||
      ⟨CAS:inDebtorsList,CAS:debtorsFalse⟩⟩, BS:offer, BS:Offers,
      LS:replyOffers, LS:SelectOffer, BS:confirmBank, LS:replySelect⟩
CES6: ⟨LS:requestLoan, ⟨⟨BLIS:checkBL,BLIS:NotinBList⟩ ||
      ⟨CAS:inDebtorsList,CAS:debtorsFalse⟩⟩, BS:offer, BS:Offers,
      LS:replyOffers, Timeout > 2h, BS:cancelBank⟩
```

The generation of data based on the initial DT is related to the *constraint satisfaction problem* (*CSP*). A CSP is defined by a set of variables $X_1, X_2, \ldots, X_n$ and a set of constraints $C_1, C_2, \ldots, C_m$. Each variable X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset (see [32]). Each rule of the DT under consideration represents a CSP.

The described algorithm is used to generate a test suite that covers all edges, namely EPs. In other words, the test suite covers all ESs with length 2. Similar processes can be performed to cover ESs with higher length k. For this purpose, it is necessary to transform the ESG4WSC model after Step 1 (see [30] for further details on the transformation). The coverage of this graph will deliver the desired test suite. Unfortunately, it might happen that a specific event appears more than once in the resulting model [30]. In this case, it might not be obvious where to add the multiple edges in Step 2 because the corresponding edge could have doubled as well. However, the solution is to remove each doubled edge and add a pseudo vertex instead, which is connected to the source and target of the removed edge. This enables to generate a coverage that contains as many vertices (representing the edges) as needed and in a minimal way [30]. After Step 3, the pseudo-vertices are to be removed from the solution.

3.3. Negative testing of Web service compositions

The previous section described the testing process for expected/desired situations. However, it is also important to test undesired situations where partner services do not work as expected. Thus, a holistic approach is worthwhile that generates positive (desired) and negative (undesired) tests.

The negative testing checks separately unexpected behavior in public events and private events. These two cases are represented by *public faulty event sequences* (*PubFESs*) and *private faulty event sequences* (*PriFESs*). Sections 3.3.1 and 3.3.2 present the definitions and algorithms to generate PubFESs and PriFESs from an ESG4WSC model.

3.3.1. Negative testing of public events. The negative testing for public events involves generating sequences that cover unspecified event pairs for public events, that is, request and response messages of the WSC interface. This part considers that a WSC can be viewed and tested as an atomic service. First, the ESG4WSC is used to derive an ESG4WS representing only public events of the WSC interface [16]. Second, faulty pairs are derived from the ESG4WS using the algorithm proposed in [16]. Finally, each faulty pair is covered by an event sequence that contains a faulty edge at the end (a negative test case). Given an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$, let F be a special event used to represent any *faulty event*, such that $F \notin V$.

Definition 8
The ordered pair $(pes; F)$, such that pes is a PES $\langle v_0, \ldots, v_k, v_{k+1}\rangle$ (see Definition 6), is a *PubFES*, if the last two events of pes are public and there is no edge connecting both; that is, v_k and v_{k+1} are public events and $(v_k, v_{k+1}) \notin V$. In a PubFES, pes is expected to produce a faulty event F.

Example 4
For the `xLoan` example given in Figure 2, the following pair is a PubFES:

```
(⟨LS:requestLoan, BS:approveBank, BS:approved, BS:offer, BS:offers,
LS:replyOffers, LS:cancel, LS:SelectOffer⟩; F)
```


DOI: 10.1002/spe

As there is no edge between the last two events `LS:cancel` and `LS:SelectOffer`, they were selected as a faulty pair. For this undesired case, it is expected that the composition produces a faulty event F. If no faulty event is produced, this test sequence fails; otherwise, it passes.

The algorithm for deriving PubFESs from an ESG4WSC is as follows:

1. Transformation from ESG4WSC to ESG4WS: an ESG4WS is a model that contains only the (public) request and response events for a single service. It can be obtained from an ESG4WSC by removing the private events and keeping edges between any public events v_i and v_j when there exists an ES from v_i to v_j. Thus, the obtained ESG4WS will contain only events related to the composition itself, and there is no private event. A formal description of this transformation is shown in the Appendix (Algorithm 3).
2. Inclusion of faulty edges: in the produced ESG4WS, faulty edges are added between pairs of events with no edges. The formal description of this step is also shown in Algorithm 3.
3. Generation of test sequences: for each faulty edge (v_i, v_j) in the ESG4WS, find a PES pes that leads to v_i in the ESG4WSC. The algorithm to find a PES is implemented by a breadth-first search, from the start event [to v_i, that considers the refining ESG4WSCs involved. A formal description of this procedure is shown in the Appendix (Algorithm 2). Then, append v_j to pes referred to as $pes \oplus v_j$. Finally, create the PubFES $(pes \oplus v_j; F)$.

The formal description of this algorithm for generating PubFESs can be found in the Appendix (Algorithm 4).

Example 5
Using `xLoan`, we obtained the ESG4WS in Figure 4 after the transformation. The faulty edges are represented by gray dashed lines. The dotted lines connect the request events with the fault that must be produced afterwards. The faulty edges are created by

- connecting all response events with request events,
- connecting the start event [with all request events, and
- connecting request events with request events,

in case that there is no edge connecting them. The self-loop from `LS:cancel` to `LS:cancel` was also considered because it represents a one-way operation.

After obtaining the ESG4WS, test cases have to be generated to cover the following faulty edges:

```
(LS:notApprovedMSG, LS:requestLoan); (LS:notApprovedMSG, LS:cancel);
(LS:notApprovedMSG, LS:SelectOffer); (LS:replyOffers, LS:requestLoan);
(LS:wrongOffer, LS:requestLoan); (LS:replySelect, LS:requestLoan);
```

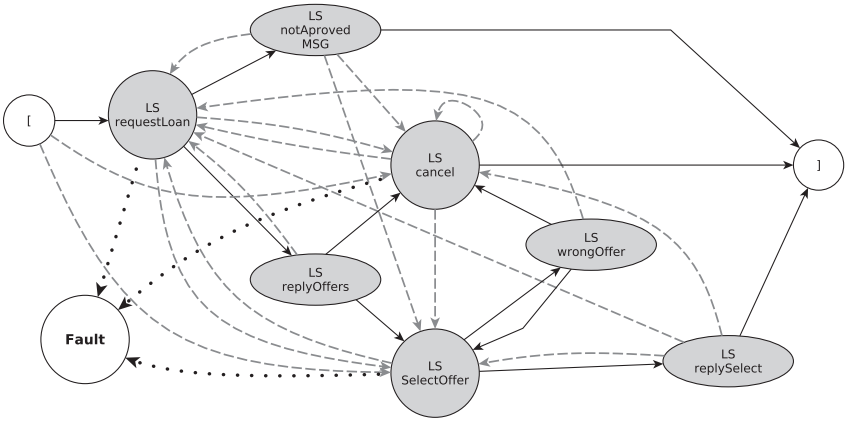

Figure 4. ESG4WS for the `xLoan` public interface.

DOI: 10.1002/spe

```
(LS:replySelect, LS:SelectOffer); (LS:replySelect, LS:cancel); (LS:cancel,
LS:requestLoan); (LS:cancel, LS:SelectOffer); (LS:cancel, LS:cancel);
(LS:requestLoan, LS:cancel); (LS:requestLoan, LS:SelectOffer);(LS:SelectOffer,
LS:cancel); (LS:SelectOffer, LS:requestLoan); ([, LS:cancel); ([,
LS:SelectOffer)
```

Then, for each faulty edge (v_i, v_j), a PES from the ESG4WSC (Figure 2) is derived to reach the event v_i. The event v_j is included after v_i, and the faulty event F is added to the tuple. For the faulty edge `(LS:replySelect, LS:cancel)`, the following PubFES is obtained:

`(⟨LS:requestLoan, ⟨⟨BLIS:checkBL,BLIS:NotinBList⟩ || ⟨CAS:inDebtorsList, CAS:debtorsFalse⟩⟩, BS:offer, BS:Offers, LS:replyOffers, LS:SelectOffer, BS:confirmBank,` **`LS:replySelect, LS:cancel`**`⟩;` F `)`

This PubFES tests a scenario in which the client successfully selects an offer (event `LS:replySelect` is a confirmation message) and tries to cancel it afterwards. This is not adequate for the composition, and a fault should be produced. To generate the final test suite of public negative tests, the same procedure is repeated for each faulty edge.

3.3.2. Negative testing of private events. The proper functioning of a WSC depends on not only its correct implementation but also the partner services. It is often not clearly defined what happens if an invoked service is not working as expected. In this section, we propose an algorithm to generate test cases that cover unexpected behavior of partner services. Event sequences are produced to reach private request and response events and create undesirable situations according to some predefined fault classes. The concept of sensitive events is also proposed in this section to deal with the oracle problem in negative testing of private events.

In this work, seven fault classes are defined on the basis of fault taxonomy and fault injection literature [10, 11, 33]. The fault classes are as follows:

No response: The invoked service does not send back a response for a request-response operation, for example, because of internal problems or modified behaviors.

Long time response: The invoked service needs inappropriate long time to send a response back.

Missing service: The service is missing; for example, the server hosting the service is not available or the service address (URL) has changed.

Unexpected fault: It can be an unexpected SOAP Fault returned by the invoked service or a fault produced by the environment, for example, a fault caused by pre/post-processing in the ESB.

Wrong XML schema: The invoked service sends back an unexpected XML Schema, for example, due to some (untold) changes of the service by the provider.

Wrong XML syntax: The response of the invoked service contains a corrupted XML file, for example, due to some noise in the network.

Right schema, wrong data: The response is a well-formed message that contains invalid data, for example, an invalid date.

Testing these undesirable situations is important to the robustness of the given WSC. Depending on the number of partner services, this results in some additional testing efforts. Fault classes 'no response', 'missing service', and 'unexpected fault' can be tested for every private request vertex of the ESG4WSC model. Fault classes 'longtime response', 'wrong XML schema', 'wrong XML syntax', and 'right schema, wrong data' can be tested for every private response of an ESG4WSC. Table III summarizes this information and also includes the symbols used to represent each class in PriFESs.

