Model-based Testing of Web Service Compositions

Fevzi Belli*, Andre Takeshi Endo†, Michael Linschulte* and Adenilso Simao†
*Electrical Engineering and Mathematics
University of Paderborn, Germany
Email: {belli,linschu}@upb.de
†Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo (USP), Brazil
Email: {aendo, adenilso}@icmc.usp.br

***Abstract*—The use of web services integrated in different applications, especially the composition of services, brings challenges for testing due to their complex interactions. In this paper, we propose an event-based approach to test web service compositions. The approach is based on event sequence graphs which we extend by facilities to consider the specific features of web service compositions. An enterprise service bus component supports the test case execution. A case study, based on a commercial web application, demonstrates the feasibility of the approach and analyzes its characteristics. The results of empirical work suggest that the approach is a promising candidate to reach a high level of confidence and reliability.**

***Keywords*-enterprise service bus; event sequence graphs; model-based testing; service composition testing**

I. Introduction

Enterprise applications have become more and more complex as they have to cope with many factors, such as business processes, interaction among different companies, and dynamic evolution of business. 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 for this purpose. SOA provides an architectural style for developing loosely coupled and distributed applications by using independent and self-contained services. These services can be combined by defining a workflow that characterizes a new (composite) service. This process is called *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], [2], [3]. The so-called *Enterprise Service Bus* (ESB) is responsible for controlling, routing, and translating messages exchanged by the services [3].

SOA testing has received much attention in order to deliver high quality and robust service-oriented applications [4]. WSC testing has gained in importance as well [5], [6], [7], [8], [9], [10]. The reason is that the behavior of the WSC now depends not only on the WSC itself but also on the integrated services. This complicates testing since the WSC can present complex communications among the integrated services in which missing or unexpected messages can lead to a failure.

A common problem in testing of any kind of application is to derive meaningful test cases automatically. The strategy of using models for test case generation is known as *Model-Based Testing* (MBT). In MBT, a tester utilizes his/her knowledge of a given system under test (SUT) to develop a model to generate test cases. 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, high level of automation, etc. (see, e.g., a detailed evaluation of MBT in [11]). In a previous work, event sequence graphs (ESGs), originally introduced for testing graphical user interfaces [12], were used to model and test stateful web services [13].

In this paper, we propose event-oriented ESG4WSC (ESG for Web Service Composition) for generating cost-effective test cases for web service compositions for several reasons:

- Message exchanges in a WSC can be viewed as events that follow an order.
- ESG modeling is easy to be understood, necessitates little manual work and is supported by specific tools [12].
- In cases, wherever no model is available, an ESG can easily be constructed in an ad-hoc way by the tester.
- To create an ESG or any other model for a service composition, artifacts (e.g. standardized service descriptions) need not be available.

However, we assume that the tester can observe and change the exchanged messages using an ESB (present in several SOAs), i.e., the service composition is considered as a black box, but the tester has control over the messages exchanged by the partner services.

The approach this paper introduces is new, as well as the novel algorithm for generating test cases from the corresponding ESG4WSC that models the service composition.

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

- An event-based approach is adapted to support WSC testing by:

  - extending the basic notions of ESG [12] referred to as ESG4WSC,
  - introducing a new algorithm to derive test cases from ESG4WSC.
- Tools are introduced for the automation of the approach:
  - A tool called Test Suite Designer (TSD) supports modeling the SUT and generating test cases,
  - An ESB-based component enables automatic test execution and analysis.
- A case study demonstrates the applicability of the approach and evaluates it by means of a practical and non-trivial web application that is commercially available. This case study also demonstrates the test execution and the use of an ESB.

To our knowledge, there is no comparable work that performs testing of WSC by modeling and testing not only the published interface (available to the consumer) but also the internal communication with other services, which is usually hidden from the consumer. Also worth to note is that the analysis of the results of the empirical work performed in the case study encourages applying the proposed approach for WSC testing.

The remainder of this paper is organized as follows. Section II briefly presents concepts of SOA, web services, and ESBs. Section III introduces the proposed approach to test service compositions, along with an example. Section IV presents a case study and discusses the results of experiments, considering also tool support for practical project work. Section V shows the related work. Finally, Section VI concludes the paper and sketches the future work planned.