When one of the fault classes is provoked, an automated test oracle, which determines the expected test outputs, needs to be established. Certainly, a tester can decide to evaluate and define the expected behavior for every single situation by hand. However, this would mean many manual work and does not scale well. A more straightforward approach is to mark events of the given ESG4WSC as *sensitive*; that is, these events are not allowed to show up after provoking one of the faulty situations. In this case, the sensitive events are used as oracle to automatically evaluate the

DOI: 10.1002/spe

Table III. Fault classes and their relation to events.

			Event of an ESG4WSC	
		Symbol	Request	Response
Fault class	No response	F_{NR}	✓	
	Longtime response	F_{LR}		✓
	Missing service	F_{MS}	✓	
	Unexpected fault	F_{UF}	✓	
	Wrong XML schema	F_{WSc}		✓
	Wrong XML syntax	F_{WSy}		✓
	Right schema, wrong data	F_{WD}		✓

test cases. If after the execution of a faulty event sequence no sensitive event is observed, the test case passes; otherwise, it fails. To our knowledge, there is no other approach that solves the oracle problem in this way for negative testing of WSCs (or any other application).

The tester should identify and highlight the sensitive events in the ESG4WSC. From a theoretical point of view, any event can be marked as sensitive. However, assuming that the sensitive event is caused by the composition under test, public response events and private request events are the main candidates. Among them, the testers needs to analyze which of these events are critical according to the domain of the SUC. The tester should check which events cannot be observed in the SUC if some faulty situation occurs.

Definition 9
Given an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$, the nonempty set $S \subset V$ represents the events marked as *sensitive*. The sensitive events are not allowed to show up after provoking a faulty situation.

Definition 10
Given that pes is a PES $\langle v_0, \ldots, v_k \rangle$, F is any faulty event, and $s \subseteq S$ is a set of sensitive events, the triple $(pes; F; s)$ is a *PriFES* if $v_k \in V_{req} \cup V_{resp}$ is a private event and s is not empty.

Example 6
For the `xLoan` example given in Figure 2, let `LS:replyOffers` be a sensitive event; the following triple is a PriFES:

```
( ⟨LS:requestLoan, ⟨⟨BLIS:checkBL⟩ || ⟨CAS:inDebtorsList, CAS:debtorsFalse⟩⟩;
F_MS; {LS:replyOffers} )
```

In this example, response event `LS:replyOffers` represents the approved loan and its offers. Thus, event `LS:replyOffers` is selected as sensitive because the loan must not be granted if any unexpected behavior happens in the process. F_{MS} represents the fault class 'missing service' that must be provoked; that is, `BLIS` is not available. This sequence passes if no sensitive event (`LS:replyOffers`) is produced by the composition after executing the PES and provoking F_{MS}; otherwise, it fails.

The negative testing for private events generates sequences that cause some unexpected behavior in the partner services, that is, in the private events, and check the service composition in these cases.

The algorithm for deriving PriFESs from an ESG4WSC is as follows:

1. Let V_{RR} be the set of all private request and response events in the ESG4WSC model. Then, for each $e \in V_{RR}$, find the shortest PES pes that reaches e (Algorithm 2 in Appendix);
2. If e is a private request event,
   (a) Copy pes and mark e with F_{NR} to provoke the 'no response' fault;
   (b) Copy pes and mark e with F_{MS} to provoke the 'missing service' fault;
   (c) Copy pes and mark e with F_{UF} to provoke the 'unexpected fault' fault.

DOI: 10.1002/spe

3. If e is a private response event,
   (a) Copy pes and mark e with F_{LR} to provoke the 'longtime response' fault;
   (b) Copy pes and mark e with F_{WSc} to provoke the 'wrong XML schema' fault;
   (c) Copy pes and mark e with F_{WSy} to provoke the 'wrong XML syntax' fault;
   (d) Copy pes and mark e with F_{WD} to provoke the 'right schema, wrong data' fault.
4. For all sequences produced in previous steps, add the set of sensitive events s that is not covered by pes in the PriFES, that is, $(pes; F; s)$.

The formal description of this algorithm to generate PriFESs can be found in the Appendix (Algorithm 5).

Example 7
Consider private request event `CAS:inDebtorsList` of `xLoan` example. The first step is to find the shortest PES that reaches `CAS:inDebtorsList`, pes=⟨`LS:requestLoan`, ⟨⟨`BLIS:checkBL,BLIS:NotinBList`⟩ || ⟨**`CAS:inDebtorsList`**⟩⟩⟩.

Notice that `CAS:inDebtorsList` is part of the refined event `check`. In this case, CESs must be generated for the other parallel ESG4WSCs so that there is no influence on event `CAS:inDebtorsList` and the provoked fault.

Next, pes is copied to pes_1, and event `CAS:inDebtorsList` is marked with F_{NR}. This privFES tests the scenario in which the WSC calls operation inDebtorsList, and no answer/response is sent back. The same procedure is performed for F_{MS} and F_{UF}, with the copies pes_2 and pes_3, respectively.

Let $s = \{$`LS:replyOffers`$\}$ be the set of sensitive events, the resulting PriFESs look as follows: $(pes_1; F_{NR}; s)$, $(pes_2; F_{MS}; s)$, and $(pes_3; F_{UF}; s)$. As the client reputation must be good in both services, `BLIS` and `CAS`, any fault in event `CAS:inDebtorsList` must not produce a successful approval represented by `LS:replyOffers`, marked as sensitive. The same steps are repeated for all other private request and response events.

4. CASE STUDY

This section describes the evaluation of the presented approach for positive and negative testing of WSCs. It presents the application used as subject in the case study, as well as its configuration and results. Lessons learned, discussion, and limitations are also presented.

4.1. System under consideration

The case study was conducted using the `xTripHandling` application, which is based on different scenarios proposed in technical and research literature [5, 34, 35]. The application was developed using SOA concepts and Web services and provides a set of facilities to query and book a trip. It also includes facilities to book trains, rent a car, book sightseeing, and order maps. The application consists of eight services, where six are atomic services and two are composite services. The atomic services are as follows:

1. *ISELTA-hotel Service* is a Web service provided by the commercial system ISELTA that enables travel and touristic enterprises to create their individual search and service offering masks [16]. It provides operations to query hotels and manage bookings.
2. *Airlines Service* provides a set of operations to manage flight tickets, which are similar to ISELTA-hotel service.
3. *Map Service* provides operations to locate places (e.g., airports, train stations) close to a city and order maps for certain cities.
4. *Car Rental Service* provides operations to search and rent vehicles to be used in a pre-defined city.
5. *Train Service* provides operations to check train lines between cities and buy train tickets.
6. *Sightseeing Service* provides operations to list available cities in which the service operates and to buy tickets for sightseeing.

DOI: 10.1002/spe

The composite services are as follows:

1. *Travel Agent Service* provides a set of facilities to query and book a trip. *Travel Agent Service* interacts with two services, ISELTA-hotel and Airlines services. It combines these two services, providing operations to search and book a travel involving flight and hotel reservation. As the flight ticket and hotel reservation are essential in any travel, a successful booking using this service guarantees hotel and flight reservations.
2. *Customer Service* combines the services *Travel Agent, Airlines, Map, Car Rental, Sightseeing, and Train* to provide a centralized resource for customers for managing a complete travel, including hotels, flights, maps, trains, cars, and sightseeing.

Figure 5 illustrates the services, their interfaces, and the interactions of composite services. The figure presents a summarized version of the information available in the WSDL interfaces. The dashed edges represent the interaction between composite services and partner services.

The interface of *Travel Agent Service* provides three operations: queryTrip, getAllOptions, and book. The *Travel Agent* workflow contains the following steps:

1. queryTrip is invoked with the search data
   (a) If there is some invalid data, TripInputException is thrown, and the process finishes (see Step 7).
2. The search data are mapped to the search operations of ISELTA-hotel and Airlines services.
3. Both search operations are called concurrently.
   (a) ISELTA-hotel service requires a login operation before the search.
   (b) If one of the services (ISELTA, Airlines) throws an exception, TripInputException is thrown and the process finishes (see Step 7).
4. Check if at least one hotel and one flight were returned. Otherwise, it throws a TripBookingException with a message showing that there is either no hotel or no flight, or both.

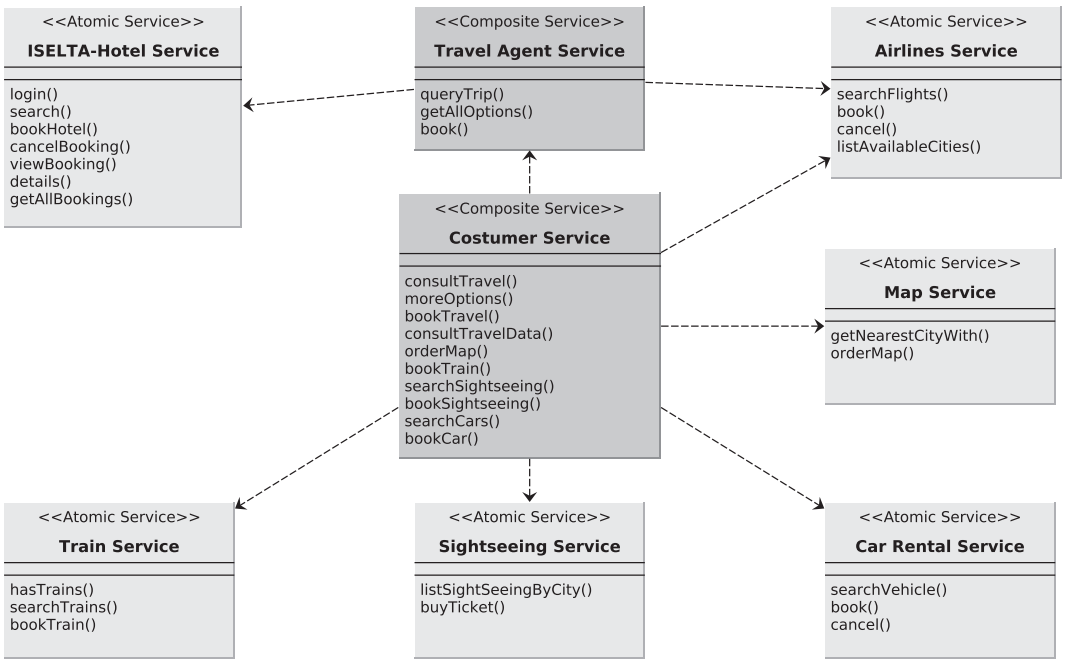

Figure 5. Service interfaces in `xTripHandling`.