II. 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 at fostering loose coupling among applications. It provides a standardized, distributed, and protocol-independent computing paradigm. Software resources are wrapped as "services" that are well-defined and self-contained modules providing business functionalities and being independent from other service states or contexts [3]. A service is a black box, since implementation details are hidden and only its interface is available. A SOA is based on three entities: provider, consumer, and registry. Using the web services technology, a SOA is realized through three main XML standards: SOAP, WSDL, and UDDI. SOAP is a W3C protocol used to define the structure of messages exchanged among the services. WSDL (Web Service Description Language) is also a W3C standard used to describe the service interface, including details like operations, data types, and adopted protocols. UDDI (Universal Description, Discovery and Integration) is an OASIS standard that defines a set of functionalities to support description and discovery of services.

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. Service composition 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 [14]. 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, e.g., WS-CDL [15] and WSCI [16]. 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. We use the terms *service* and *web service* as synonyms in this work.

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 SOA [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 architecture to an ESB-based architecture. According to Schmidt et al. [1], the ESB is an infrastructure which fully supports an integrated and flexible SOA. These features are reached by receiving messages of services, operating or mediating on them, as they flow through the bus. There are many possible uses for mediation, such as security, 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 of all messages that pass through the ESB, not applying any type of change in the messages. This pattern can be applied to logging, audition, monitor service levels, measure 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, e.g., a set of events can be mapped to a single event.

The functionalities of an ESB can be provided by software

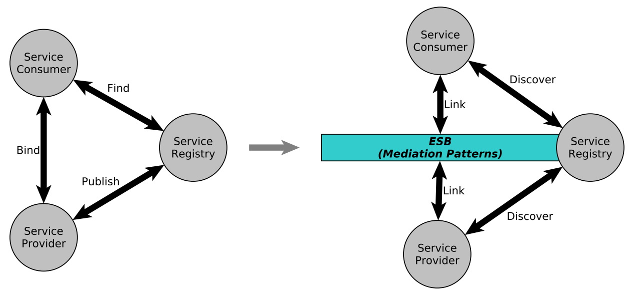


Figure 1. ESB representation in a SOA (adapted from [1]).

that implements the concepts of those functionalities. Several ESB applications are available, from proprietary vendors to open-source solutions[1]. In this work, we adopt the open-source version of Mule-ESB[2], which is a lightweight Java-based ESB that includes much functionalityto integrate existing systems.

III. Event-based Testing of Web Service Compositions

This section presents our approach for testing service compositions using event sequence graphs, whereby a "running example" illustrates the approach. Moreover, we introduce processes and algorithms to generate test cases and run tests.

A. Running Example

The business process to grant loans called `xLoan`, proposed in [8], is the example we use to explain and illustrate the approach. Note that this example is not the case study that includes a non-trivial, commercial web application in Section IV.

The example involves three services: `LoanService` (`LS`), `BankService` (`BS`), and `BlackList-InformationService` (`BLIS`). The `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 options. 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.

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

[1]http://en.wikipedia.org/wiki/Enterprise_service_bus
[2]http://www.mulesoft.org/

B. Testing Web Service Compositions with ESGs

In previous work, we proposed ESGs for event-based modeling and testing of graphical user interfaces [12] as well as (stateful) web services [13]. This paper extends the ESG notion to consider events of WSCs. This new modeling approach supports automatic generation of test cases for detecting faults in communication and in data selection, which have also the potential to violate time constraints.

Decision tables (*DT*) are popular in information processing and are also traditionally used for testing, e.g., in cause and effect graphs [17]. A DT logically links constraints ("if") with events ("then") that are to be triggered, depending on combinations of constraints ("rules"). Table I presents a rudimentary DT for the operation checkBL. DTs are a powerful means for:

- handling sequences of events which depend on constraints; and
- refining data modeling of calls to invoked services [13].

Table I
A Decision Table with 3 Rules, 2 Constraints and 3 Events.

		Rules		
	checkBL(uniqueID)	R1	R2	R3
Constr.	**uniqueID is valid**	T	T	F
	uniqueID in Blacklist	T	F	-
Events	inBList < 200ms	X		
	notinBList < 100ms		X	
	SOAPFault - invalid identification			X

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

- $C \neq \emptyset$ is the set of constraints (conditions), that can be evaluated as true or false,
- $E \neq \emptyset$ is the set of events, and
- $R \neq \emptyset$ is the 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 $R_i = (C_{True}, C_{False}, E_x)$ where:

- $C_{True} \subseteq C$ is the set of constraints that have to be evaluated as true,
- $C_{False} \subseteq C \backslash C_{True}$ is the set of constraints that have to be evaluated as false, and
- $E_x \subseteq E$ is the set of events that should be executable if all constraints $t \in C_{True}$ are resolved to true and all constraints $f \in C_{False}$ are resolved to false.