DOI: 10.1002/spe

5. Generate a search code and return a response for operation queryTrip with the five cheapest prices of flights and hotels.
6. At this point, the process will wait for one of three possible events:
   (a) After 5 min without a new request, the process finishes (see Step 7).
   (b) getAllOptions request will reply all options for hotels and flights. If the search code is invalid (process finishes or does not exist), TripInputException is thrown and the timeout is restarted.
   (c) book request.
      i. Throw TripInputException if some input data are invalid or the process is finished.
      ii. If all input data are correct, then the booking operations for hotel and flight are called concurrently. A booking message is returned if the bookings of hotel and flight have been successful. The process finishes (see Step 7).
         A. For booking a hotel, login and search operations are required first because there is a timeout of 30 s for the authentication. If the search does not return the same combination of hotel and price to be booked, a TripBookingException is thrown.
      iii. If some fault happens during the hotel and flight booking, TripBookingException is thrown with the problem description. The consumer can try again coming back to Step 6.
         A. Successful hotel or flight booking should be canceled. To cancel a hotel, you need to login again first. If the cancelation fails, a TripBookingException is thrown with a message 'Contact administrator, code: `$searchcode`!'.
7. If the process finishes,
   (a) getAllOptions leads to a TripInputException (always, i.e., independent of input values).
   (b) book leads to a TripInputException (always, i.e., independent of input values).

The description of *Customer Service* is not included in this paper because it is not relevant for the case study explanation. The complete workflow specification for the composite services (*Customer Service* and *Travel Agent Service*), as well as the interface descriptions (WSDL and Java interfaces), can be found on the Internet [‖].

4.2. Configuration and results

The approach proposed in this paper was applied to test the composite services during the development process of the `xTripHandling` application. The case study involved a developer and a tester. The developer described a functional specification, which was used to implement the services. The specification and service interfaces were provided to the tester that created ESG4WSC models and associated DTs. The tester had no access to the source code to avoid mixing modeling strategies. Because the approach is black-box-oriented, an access to the source code is not necessary. Test cases were derived according to the ESG4WSC approach. The tester also performed the concretization and execution of the test cases.

The correction of faults in the case study was based on the following scheme: When the tester finds a fault (some test case fails), the testing process stops. The tester analyzes if the fault is in the specification or in the model. If both the specification and the model are correct with respect to the test case, the tester concludes that the fault is in the implementation. (i) If the fault is in the specification, developer and tester update the specification, and the tester updates the model on the basis of the new specification version. (ii) If the fault is in the model, the tester updates the model to correct the mistake. In case of a change in the model, test cases are regenerated, and the tester resumes the testing process. (iii) If the fault is in the implementation, the developer executes the test

[‖]Website: http://www.labes.icmc.usp.br/∼aendo/esg4wsc

DOI: 10.1002/spe

Table IV. Test model information.

		Travel Agent	Customer Service
1:	# request events	15	204
2:	# response events	34	449
3:	# generic events	1	13
4:	# refined events	2	34
5:	# events (total)	52	700
6:	# edges (total)	75	947
7:	# refining ESG4WSCs	4	68
8:	# ESGs in parallel	4	44
9:	# DTs	7	108
10:	# constraints	32	197
11:	# rules	46	300
12:	initial modeling time	~8h	~20h

case and corrects the fault with the restriction that all previously executed test cases, including the failed test case, must also pass. This scheme is followed until all positive and negative test cases are successfully executed.

The ESG4WSC approach was applied to test the two composite services in `xTripHandling`, *Travel Agent Service* and *Customer Service*. First, *Travel Agent Service* was tested because its partner services are just atomic ones. Then, the approach was applied to *Customer Service*. The testing process based on the established fault correction scheme had several iterations, producing different versions of models and test suites.

Table IV summarizes the information about the ESG4WSC models on the basis of their last versions. Lines 1–4 refer to the number of each type of event. Lines 5 and 6 show the total number of events and edges, respectively. Line 7 refers to the number of refining ESG4WSCs and those that are to be executed in parallel in Line 8. Lines 9–11 show the number of DTs, constraints, and rules, respectively. Line 12 refers to the elapsed time to study the specification and interfaces and produce the first model version.

The model for *Travel Agent Service* is smaller because it involves two partner services. The initial modeling time was around 8 h. The model for *Customer Service* is considerably more complex and larger involving a great number of events and edges. Its initial modeling time was approximately 20 h.

Refining ESG4WSCs in *Travel Agent Service* were used exclusively to represent parallel execution; that is, the four refining ESG4WSCs are in parallel. In *Customer Service*, the model contains 34 refined events and 68 refining ESG4WSCs, 44 being in parallel. Notice that 24 refining ESG4WSCs are not in parallel and were used to modularize the model. Thus, refined events and refining ESG4WSCs were used to not only represent parallelism but also manage the complexity through hierarchy. For instance, after booking a basic trip (hotel + flight), the client can search and book a car. This workflow can be abstracted as a refined event 'rentCar' and its details expressed in an associated refining ESG4WSC. Similar refined events were defined for maps, sightseeing, and trains, using the hierarchy of refining ESG4WSCs to organize the model.

In both models, there are more response events than request events because a request message can have several relevant response messages and instances of response messages. For example, a search request can return one of the following responses: (i) a message with zero items, (ii) a message with one or more items, or (iii) an expected fault. Generic events facilitate the description of time constraints or changing points. DTs mainly supplement public request events for which input data must be generated. The constraints are defined over request parameters, and rules test different combinations of these constraints. In addition, DTs were also used to prune extra edges in refined events with parallel execution.

The designed test models are used as input by the supporting tool (described in Section 5) to generate test suites according to the holistic ESG4WSC approach. Table V summarizes the information about the test suites, divided into positive and negative testing. The number of executed

DOI: 10.1002/spe

Table V. Test suite information.

	Travel Agent	Customer Service
	Positive testing	
#test cases ($k = 2$)	25	1054
#executed events ($k = 2$)	388	89,536
#test cases ($k = 3$)	33	998
#executed events ($k = 3$)	540	139,388
#test cases ($k = 4$)	49	20,537
#executed events ($k = 4$)	922	3,406,148
#test cases (total)	107 ($25 + 33 + 49$)	22,589 ($1054 + 998 + 20,537$)
#executed events	1850	3,635,072
	Negative testing	
#PubFESs	181	6535
#executed events PUBFESs	1706	152,125
#PriFES (F_{NR})	10	95
#PriFES (F_{MS})	10	95
#PriFES (F_{UF})	10	95
#PriFES (F_{LR})	24	194
#PriFES (F_{WSc})	24	194
#PriFES (F_{WSy})	24	194
#PriFES (F_{WD})	24	194
#PriFESs	126	1061
#executed events PriFESs	1544	22,486
#test cases (total)	307 ($181 + 126$)	7596 ($6535 + 1061$)
#executed events	3250	174,611

events for each test suite is also provided. Positive test suites are divided by the length of covered ESs ($k = 2$, $k = 3$, and $k = 4$). Negative test suites are divided into public and private cases.

Because the intended coverage is dependent on the model characteristics, the size difference observed in Table IV is also observed in the number of test cases. For the *Travel Agent Service* model, 107 positive test cases and 307 negative test cases were generated. For the *Customer Service* model, 22,589 positive test cases and 7596 negative test cases were generated.

In positive testing, the increase of length k causes a higher number of executed events. However, these test suites do not need to be applied one by one. For instance, if the tester chooses test suites with $k = 4$, test suites with lengths 2 and 3 can be skipped because the test requirements for smaller lengths are contained in $k = 4$ as well (in terms of coverage).

In negative testing, the number of test cases for a given fault class is equal to the number of private request or response events (as shown in Table III). That is why F_{NR}, F_{MS}, and F_{UF} have the same number of test cases, similar as F_{LR}, F_{WSc}, F_{WSy}, and F_{WD}.

It is important to emphasize that the number of test cases is only used to show the computational effort in this study. Because the test suites are generated automatically from the model, the manual effort is mainly measured by modeling and concretization activities.

Table VI presents the information about faults detected using positive and negative test suites. Faults are also divided by the artifact, specification (Spec), or implementation (Impl).

After several iterations, the tester found 18 faults in *Travel Agent Service*, 12 in the implementation and six in the specification. The six faults were related to some behavior not described in the specification and have been observed during the test case execution. Faults related to missing and unexpected messages have been detected for both the specification and the implementation.

Customer Service presents a higher number of specification faults because its first specification version was very incomplete. The tester found 12 specification faults while designing the first model version and six other faults afterwards. Although *Customer Service* had a high number of executed test cases, the faults in the implementation were mainly identified during the first executed positive and negative test cases. In the end, 37 faults were detected using positive and negative test suites.

DOI: 10.1002/spe

Table VI. Detected faults information.

Test Suites	Travel Agent		Customer Service	
	Spec	Impl	Spec	Impl
Positive test suites				
$k = 2$	4	11	16	12
$k = 3$	0	0	0	1
$k = 4$	0	0	0	0
Total	4	11	16	13
Negative test suites				
PriFES (F_{NR})	1	0	1	0
PriFES (F_{LR})	0	0	0	0
PriFES (F_{MS})	0	0	0	0
PriFES (F_{UF})	0	0	0	0
PriFES (F_{WSc})	0	0	0	0
PriFES (F_{WSy})	0	0	0	3
PriFES (F_{WD})	0	0	0	0
PubFES (resp-req)	0	0	0	1
PubFES ([-req)	0	0	0	0
PubFES (req-req)	1	1	1	2
Total	2	1	2	6
Positive and negative test suites				
Total	6	12	18	19

Table VI also shows the order in which the test suites were executed. Thus, all positive test suites were applied first using the established fault correction scheme. As a consequence, this order obviously reduced the number of faults to be detected by subsequent test suites. Note that most of the faults were detected by the positive test suites covering sequences of length $k = 2$. This result corresponds to experimental results achieved in previous studies for testing graphical user interfaces [29]. Although test sequences of higher length facilitate the detection of critical faults that can only be detected in specific contexts, test suites with length 2 detected most of the faults [29]. The negative test suites revealed less faults for both compositions because several exceptions were properly handled in the implementation when the developer corrected the detected faults.