Note that $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$ under regular circumstances. In certain cases it is inevitable to have constraints with a *don't care* (noted as '-' in a DT), i.e., such a constraint is not considered in a rule and $C_{True} \cup C_{False} \subset C$. We use DTs to refine input parameter of invoked services. The example in Table I models the invoking process of checkBL of `BlackListInformationService`. Constraints in boldface model possible domains of input parameters and constraints in italics represent additional constraints to that input data. If constraints are evaluated according to R1 of Table I (i.e., if R1 is true), a response inBList is expected. Moreover, the events of Table I can be extended by time constraints whenever a response is expected within a certain time range.

DTs are useful to describe constraints, but they are not appropriate for describing WSC interactions. Hence, we extend the ESG notion and combine it with DTs in order to consider additional aspects like communication, parallel flow and conditional activities.

Definition 3. An *event sequence graph for web service compositions* $ESG4WSC = (V, E, M, R, DT, f, \Xi, \Gamma)$ is a directed graph, where:

- $V \neq \emptyset$ is a finite set of vertices (representing events),
- $E \subseteq V \times V$ is a finite set of arcs (edges),
- M is the 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 the set of DTs that refine some 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 decision tables, respectively.

Definition 4. Let V be defined as in Definition 3. Then, $V \neq \emptyset$ is a finite set of vertices (representing events) with $V = V_e \cup V_{refined} \cup V_{req} \cup V_{resp}$, such that $V_e, V_{refined}, V_{req}, V_{resp}$ are pairwise disjoint and where

- V_e is a set of generic events,
- $V_{refined} = \{v \in V | \exists m \in M \wedge (v, m) \in R\}$ is a set of vertices refined by one or more ESG4WSCs. A refinement with more than one ESG4WSC represents behavior running in parallel,
- V_{req} is a set of vertices modeling a request to its own interface/operations (public) or an invoked service (private),
- 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, $DT = DT_{seq} \cup DT_{input}$ is the set of DTs that refine some events. There are two types of DTs, a decision table $dt \in DT_{seq}$ models the execution restrictions for following events and a decision table $dt \in DT_{input}$ models constraints for input parameter of operations.

Since WSCs always initiate with one or more request events, the set Ξ contains only vertices $v \in V_{req}$. 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 a vertex $v \in V$ has more than one successor, the vertex v is to be refined by a DT.

The semantics of an ESG4WSC is as follows: Any $v \in V$ represents an event, such as a request or a response which 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 done by a consumer or, in our case, 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 which represent invoked services of the WSC. They are usually not observable by a consumer; however, we assume 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 6. Figure 2 represents an ESG4WSC for the `xLoan`. The operations checkBL and inDebtorsList are executed concurrently. Requests are represented by light gray vertices; responses are represented by dark gray vertices. Vertices with a bold edge represent public requests and responses. Vertices refined by DTs are double-circled. The corresponding ESG4WSC looks like this:

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

- $V_e = \{Timeout > 2h\}$,
- $V_{refined} = \{check\}$,
- $V_{req} = \{LS : requestLoan,\ BS : approveBank,$ $BS : offer,\ LS : cancel,\ LS : SelectOffer,$ $BS : cancelBank,\ BS : confirmBank\}$
- $V_{resp} = \{BS : approved,\ BS : Notapproved,$

$LS : notAprovedMSG$, $BS : Offers$,
$LS : replyOffers$, $LS : wrongOffer$,
$LS : replySelect\}$

- $E = \{(LS : requestLoan,$
  $BS : approveBank), \ldots\}$,
- $M = \{M_{BLIS}, M_{CAS}\}$,
- $R = \{(check, M_{BLIS}), (check, M_{CAS})\}$,
- $DT = DT_{seq} \cup DT_{input} = \{dt_{check}\} \cup$
  $\{dt_{LS:requestLoan}, dt_{BS:approveBank}, \ldots\}$,
- $f(check) = dt_{check}$,
  $f(LS : requestLoan) = dt_{LS:requestLoan}$,
  $f(BS : approveBank) = dt_{BS:approveBank}, \ldots$
- $\Xi = \{LS : requestLoan\}$,
- $\Gamma = \{LS : notAprovedMSG,\ BS : cancelBank,$
  $LS : replySelect\}$

Definition 7. Let V, 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 $i = 0, \ldots, k-1$. 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 $v_k \in \Gamma$.