The faults in the implementation were mainly identified by testing different rules (from the DTs) and checking expected events and their order. The correction of these faults was not critical and was performed only in the implementation. The specification faults were identified by checking expected event sequences. The specification faults were more critical because the tester needed to modify the ESG4WSC and to regenerate the test cases. Moreover, the specification and, consequently, the implementation were corrected as well. As the modeling task is a learning activity, faults were also introduced and identified in the test models. However, an accurate number of these faults is not available because they were directly and dynamically corrected by the tester.

4.3. Lessons learned

In the case study, the tester designed the entire models before implementing the adaptors for test execution. This strategy fits for cases in which the development phase is ongoing and the implementation is not yet available. However, it takes some time to have test cases executing in the implementation. If the implementation has a preliminary version, the test model and adaptors can be built partially and iteratively to obtain some executable test cases sooner.

Depending on the modeling strategy, the number of test cases can increase very rapidly because the applied methods intend to cover event sequences of a given length in the ESG4WSC. Eliminating irrelevant edges is a strategy to reduce the test cases. For instance, for the *Customer Service* model,

DOI: 10.1002/spe

several edges to and from special vertices [and] were removed. This simple change eliminated a considerable number of test cases.

This paper addressed seven fault classes. Additional classes can be defined and implemented using the current infrastructure. Notice that covering all fault classes generates a high number of negative test cases. If the cost of test execution is critical in the current project, a subset of the negative test suite can be selected. On the basis of the case study experience, a strategy is to concentrate on the fault classes 'longtime response' and 'unexpected fault'. They are usually enough to test the WSC robustness because the fault correction for those classes indirectly handles other fault classes.

The evaluation of negative test results should be performed carefully. The information used in negative testing (undesired situations) is usually misleading, scarce, and even missing. For instance, the 'longtime response' class can expose unplanned issues that must be handled by the composition, such as timeouts in the implementation, missing specification, and incomplete workflows. This fact hinders an accurate and automatic evaluation of test sequences. Therefore, a mechanism was presented to handle this issue by using *sensitive events*. This strategy avoids false positives, but faults can be missed by the test cases. Thus, it is recommended that the tester inspects a subset of each fault class to avoid false negatives.

4.4. Discussion of results and limitations of the approach

In the previous sections, it has been described how to apply the proposed approach. The case study demonstrated that the approach is applicable to a nontrivial Web service-oriented application. Numerous faults were revealed not only in the implementation but also in the specification. Thus, the approach helped to keep implementation and specification. synchronized. Moreover, the specification has been set up first, and the implementation and test model creation have been carried out in parallel by two different persons. Hence, the tester does not need to wait for the implementation to set up a model and derive tests because the new approach is not based on artifacts like BPEL or WS-CDL as in [5, 8, 36].

Orchestration and choreography have not been distinguished because the approach can be applied in both contexts. The only restriction is that the tester has control over messages exchanged by the partner services. The tool that was used to support the test execution (Section 5.2) requires that all messages pass through an ESB. Although ESBs may not be part of the service-oriented application under test, deploying the services in an ESB is a simple task.

The proposed approach assumes that ESG4WSCs in a refined event are independent. This enables a simple way to model some parallelism, and for the example and case study, this was sufficient. It is possible that more complex scenarios occur in WSCs and the tester might also want to test combinations of message interleaving. Although the approach can be adapted to test these scenarios, it is recommended to use specific models and testing techniques for concurrent programs [37, 38].

The case study was conducted to check the approach concerning its applicability to the development of a service-oriented application, invoking composite services of very different sizes. Although it has been noticed that the approach could systematically detect faults, further experimental research is necessary considering alternative application areas and sizes. Moreover, investigating in which scenarios MBT is not cost-effective remains an open topic, for example, when the costs of maintaining a model and generating tests are higher than the costs of maintaining a traditional test suite. The study gave evidences that the ESG4WSC model is applicable and intuitive to test WSCs. The approach also has a few assumptions and supporting tools (Section 5), which ease its application to SOA projects.

Along the paper, we presented the target fault classes for both positive and negative testing. During the case study, the results gave evidences that these faults can be revealed by the approach. Besides, different faults were also identified in the specification as observed in MBT literature [39, 40]. However, we do not compare the proposed approach with other techniques, for example, structural testing of WSCs. In other domains, experimental results on this topic have shown that MBT approaches tends to complement the structural testing technique [41]. Further research on this comparison will be carried out in future work.

DOI: 10.1002/spe

The high number of test cases and events executed might be a limitation in some contexts. Thus, a smaller but still effective test suite is desirable. In the previous section, we discussed a strategy to reduce the test suite by modifying the model. Another strategy would be to modify the test case generation algorithms. Intuitively, this seems to be a promising candidate for further research because most of the faults were detected by the first test cases executed. Further investigation on test suite minimization and prioritization can shed light on this topic. However, please note that a smaller test suite will miss some behavior to be tested.

While using MBT, very often the users assume that the underlying model is correct from the beginning. This assumption is a potential threat to the validity because it cannot hold in practice. From a practical point of view, this is a critical assumption because a (formal) model set up on a (informal) specification regularly needs some time to reflect the specification correctly. The tester should always consider that an error might be in the test model and not only in the specification or in the implementation when a fault is detected.

Requirements evolution is also one of the main benefits of MBT [40] and is also shown in our case study. This is particularly relevant for dynamical and loosely coupled environments based on SOAs [2, 3]. As the model is usually smaller and easier to maintain than a large test suite, it is faster to modify the model and regenerate the test suite when the requirements (specification) change [40]. In the conducted case study, if no event is added to test model, the cost of regenerating the test suite is equivalent to the cost of executing the test generation algorithms. This process can be systematically controlled by tracking the changes between the specification and the tests. Nevertheless, the requirement-test traceability is out of the scope of this paper.

5. TOOL SUPPORT

For large models, test generation and execution can hardly be carried out by hand. Therefore, two tools have been developed and used to support test case generation and test execution in the conducted case study.[**]

5.1. Test generation

A tool called *TSD* provides a graphical user interface for the tester to model all features of an ESG4WSC that are necessary for test generation. Figure 6 shows a screenshot of TSD as well as the model set up for testing *Travel Agent Service*. TSD implements the algorithms described in Sections 3.2.2, 3.3.1, and 3.3.2 for generating positive and negative test suites. Furthermore, TSD allows generating test data out of DTs (see Figure 7).

An XML format was defined to describe test cases generated by TSD. The resulting XML files are used as input to the test execution environment. This integrates the test generation and test execution and reduces the dependency between both environments. Thus, new tests can be generated, and no extra effort is necessary for concretization, except for the adaptors.

Figure 8 presents an example of a test case for *Travel Agent Service* in the XML format. The file starts with an element `<TestCase>`, followed by an element `<CompleteEventSequence>` that represents a CES. Events are represented by the element `<Event>` with attributes to label and classify (type and public) the event. Events with associated DT (Line 3) have an attribute to define which rule must be tested and may also include the child elements `<Param>` to provide input data (Lines 4 and 5). A refined event is represented by the element `<RefinedEvent>` (Line 7) and can include one or more CESs. In the example, there are two CESs in parallel, the first in Lines 8–15 and the second in Lines 16–21. When the response event is private and is supposed to answer a specific message, the element `<Message>` can be used within the event, like in Line 31. A predefined SOAP message can be provided in TSD and will be available within a CDATA section.[††] The negative test cases are also represented with special elements and attributes for sensitive events, faulty edges, and fault classes.

[**]The tools and a guide for their use can be found at http://adt.uni-paderborn.de/en/test-tools.html
[††]All text within a CDATA section is ignored by the XML parser.

DOI: 10.1002/spe

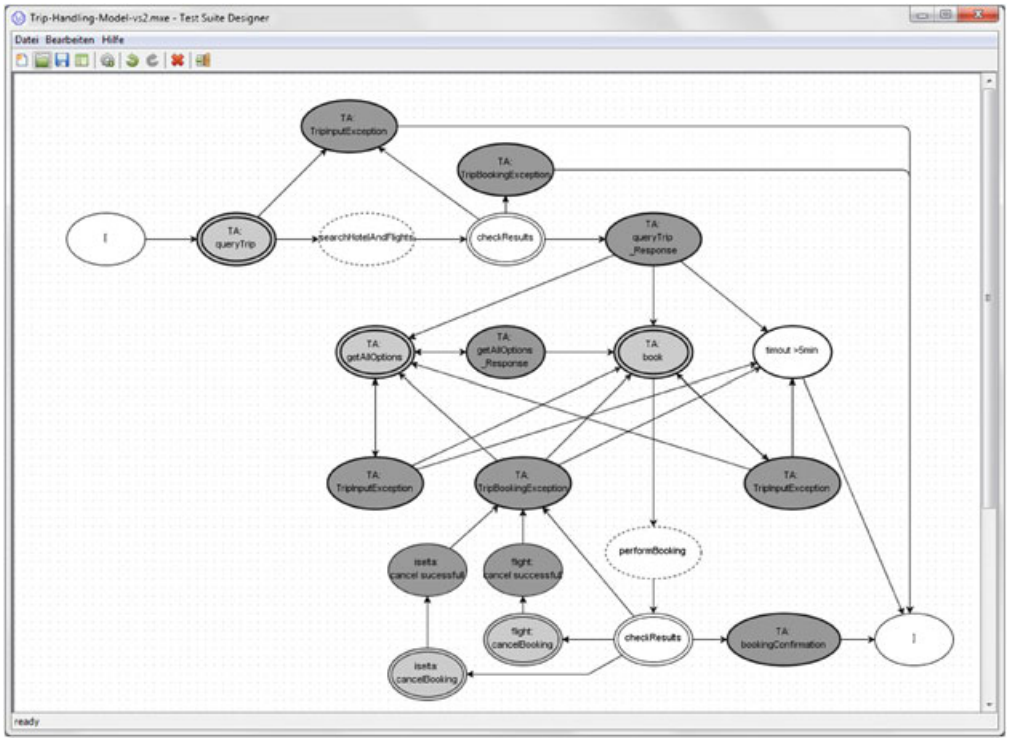

Figure 6. ESG4WSC for testing *Travel Agent Service* in Test Suite Designer.

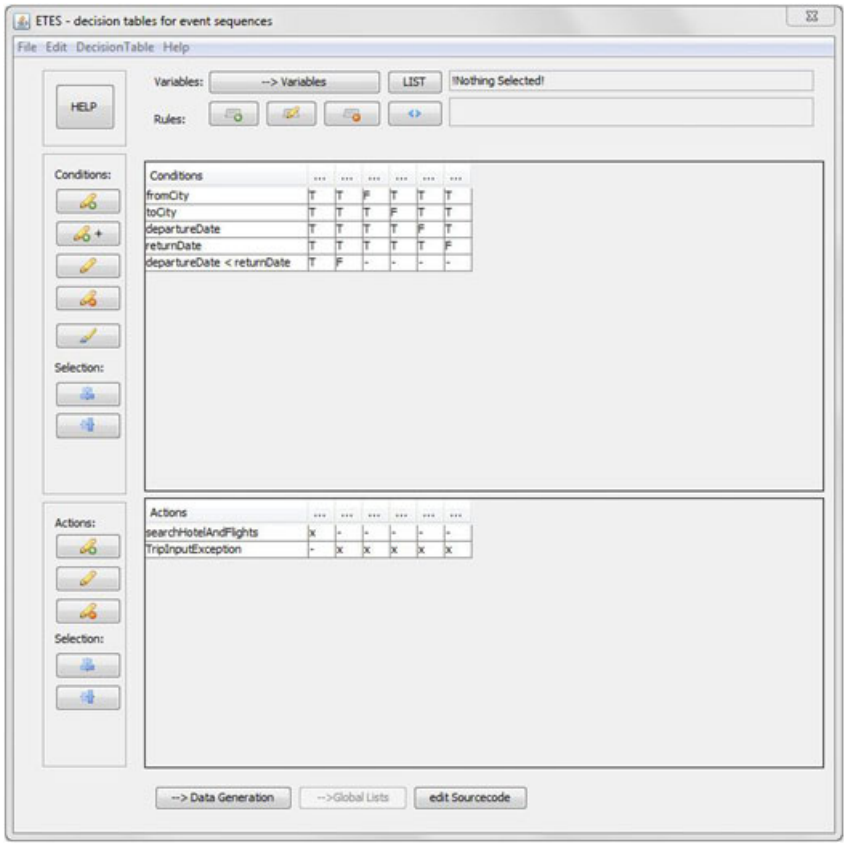

Figure 7. Decision tables in Test Suite Designer.

DOI: 10.1002/spe

```
01:<TestCase>
02: <CompleteEventSequence>
03:  <Event label="TA:queryTrip" type="request" public="true" rule="R1" >
04:   <Param name="departureDate">31.12.2011</Param>
05:   <Param name="toCity">Sao Carlos</Param>
        ...
06:  </Event>
07:  <RefinedEvent>
08:   <CompleteEventSequence>
09:    <Event label="IS:login" type="request" public="false"/>
10:    <Event label="IS:login_Response" type="response" public="false" />
11:    <Event label="IS:search" type="request" public="false"/>
12:    <Event label="IS:searchResults_greaterEqThanOne" type="response" public="false" >
13:     <Message><![CDATA[ ... ]]></Message>
14:    </Event>
15:   </CompleteEventSequence>
16:   <CompleteEventSequence>
17:    <Event label="FL:search" type="request" public="false"/>
18:    <Event label="FL:searchResults_greaterEqThanOne" type="response" public="false" >
19:     <Message><![CDATA[ ... ]]></Message>
20:    </Event>
21:   </CompleteEventSequence>
22:  </RefinedEvent>
23:  <Event label="TA:queryTrip_Response" type="response" public="true" />
24:  <Event label="TA:book" type="request" public="true" rule="R1" >
25:   <Param> ... </Param>
26:  </Event>
27:  <RefinedEvent>
28:   <CompleteEventSequence>
29:    <Event label="FL:book" type="request" public="false"/>
30:    <Event label="FL:bookingSuccess" type="response" public="false" >
31:     <Message><![CDATA[ <soap:Envelope><soap:Body> ... </soap:Envelope> ]]></Message>
32:    </Event>
33:   </CompleteEventSequence>
34:   <CompleteEventSequence>
35:    <Event label="IS:login" type="request" public="false"/>
36:    <Event label="IS:login_Response" type="response" public="false" />
37:    <Event label="IS:search" type="request" public="false"/>
38:    <Event label="IS:searchResults_sameHotelPrice" type="response" public="false" >
39:     <Message><![CDATA[ ... ]]></Message>
40:    </Event>
41:    <Event label="IS:book" type="request" public="false"/>
42:    <Event label="IS:bookingSuccess" type="response" public="false" >
43:     <Message><![CDATA[ ... ]]></Message>
44:    </Event>
45:   </CompleteEventSequence>
46:  </RefinedEvent>
47:  <Event label="TA:bookingConfirmation" type="response" public="true" />
48:  <Event label="TA:getAllOptions" type="request" public="true" rule="R1" >
49:   <Param name="searchCode">[$validSearchCode$]</Param>
50:  </Event>
51:  <Event label="TA:TripInputException" type="response" public="true" />
52: </CompleteEventSequence>
53:</TestCase>
```

Figure 8. XML file for a test case.

5.2. *Test execution*

Mule-ESB [24] is used as the infrastructure software. Initially, all services involved in the composition (including the composite service) are deployed in the bus; that is, the entire communication (SOAP messages) passes through the ESB before reaching the destination service. The test execution is supported by a tool named ERunTE, which is composed of three modules, a Web service (ERunTE-service), an ESB component (ERunTE-esbcomp), and an event runner (ERunTE-runner). ERunTE-service contains four main operations:

- `startObservation` prepares the test execution and identifies the start of a new test case. The partner services whose messages will be modified are passed as parameters;
- `modifyMessage` sets the responsive message for a certain service. This operation is useful to force an event and test a specific scenario. Besides the expected events (positive tests), this operation is also used to provoke the fault classes ‘unexpected fault’, ‘wrong XML schema’, ‘wrong XML syntax’, ‘right schema, wrong data’, and ‘missing service’. In the ‘missing service’ class, this operation turns off the proxy of the service; that is, it simulates an unavailable service;
- `modifyMessageWithTimeout` is similar to the previous operation, but it is possible to set up a delay to answer the response. This operation is used to support the fault classes ‘longtime response’ and ‘no response’;

DOI: 10.1002/spe

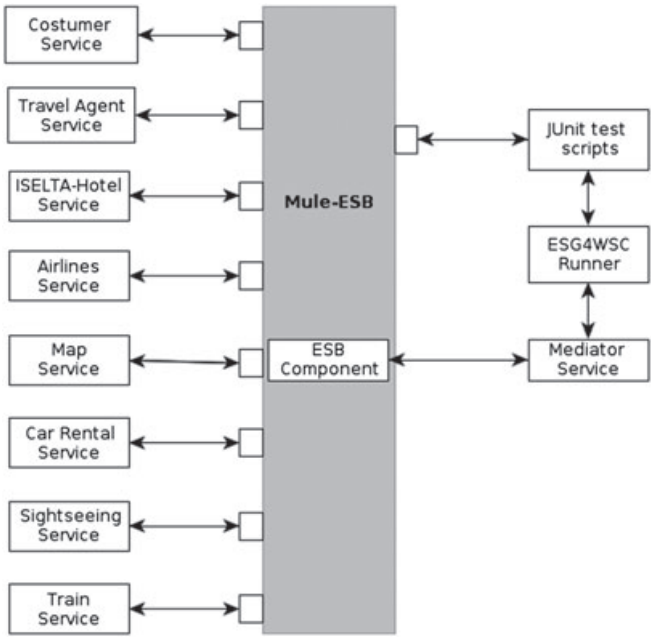

Figure 9. Architecture to execute the tests.

- `getAllMessages` retrieves all messages that pass through the ESB after the last *startObservation* call.

ERunTE-service can be used directly in the test code. In the test environment, it has been integrated in ERunTE-runner. The second module, ERunTE-esbcomp, is the ESB component that implements the monitor and aggregator patterns. This component is integrated with mule-ESB and is able to interact with ERunTE-service. The component records all messages that pass through the ESB and also modifies messages according to operation `modifyMessage`. Figure 9 summarizes the architecture adopted to execute the tests in this case study.

The test cases generated using the ESG4WSC approach were implemented using Java/JUnit and executed with ERunTE-runner. Two adaptors are necessary to execute a test sequence like the one presented in Figure 8: `PublicEventAdaptor` and `MessageCheckingAdaptor`. `PublicEventAdaptor` is responsible for invoking and checking the messages of the WSC interface, that is, the public request and response events. `MessageCheckingAdaptor` is responsible for checking SOAP messages produced during the test case execution. The adaptors were developed as classes with specific Java annotations for allowing ERunTE-runner to map methods to events in the test model.

ERunTE-runner uses these two adaptors and the test sequence in XML generated by TSD to execute the tests. It works in three phases:

1. *Setting up private messages*: the test sequence is read by the runner that traverses the event sequence. It calls ERunTE-service to set up the messages for each private response. The message that must be returned is retrieved from the test case file, as presented in Figure 8 Line 31. The order is defined by counting previous response events of the same service. After this phase, the ERunTE tool has the configuration for executing the test case.