Definition 8. Two events or ESs a and b that are to be executed in parallel are denoted as $a||b$. The operator $||$ is commutative, i.e., $a||b = b||a$, and associative, i.e., $(a||b)\,||c = a||\,(b||c)$.

Example 9. For the `xLoan` example given in Figure 2, the services `CAS` and `BLIS` are to be executed in parallel, e.g., following sequence might hold:

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

1) Fault Model and the Overall Test Process: A CES 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, following faults might occur during the execution:

- there are calls to services which are *not defined* in the CES
- there are *missing* calls to services which are defined in the CES
- the *sequence* of calls is different compared to the given CES
- the *parameter* of calls to the invoked services do not correspond to the expected ones

In order to provoke and control a specific CES of the WSC, it is often inevitable to take control of the invoked services since they communicate with the SUT and the flow of the WSC might depend on a returned response. In Section IV-C, we present the *ESB-based Mediator Service* to cope with this technical issue during the test execution. The modeled constraints of DTs enable to validate the passed data. 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 has 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 SUT and especially the expected responses of operation calls. Thus, we first have to generate CESs that describe an execution of the SUT. The overall test process consists of the following steps:

1) Generate CESs out of 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 SUT with the different input data generated. During the execution,
    a) observe calls to invoked services during execution
    b) send back the expected response (if necessary) for the invoked service according to the CES

2) Test Case Generation: In order to reveal possible faults in the WSC, the generated test suite must cover at least all EPs. Moreover, the cost should be minimal, i.e., CESs generated to cover all EPs have a minimal total length. In this context, the CES-generation represents the *Chinese Postman Problem* (CPP) solution of which is expected to generate a minimal spanning set of CESs (see [18]). The algorithm for deriving CESs from an ESG4WSC can be described in following steps:

1) Generate CESs for the refined vertices first (recursive call)
2) Add multiple edges 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
        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)
3) Generate CESs according to the CPP
4) Replace refined vertices in the resulting CESs with respect to their allowed successors

Note that Step 2 adds multiple edges to the underlying ESG4WSC, that is, every edge represents a CES of the refined vertex to be combined later on with the resulting CES set. The benefit of this approach is that the resulting CES set derived in Step 3 contains the vertex as much as needed and is minimal. Algorithm 1 along with Table III gives a detailed description of the CES generation process. 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

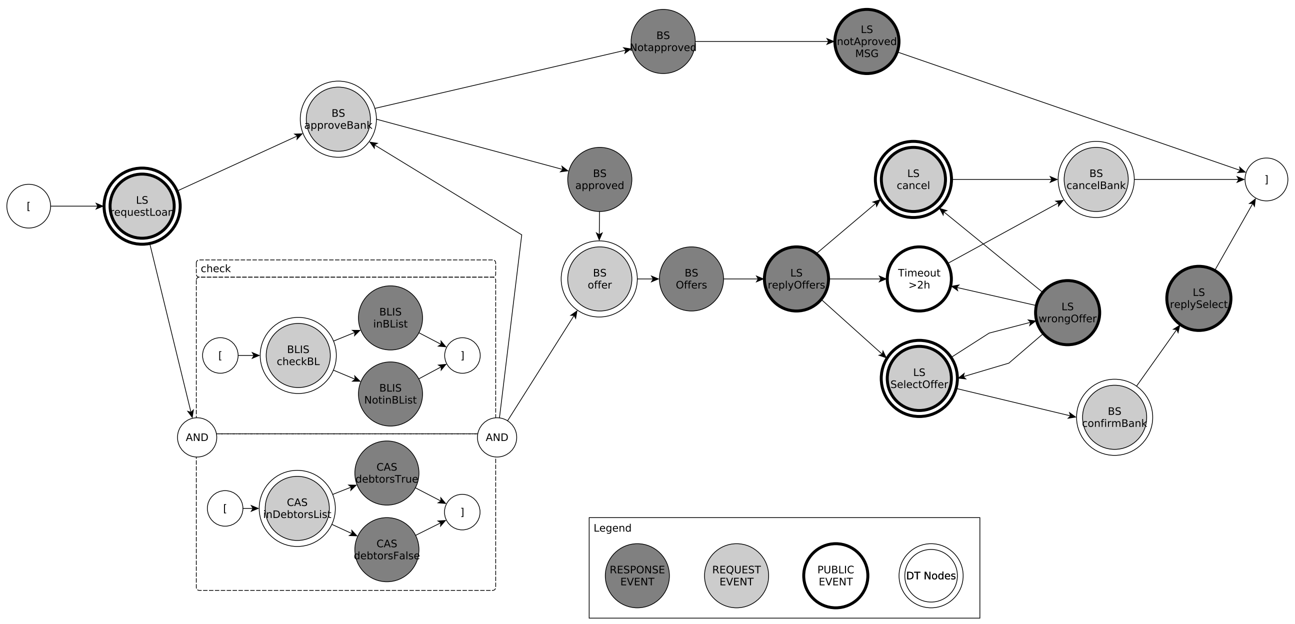


Figure 2. ESG4WSC for the `xLoan` example.