2. *Calling the public interface*: the test execution starts with requests for the WSC. Thus, `PublicEventAdaptor` is used to call the public events and check their responses. The test sequence is traversed and public events are executed by calling the respective methods in `PublicEventAdaptor`. If some response of the WSC interface is different from the expected in the test sequence, the test case fails. Figure 10 presents a code snippet for `PublicEventAdaptor`. Line 3 presents annotation *@Event* for event 'TA:queryTrip'.

DOI: 10.1002/spe

```
01:public class PublicEventsAdaptor {
02:  ...
03:  @Event(label="TA:queryTrip", rule="R2")
04:  public boolean m01() {
05:    S_exception = null;
06:    TripSearchData tripSearchData = new TripSearchData();
07:    tripSearchData.setFromCity("Paderborn");
08:    tripSearchData.setToCity("Sao Paulo");
09:    tripSearchData.setDepartureDate( 2012, 07, 19 );
10:    tripSearchData.setReturnDate( 2012, 07, 13 );
11:    try {
12:      travelAgentService.queryTrip(tripSearchData);
13:    }catch (Exception e) {
14:      S_exception = e;
15:    }
16:    return true;
17:  }
18:  ...
19:  @Event(label="TA:queryTrip_Response")
20:  public boolean m11() {
21:    if(S_tripOptions == null)
22:      return false;
23:    if(S_tripOptions.getFlightInfos().size() != 5)
24:      return false;
25:    if(S_tripOptions.getHotelInfos().size() != 5)
26:      return false;
27:    return true;
28:  }
29:  ...
30:}
```

Figure 10. Sample code for `PublicEventAdaptor`.

It defines that the marked method should be called for event 'TA:queryTrip'. This event has an associated DT, and this method (Lines 4-17) is specific for rule 'R2'. Internal variables (starting with 'S_') are used to store temporary values used by the methods. All methods return a Boolean value to represent a successful/failed execution. For instance, the method in Lines 20–28 returns **true** if event 'TA:queryTrip_Response' is correctly executed and **false**, otherwise.

3. *Checking messages*: all SOAP messages produced during the test case execution are retrieved using ERunTE-service. The messages are checked using `MessageCheckingAdaptor`. The sequence of events is also expected to be followed. If the number of messages, messages, or event order are not in accordance with the test sequence, the test case fails. Figure 11

```
01:public class EventCheckerAdaptor {
02:  ...
03:  @Event(label="TA:queryTrip")
04:  public boolean isTA_queryTrip(String message) {
05:    try {
06:      NamespaceContext context = new NameSpaceContextMap("soap", "http://./soap/envelope/",
                                        "serv", "http://TravelAgent.service.triphandling.cs/");
07:      xpath.setNamespaceContext(context);
08:      Node node = xpath.evaluate( "/soap:Envelope/soap:Body/serv:queryTrip",
                        new InputSource(new StringReader(message)), XPathConstants.NODE);
09:      return node != null;
10:    }
11:    catch (XPathExpressionException e) { }
12:    return false;
13:  }
14:  ...
15:  @Event(label="TA:queryTrip_Response")
16:  public boolean isTA_queryTripResponse(String message) {
17:    try {
18:      NamespaceContext context = new NameSpaceContextMap("soap", "http://./soap/envelope/",
                                        "serv", "http://TravelAgent.service.triphandling.cs/");
19:      xpath.setNamespaceContext(context);
20:      Node node = xpath.evaluate("/soap:Envelope/soap:Body/serv:queryTripResponse",
                        new InputSource(new StringReader(message)), XPathConstants.NODE);
21:      return node != null;
22:    }
23:    catch (XPathExpressionException e) { }
24:    return false;
25:  }
26:  ...
27:}
```

Figure 11. Sample code for `MessageCheckingAdaptor`.

DOI: 10.1002/spe

presents a code snippet for `MessageCheckingAdaptor`. The methods are also marked with annotation *@Event*. The methods return a Boolean value and have a string as parameter. This string represents the message to be checked. The method returns **true** if the message represents the labeled event and **false**, otherwise. In this case study, XPath [42] expressions are used to check whether or not the messages are correct. Lines 4–13 and 16–25 check events 'TA:queryTrip' and 'TA:queryTrip_response', respectively.

The negative testing also requires special configurations of the test execution environment. ERunTE-esbcomp implements a configurable delay for fault classes 'no response' and 'longtime response'. ERunTE-service has an operation to shut down service proxies, helping to reproduce fault class 'missing service'. For fault class 'unexpected fault', possible unexpected fault messages have been identified and simulated. An example is SOAP Faults thrown by Web service frameworks when exceptions are not correctly handled in the application. For fault classes 'wrong XML syntax' and 'wrong XML schema', ERunTE-runner makes small modifications in the original messages to reproduce these faults.

The test generation and execution is fully supported by TSD and ERunTE tools, although some improvements are required in the adaptor development. The version control between model and adaptors is needed, and repeated code can be generated for the adaptors.

6. RELATED WORK

Service-oriented architecture testing has been studied intensively in the last years [4, 43, 44], with a particular effort on formal testing approaches (for a systematic review of the literature, see [45]).

Benharref *et al.* [6] propose a multi-observer architecture to detect and locate faults in composite Web services. The proposed architecture is composed by a global observer and local observers that cooperate to collect and manage faults found in the composite service. The concept of passive testing is used, on the basis of collecting and analyzing traces. The concept of observers is similar to ERunTE and can also be implemented using an ESB. However, the approach aims at actively testing the service composition. A strategy similar to Benharref *et al.* can be implemented using ESG4WSCs and ERunTE.

Mei *et al.* [5] propose a new model to describe a service choreography that manipulates data flow by means of XPath queries. In a choreography, XPath queries can handle different XML schema files. XPath expressions are represented using XPath Rewriting Graphs. On the basis of Labeled Transition Systems (LTS), the LTS-based Choreography model (C-LTS) is proposed with XPath Rewriting Graphs attached in transitions that represent service invocations. New types of definition-use associations are proposed, and test adequacy criteria are presented. The current work is focused on test case generation, which is not approached by Mei *et al.* Thus, both approaches are complementary because the coverage criteria can be used to give more information about the test suite generated by the ESG4WSC approach.

Transforming composition specifications (such as BPEL and WS-CDL) into formal models to support test case generation has also been researched. Bentakouk *et al.* [8] propose a mapping from BPEL to Symbolic Transition Systems. Test cases are generated using symbolic execution and applied to the SUC using online testing. Hou *et al.* [36] model a BPEL program using message sequence graphs and generate message sequences. In an extended version [7], the authors formalize the approach and make an experimental comparison with two other techniques. This work differs from [7, 8, 36] concerning the available artifacts. It is not assumed that the composition was developed using BPEL and the tester has access to this artifact. Although the ESG4WSC approach requires more effort to develop the test model, the SUC is verified from a different black-box point of view.

Wieczorek *et al.* [9] present a model-based integration testing for service choreography using a proprietary model, called message choreography model (MCM). MCM models are translated to Event-B, and test cases are generated using model checking. In [28], the authors present a case study about the application of this MBT approach to test service choreographies in a real-world project. MCM models were designed to support the tests. The work of Wieczorek *et al.* is close to

DOI: 10.1002/spe

the ESG4WSC approach, with two main differences: (i) ESG4WSCs are more abstract models compared with MCMs, which form a domain-specific language created to design service choreography. (ii) The ESG4WSC approach also performs the tests using a test execution environment based on ESBs, which can be adapted to MCM approach.

The negative testing included in this paper can be associated with fault injection techniques for WSCs. Chan *et al.* [33] describe a fault taxonomy for WSCs. The authors identified a set of fault classes, which are classified into physical, development, and interaction faults. In a matrix, these faults are related to six elementary fault classes to explain observed effects in the composition. This taxonomy has been used in fault injection for WSCs.

Cavalli *et al.* [10] propose the framework WebMov, which is composed by a methodology and tools for modeling, validation, and testing of WSCs. It is mainly based on variations of Timed Extended Finite State Machines to model BPEL compositions. The methodology also includes fault injection to test the robustness of the composition. Ilieva *et al.* [11] propose a similar framework, named TASSA, for robustness testing of BPEL orchestrations using fault injection mechanisms. Both papers approach the use of fault injection techniques for negative/robustness testing of BPEL compositions. Nevertheless, they do not deal with faulty message sequences as tested by PubFESs and PriFESs. We believe that the ESG4WSC approach can be deployed in WebMov and TASSA, as well as enriched and benefited from their fault classes and supporting tools.

7. CONCLUSION AND FUTURE WORK

This paper proposed an event-oriented approach, named *ESG4WSC*, for *MBT* of Web service compositions. Test cases are generated on the basis of an ESG4WSC model verifying desired scenarios (positive testing) and unexpected situations (negative testing). A case study was conducted to evaluate the approach. To support the test generation and execution, two tools have been developed, TSD and ERunTE.

ESG4WSC brings the benefits of black-box-oriented MBT to Web service compositions, providing a holistic approach for positive and negative testing. Black-box tests can be generated by modeling an ESG4WSC and observing/modifying messages exchanged in the composition. Thus, faults are detected by observing the exchanged messages. The results of the case study gave evidences that the approach scales well with larger compositions. Moreover, faults have been detected not only in the SUC but also in the specification of a nontrivial service-oriented application.

The approach can be applied to many different scenarios. However, its strength stems from its potential to fit well for cases where BPEL or WS-CDL specifications are not available. Thus, it is independent of the type of composition, either orchestration or choreography, and therefore allows to simultaneously perform the steps for implementation and the testing of the SUC. Other testing approaches strictly require the availability of the artifacts, such as BPEL codes and WS-CDL specifications.

For the first time, it has been described how an ESB can support the test execution. In our case, the ESB has been used to perform the necessary observations and modifications of exchanged messages as well as to provoke the unexpected situations for negative testing.

We plan to investigate how to evolve ESG4WSCs to a state machine to not only consider events but also take states into account and cope with the complex behavior of SUCs in the practice. This may require a considerable increase of the automation power of the presented tools, which is also subject of present research. It is also essential to conduct experimental comparisons with other approaches, such as structural testing of Web service compositions. Further work on online testing based on ESG4WSC models is also necessary. Moreover, the approach may also be extended to verify nonfunctional requirements, such as real-time properties and reliability.