call. If a constraint set cannot be fulfilled, CESs can be deleted, e.g., where two contradicting constraints are to be satisfied.

Example 10. According to Figure 2, the test generation process looks as follows:

Step 1: generate CESs for refined vertices

The following sequences for the 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

Add following edges according to Table II:

- 3 edges $(check, BS : approveBank)$ for sequences S1 to S3
- 1 edge $(check, BS : offer)$ for sequence S4

Step 3: generate CESs

The following CESs have been generated (refined vertices 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				X
BS:approveBank	X	X	X	

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, Timout > 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, Timout > 2h, BS:cancelBank`⟩

Step 4: replace refined vertices by sequences of Step 1:

```
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, Timout >
       2h, BS:cancelBank⟩
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,
       Timout > 2h, BS:cancelBank⟩
```

The generation of data out of the initial DT leads to the *Constraint Satisfaction Problem* (*CSP*). CSP is defined by a set of variables X_1, X_2,...,X_n and a set of constraints, C_1,C_2,...,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 [19]). Each rule of the DT under consideration represents a CSP.

IV. Case Study

This section describes the evaluation of the approach ESG4WSC. First, we describe the SUT used in the case study, as well as the obtained results. We present some prototype tools we used during the case study. Finally, discussion and limitations are presented.

A. System under Test

The case study was conducted using the composite service *Travel Agent* that provides a set of facilities to query and book a trip. The *Travel Agent Service* interacts with two services, ISELTA-hotel and Airlines service. ISELTA-hotel 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 [13]. It provides operations to query hotels and manage bookings. The Airlines service provides a similar set of operations, but to manage flight tickets. The *Travel Agent Service* combines these two services, providing operations to search and book a travel involving flight and hotel reservation.

Table III
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	$\langle a,b,c\rangle\,[2]=b$
$s[i..j]$	the sequence from i to j	$\langle a,b,c\rangle\,[1..2]=\langle a,b\rangle$
$s\|\|t$	sequence s in parallel to sequence t	
$s\oplus t$	concatenate s and t	$\langle a,b\rangle\oplus\langle c\rangle=\langle a,b,c\rangle$

A summarized description of the *Travel Agent Service* is presented as follows. Its interface provides three operations: queryTrip, getAllOptions, and book. The *Travel Agent* workflow can be performed with the following steps:

1) queryTrip is invoked with the search data.
    a) If there is some invalid data, TripInputException is launched and the process finishes (see Step 7).
2) The search data is mapped to the search operations for ISELTA-hotel and Airlines services.
3) Both search operations are called concurrently.
    a) HotelService requires a login operation before the search.
    b) If one of the services (ISELTA, Airlines) launches an exception, TripInputException is launched and the process finishes (see Step 7).
4) Check if at least one hotel and flight was returned. Otherwise it launches a TripBookingException with a message showing that there is no hotel or flight or both.
5) Generate a search code and return a response for the operation queryTrip with the five cheapest prices for flights and hotels.
6) At this point, the process will wait for 3 events, book or getAllOptions calls or a timeout (5 min).
    a) After 5 minutes 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