APPENDIX

This appendix presents a formal description of the algorithms proposed in this paper. Table VII shows notations used in the following algorithms.

DOI: 10.1002/spe

Table VII. Legend.

Symbol	*Meaning*	*Example*
$\langle\,\rangle$	The empty sequence	
$\langle a\rangle$	The sequence containing only a	
$\langle a,b,c\rangle$	The sequence with three events, a then b, then c	
$\alpha(s)$	The first event of sequence s	$\alpha(\langle a,b,c\rangle)=a$
$\omega(s)$	The last event of sequence s	$\omega(\langle a,b,c\rangle)=c$
$s[i]$	The ith event of sequence s	$\langle a,b,c\rangle\,[2]=b$
$s[i\ldots j]$	The sequence from i to j	$\langle a,b,c\rangle\,[1\ldots 2]=\langle a,b\rangle$
$s\|\|t$	Sequence s in parallel to sequence t	
$s\oplus t$	Concatenate s and t	$\langle a,b\rangle\oplus\langle c\rangle=\langle a,b,c\rangle$
$N^+(v)$	Successors of a vertex v in an ESG4WSC/ESG4WS	
$N^-(v)$	Predecessors of a vertex v in an ESG4WSC/ESG4WS	

Algorithm 1: generateCESs().

```
function : generateCESs()
input    : an ESG4WSC
output   : CES
1  foreach re ∈ V_refined do
       // step 1
2      resCES ← ∅;
3      foreach esg ∈ re do
4          CES = generateCESs(esg);
5          if resCES==∅ then
6              resCES = resCES ∪ {(re × CES)};
7          else
8              foreach ces_1 ∈ {ces|(re,ces) ∈ resCES} do
9                  foreach ces_2 ∈ CES do
10                     resCES = resCES ∪ {(re,(ces_1||ces_2))};
11                 resCES = resCES \ {(re,ces_1)};
       // step 2
12     if f(re) ≠ ε then
13         DT_seq = f(re);
14         foreach ces ∈ {ces|(re,ces) ∈ resCES} do
15             v = getAllowedSuccessor(ces, DT_seq);
16             E := E ∪ {(re,v)};
               // store a Mapping, i.e., Map ⊆ E × ces
17             Map := Map ∪ {((re,v),ces)};
18             resCES := resCES \ {(re,ces)};
       // add multiple edges for each dataset to be tested
19     foreach DT_input,public ∈ V do
20         foreach a ∈ A do
21             E := E \ {(DT_input, a)};
22         foreach (C_true, C_false, E_x) ∈ R do
23             E := E ∪ {(DT_input, E_x)};
       // step 3
24     CES = solveCPP(ESG4WSC);
       // step 4
25     foreach ves ∈ CES do
26         for i = 1 to #ces do
27             if ces[i] ∈ V_refined then
28                 if |{ces|((ces[i],ces[i+1]),ces) ∈ Map]}| > 0 then
29                     new = es with es ∈ {ces|((ces[i],ces[i+1]),ces) ∈ Map]};
30                     Map = Map \ {((ces[i],ces[i+1]),ces)};
31                     ces = ces[1..(i−1)] ⊕ new ⊕ ces[(i+1)..#ces];
32                 else if |{ces|(ces[i],ces) ∈ resCES}| > 0 then
33                     new = es with es ∈ {ces|(ces[i],ces) ∈ resCES]};
34                     resCES = resCES \ {(ces[i],ces)};
35                     ces = ces[1..(i−1)] ⊕ new ⊕ ces[(i+1)..#ces];
36 return CES;
```


DOI: 10.1002/spe

Algorithm 2: Algorithm to generate a partial event sequence that covers an event e_i.

function : generatePES()
input : an ESG4WSC, a nonrefining event e_i
output : PES

$re = \epsilon$;
if $e_i \in V$ **then**
  $e_k = e_i$;
else
  Select $re \in V_{\text{refined}}$ such that e_i is within re;
  $e_k = re$;
$pes = getShortestPath(\Xi, e_k)$;
// solve the refined events in pes
foreach $re_i \in \{re_i \mid re_i \in V_{\text{refined}} \setminus \{re\}$ *and* $re_i \in pes\}$ **do**
  // this procedure also handles DTs
  $pes = solveCESforRefinedEvent(pes, re_i)$;
if $re \neq \epsilon$ **then**
  $par = \{\}$;
  foreach $esg4wsc \in re$ **do**
    if $e_i \in esg4wsc$ **then**
      // recursive call
      $pes_i = generatePES(esg4wsc, e_i)$;
    else
      $pes_i = getShortestCES(esg4wsc)$;
    $par = par || pes_i$;
  $pes = replace(pes, \omega(pes), par)$;
return *pes*;

Algorithm 3: Algorithm to transform an ESG4WSC to an ESG4WS and obtain the faulty edges.

function : transform()
input : an $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$
output : an $ESG4WS = (V', E', \Xi', \Gamma')$, faulty edges FE

$ESG4WS = (V', E', \Xi', \Gamma')$;
foreach $v \in V$ **do**
  if *(*$v \notin V_{\text{req}}$ *AND* $v \notin V_{\text{resp}}$*) OR* v *is private* **then**
    foreach $pre \in N^-(v)$ **do** //predecessors of v
      foreach $post \in N^+(v)$ **do** //successors of v
        if $(pre, post) \notin E$ **then**
          $E := E \cup \{(pre, post)\}$;
        $E := E \setminus \{(v, post)\}$;
      $E := E \setminus \{(pre, v)\}$;
    $V := V \setminus v$;
    if $v \in \Xi$ **then** $\Xi := \Xi \setminus \{v\}$;
    if $v \in \Gamma$ **then** $\Gamma := \Gamma \setminus \{v\}$;
$V' := V$; $E' := E$; $\Xi' := \Xi$; $\Gamma' := \Gamma$;
$FE := \{\}$;
foreach $v_{\text{req}} \in V'_{\text{req}}$ **do**
  foreach $v_{\text{resp}} \in V'_{\text{resp}}$ **do**
    if $(v_{\text{resp}}, v_{\text{req}}) \notin E'$ **then**
      $FE := FE \cup \{(v_{\text{resp}}, v_{\text{req}})\}$;
  foreach $v_{\text{req2}} \in V'_{\text{req}}$ **do**
    if $v_{\text{req2}} \neq v_{\text{req}}$ *AND* $(v_{\text{req2}}, v_{\text{req}}) \notin E'$ **then**
      $FE := FE \cup \{(v_{\text{req2}}, v_{\text{req}})\}$;
  if $v_{\text{req}} \notin \Xi'$ **then**
    $FE := FE \cup \{([, v_{\text{req}})\}$;
return *ESG4WS, FE*;

DOI: 10.1002/spe

Algorithm 1 gives a formal description for the algorithm to generate CESs that cover all event pairs (positive test cases) from an ESG4WSC. This algorithm is described along with an example in Section 3.2.2.

Algorithm 2 gives a formal description for the algorithm that generates a PES to reach a given event. Examples of PESs generated using this algorithm can be found in Sections 3.3.1 and 3.3.2.

Algorithm 3 gives a formal description for the algorithm that translates an ESG4WSC to an ESG4WS and produces faulty event pairs from the latter. An example of this transformation and faulty edges can be found in Section 3.3.1.

Algorithm 4 gives a formal description for the algorithm to generate PubFESs from an ESG4WSC. This algorithm is described along with an example in Section 3.3.1.

Algorithm 5 gives a formal description for the algorithm to generate PriFESs from an ESG4WSC. This algorithm is described along with an example in Section 3.3.2.

Algorithm 4: Algorithm to generate PubFESs.

function : generatePubFESs()
input : an ESG4WSC, faulty edges FE
output : the test suite PubFES

PubFES $= \{\}$;
foreach $(e_i, e_j) \in FE$ **do**
  $pes_i = generatePES(\mathrm{ESG4WSC}, e_i)$;
  $pes_i = pes_i \oplus e_j$;
  PubFES = PubFES $\cup \{(pes_i; F)\}$;
return *PubFES*;

Algorithm 5: Algorithm to generate PriFESs.

function : generatePriFESs()
input : an ESG4WSC, set of sensitive events s
output : the test suite PriFES

PriFES $= \{\}$;
Let V_{REQ} be the union of sets V_{req} for the ESG4WSC and their refining ESG4WSCs;
Let V_{RESP} be the union of sets V_{resp} for the ESG4WSC and their refining ESG4WSCs;
foreach $e_i \in V_{\mathrm{REQ}}$ **do**
  $pes_i = generatePES(\mathrm{ESG4WSC}, e_i)$;
  $fes_1 = (pes_i; F_{\mathrm{NR}}; s)$;
  $fes_2 = (pes_i; F_{\mathrm{MS}}; s)$;
  $fes_3 = (pes_i; F_{\mathrm{UF}}; s)$;
  PriFES = PriFES $\cup \{fes_1, fes_2, fes_3\}$;
foreach $e_j \in V_{\mathrm{RESP}}$ **do**
  $pes_j = generatePES(\mathrm{ESG4WSC}, e_j)$;
  $fes_1 = (pes_j; F_{\mathrm{LR}}; s)$;
  $fes_2 = (pes_j; F_{\mathrm{WSc}}; s)$;
  $fes_3 = (pes_j; F_{\mathrm{WSy}}; s)$;
  $fes_4 = (pes_j; F_{\mathrm{WD}}; s)$;
  PriFES = PriFES $\cup \{fes_1, fes_2, fes_3, fes_4\}$;
return *PriFES*;

ACKNOWLEDGEMENTS

Andre Takeshi Endo is financially supported by FAPESP/Brazil (grant 2009/01486-9) and CAPES/Brazil (grant 0332-11-9). Adenilso Simao is financially supported by CNPq/Brazil (grant 474152/2010-3). The authors are grateful to the anonymous reviewers for their useful comments and suggestions.

REFERENCES

1. Schmidt MT, Hutchison B, Lambros P, Phippen R. The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal* October 2005; **44**:781–797. DOI: 10.1147/sj.444.0781.

2. Josuttis N. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc.: Sebastopol, CA, USA, 2007.
3. Papazoglou MP, Heuvel WJ. Service oriented architectures: approaches, technologies and research issues. *The International Journal on Very Large Databases (VLDB)* 2007; **16**(3):389–415. DOI: 10.1007/s00778-007-0044-3.
4. Canfora G, Di Penta M. Service-oriented architectures testing: a survey. In *Software Engineering: International Summer Schools (ISSSE)*. Springer-Verlag: Berlin, Heidelberg, 2009; 78–105, DOI: 10.1007/978-3-540-95888-8_4.
5. Mei L, Chan WK, Tse TH. Data flow testing of service choreography. In *Symposium on the Foundations of Software Engineering (FSE)*. ACM: New York, NY, USA, 2009; 151–160, DOI: 10.1145/1595696.1595720.
6. Benharref A, Dssouli R, Glitho R, Serhani MA. Towards the testing of composed Web services in 3rd generation networks. In *IFIP International Conference on Testing of Communicating Systems (TESTCOM)*, Vol. 3964/2006. Springer: Berlin/ Heidelberg, Germany, 2006; 118–133, DOI: 10.1007/11754008_8.
7. Ni Y, Hou S, Zhang L, Zhu J, Li Z, Lan Q, Mei H, Sun J. Effective message-sequence generation for testing BPEL programs. *IEEE Transactions on Services Computing* 2011; **PP**(99). DOI: 10.1109/TSC.2011.22.
8. Bentakouk L, Poizat P, Zaïdi F. A formal framework for service orchestration testing based on symbolic transition systems. In *International Conference on Testing of Software and Communication Systems (TESTCOM)*. Springer-Verlag: Berlin, Heidelberg, 2009; 16–32, DOI: 10.1007/978-3-642-05031-2_2.
9. Wieczorek S, Kozyura V, Roth A, Leuschel M, Bendisposto J, Plagge D, Schieferdecker I. Applying model checking to generate model-based integration tests from choreography models. In *International Conference on Testing of Software and Communication Systems (TESTCOM)*. Springer-Verlag: Berlin, Heidelberg, 2009; 179–194, DOI: 10.1007/978-3-642-05031-2_12.
10. Cavalli A, Cao TD, Mallouli W, Martins E, Sadovykh A, Salva S, Zaidi F. WebMov: a dedicated framework for the modelling and testing of Web services composition. *IEEE International Conference on Web Services (ICWS)*, Miami, Florida, USA, 2010; 377 –384, DOI: 10.1109/ICWS.2010.24.
11. Ilieva S, Manova D, Manova I, Bartolini C, Bertolino A, Lonetti F. An automated approach to robustness testing of BPEL orchestrations. *The 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2011)*, Irvine, CA, USA, 2011; 193–203, DOI: 10.1109/SOSE.2011.6139108.
12. Grieskamp W, Kicillof N, Stobie K, Braberman VA. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability* 2011; **21**(1):55–71. DOI: 10.1002/stvr.427.
13. van der Aalst WMP. Formalization and verification of event-driven process chains. *Information & Software Technology* 1999; **41**(10):639–650.
14. Belli F, Budnik CJ, White L. Event-based modelling, analysis and testing of user interactions: approach and case study. *Software Testing, Verification & Reliability* 2006; **16**(1):3–32. DOI: 10.1002/stvr.v16:1.
15. Yuan X, Cohen MB, Memon AM. GUI interaction testing: incorporating event context. *IEEE Transactions on Software Engineering* 2011; **37**(4):559–574.
16. Belli F, Linschulte M. Event-driven modeling and testing of real-time Web services. *Service Oriented Computing and Applications Journal* 2010; **4**(1):3–15. DOI: 10.1007/s11761-010-0056-5.
17. Belli F, Budnik CJ, Linschulte M, Schieferdecker I. Testen Web-basierter Systeme mittels strukturierter, graphischer Modelle—Vergleich anhand einer Fallstudie. In *Gi jahrestagung (2)*. Gesellschaft für Informatik e.V.: Bonn, Germany, 2006; 266–273.
18. Belli F. Finite state testing and analysis of graphical user interfaces. *12th International Symposium on Software Reliability Engineering (ISSRE)*, Hong Kong, China, 2001; 34–43.
19. Belli F, Endo AT, Linschulte M, Simao AS. Model-based testing of Web service compositions. *The 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2011)*, Irvine, CA, USA, 2011; 181–192, DOI: 10.1109/SOSE.2011.6139107.
20. Gudgin M, Hadley M, Mendelsohn N, Moreau JJ, Nielsen HF, Karmarkar A, Lafon Y. SOAP version 1.2 (W3C recommendation), 2007. Available at: http://www.w3.org/TR/soap12-part1/ [last accesed 24 February 2012].
21. Jordan D, Evdemon J, Alves A, Arkin A, Askary S, Barreto C, Bloch B, Curbera F, Ford M, Goland Y, Guízar A, Kartha N, Liu CK, Khalaf R, Konig D, Marin M, Mehta V, Thatte S, van der Rijn D, Yendluri P, Yiu A. OASIS Web Services Business Process Execution Language (WSBPEL) v2.0, Organization for the Advancement of Structured Information Standards, 2007. Available at: http://docs.oasis-open.org/wsbpel/2.0/ [last accessed 24 February 2012].
22. Kavantzas N, Burdett D, Ritzinger G, Fletcher T, Lafon Y, Barreto C. Web Services Choreography Description Language Version 1.0, 2005. Available at: http://www.w3.org/TR/ws-cdl-10/ [last accessed 24 February 2012].
23. Arkin A, Askary S, Fordin S, Jekeli W, Kawaguchi K, Orchard D, Pogliani S, Riemer K, Struble S, Takacsi-Nagy P, Trickovic I, Zimek S. W3C. Web Service Choreography Interface (WSCI) 1.0, 2002. Available at: http://www.w3.org/TR/wsci/ [last accessed 24 February 2012].
24. MuleSoft. Mule ESB: Open source ESB and integration platform. Available at: http://www.mulesoft.org/ [last accessed 24 February 2012].
25. Myers GJ, Sandler C, Badgett T, Thomas TM. *The Art of Software Testing*. John Wiley & Sons, Inc.: Hoboken, New Jersey, 2004.
26. Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys* December 1997; **29**(4):366–427.
27. Ammann P, Offutt J. *Introduction to Software Testing*. Cambridge University Press: New York, NY, USA, 2008.

28. Wieczorek S, Stefanescu A, Roth A. Model-driven service integration testing—a case study. In *International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE Computer Society: Washington, DC, USA, 2010; 292–297, DOI: 10.1109/QUATIC.2010.49.
29. Belli F, Güler N, Linschulte M. Are longer test sequences always better?—a reliability theoretical analysis. *Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, Singapore, Singapore, 2010; 78–85, DOI: 10.1109/SSIRI-C.2010.26.
30. Belli F, Güler N, Linschulte M. Does 'depth' really matter? on the role of model refinement for testing and reliability. *IEEE 35th Annual Computer Software and Applications Conference (COMPSAC)*, Munich, Germany, 2011; 630–639, DOI: 10.1109/COMPSAC.2011.17.
31. Belli F, Budnik CJ. Minimal spanning set for coverage testing of interactive systems. In *First International Colloquium on Theoretical Aspects and Computing (ICTAC)*. Springer Verlag: Berlin, Heidelberg, Germany, 2004; 220–234.
32. Russell S, Norvig P. *Artificial Intelligence: A Modern Approach*, 2nd edition, chap. Constraint Satisfaction Problems. Prentice-Hall: Englewood Cliffs, NJ, 2003; 137–160.
33. Chan KS, Bishop J, Steyn J, Baresi L, Guinea S. A fault taxonomy for Web Service Composition. In *International Conference on Service-Oriented Computing (Workshops)*. Springer Verlag: Berlin, Heidelberg, 2009; 363–375.
34. Sourceforge.net. WS-CDL eclipse (with examples). Available at: http://sourceforge.net/projects/wscdl-eclipse/ [last accessed 24 February 2012].
35. Netbeans.org. NetBeans SOA project home. Available at: http://soa.netbeans.org/ [last accessed 24 February 2012].
36. Hou SS, Zhang L, Lan Q, Mei H, Sun JS. Generating effective test sequences for BPEL testing. In *International Conference on Quality Software (QSIC)*. IEEE Computer Society: Washington, DC, USA, 2009; 331–340, DOI: 10.1109/QSIC.2009.50.
37. Hoare CAR. *Communicating Sequential Processes*. Prentice Hall International, 2004. Available online: http://www.usingcsp.com/.
38. Lei Y, Carver RH. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering* June 2006; **32**(6):382 –403. DOI: 10.1109/TSE.2006.56.
39. Pretschner A, Prenninger W, Wagner S, Kühnel C, Baumgartner M, Sostawa B, Zölch R, Stauner T. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. ACM: New York, NY, USA, 2005; 392–401, DOI: http://doi.acm.org/10.1145/1062455.1062529.
40. Utting M, Legeard B. *Practical Model-based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2006.
41. Mouchawrab S, Briand LC, Labiche Y. Assessing, comparing, and combining statechart-based testing and structural testing: an experiment. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society: Washington, DC, USA, 2007; 41–50, DOI: http://dx.doi.org/10.1109/ESEM.2007.24.
42. Berglund A, Boag S, Chamberlin D, Fernandez MF, Kay M, Robie J, Simeon J. XML Path Language (XPath) 2.0 (second edition), 2010. Available at: http://www.w3.org/TR/xpath20/ [last accessed 24 February 2012].
43. Bozkurt M, Harman M, Hassoun Y. Testing Web services: a survey. *Technical Report TR-10-01*, Department of Computer Science, King's College London, January 2010.
44. Rusli HM, Puteh M, Ibrahim S, Tabatabaei SGH. A comparative evaluation of state-of-the-art Web service composition testing approaches. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST)*. ACM: New York, NY, USA, 2011; 29–35, DOI: 10.1145/1982595.1982602.
45. Endo AT, Simao AS. A systematic review on formal testing approaches for Web services. *4th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, Natal, Brazil, 2010; 89–98.