Algorithm 1. generateCESs()

```
function  : generateCESs()
input     : an ESG4WSC
output    : CES
1  foreach $re \in V_{refined}$ do
       // step 1
2      $resCES \leftarrow \emptyset$;
3      foreach $esg \in re$ do
4          $CES = generateCESs(esg)$;
5          if $resCES == \emptyset$ then
6              $resCES = resCES \cup (re \times CES)$;
7          else
8              foreach $ces_1 \in \{ces|(re, ces) \in resCES\}$ do
9                  foreach $ces_2 \in CES$ do
10                     $resCES = resCES \cup (re, (ces_1||ces_2))$;
11                 $resCES = resCES \setminus (re, ces_1)$;
       // step 2
12     if $f(re) \neq \varepsilon$ then
13         $\mathrm{DT}_{seq} = f(re)$;
14         foreach $ces \in \{ces|(re, ces) \in resCES\}$ do
15             $v = getAllowedSuccessor(ces, DT_{seq})$;
16             $E = E \cup (re, v)$;
               // store a Mapping, i.e., $Map \subseteq E \times ces$
17             $Map = Map \cup ((re, v), ces)$;
18             $resCES = resCES \setminus (re, ces)$;
       // add multiple edges for each dataset to be tested
19     foreach $DT_{input,public} \in V$ do
20         foreach $a \in A$ do
21             E=E$\setminus(DT_{input}, a)$;
22         foreach $(C_{true}, C_{false}, E_x) \in R$ do
23             E=E$\cup(DT_{input}, E_x)$;
       // step 3
24     $CES = solveCPP(ESG4WSC)$;
       // step 4
25     foreach $ves \in CES$ do
26         for $i = 1$ to $\#ces$ do
27             if $ces[i] \in V_{refined}$ then
28                 if $|\{ces|((ces[i], ces[i+1]), ces) \in Map]\}| > 0$ then
29                     $new = es$ with $es \in \{ces|((ces[i], ces[i+1]), ces) \in Map]\}$;
30                     $Map = Map \setminus ((ces[i], ces[i+1]), ces)$;
31                     $ces = ces[1..(i-1)] \oplus new \oplus ces[(i+1)..\#ces]$;
32                 else if $|\{ces|(ces[i], ces) \in resCES\}| > 0$ then
33                     $new = es$ with $es \in \{ces|(ces[i], ces) \in resCES]\}$;
34                     $resCES = resCES \setminus (ces[i], ces)$;
35                     $ces = ces[1..(i-1)] \oplus new \oplus ces[(i+1)..\#ces]$;
36 return CES;
```

(process finishes or does not exist), TripInputException is launched and the timeout is restarted.

c) book request.

i) Launch TripInputException if some input data is invalid or the process is finished.

ii) If all input data is correct, the bookings for hotel and flight are called concurrently and successfully, a booking message is returned. The process finishes (see Step 7).

A) for booking a hotel, login and search operations are required first, since there is a timeout of 30 seconds. If the search does not contain the same combination of hotel and price to be booked, a TripBookingException is launched.

iii) If some fault happens during the hotel and flight booking, TripBookingException is launched with the problem description. The consumer can try again 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 cancellation fails, a TripBookingException is launched 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).

B. Configuration and Results

We applied our approach to test the *Travel Agent Service*. 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 a tester that created the ESG4WSC model and DTs for the *Travel Agent service*. The tester had no access to the source code in order to avoid mixing modeling strategies (our approach is black-box). Test cases were derived according to the ESG4WSC approach. The tester also performed the concretization and execution of the test cases.

The ESG4WSC model for *Travel Agent Service* has 46 events and 17 DT nodes. It contains two $V_{refined}$ nodes with two ESG4WSCs in each one. The effort to study the specification and interfaces plus the ESG4WSC modeling was around eight hours. The test generation algorithm proposed in Section III derived 25 test cases from the *Travel Agent Service* ESG4WSC. The tester started to concretize and execute the test cases. When a fault was revealed, the tester stopped the tests and provided the failed test case to the developer. The developer corrected the fault and the tester resumed the tests. After ten iterations, 14 faults were identified, ten in the SUT and four in the specification. The faults in the SUT were mainly identified by testing different rules (from the DTs) and checking expected events. The correction for these faults was not critical and was performed only in the SUT. The faults in the specification were identified by checking expected event sequences. The four faults were related to some behavior not described in the specification and observed during the test case execution. This type of fault is more critical since the tester needed to modify the ESG4WSC and regenerated the test cases. Moreover, the specification and, consequently, the SUT were also corrected.

C. Exemplifying the Test Steps and Tool Support

We developed and used some tools that support the proposed test case generation method and the test execution.

1) Test Generation: Test generation was supported by a tool called *Test Suite Designer* (TSD). Figure 3 shows a screenshot of TSD and the model set up for the case study. TSD implements a solution of the CPP and enables the coverage of a single model. Thus, step three, forming the most important step of the test generation process, is automatically executed; for large models it can hardly be done by hand. Furthermore, TSD allows generating test data out of DTs (see Figure 4). For test generation, steps described in Section III are followed. Steps two and four had to be done by hand. Steps one and three are (partially) automated. For the future, we plan to integrate the proposed Algorithm 1 into TSD.

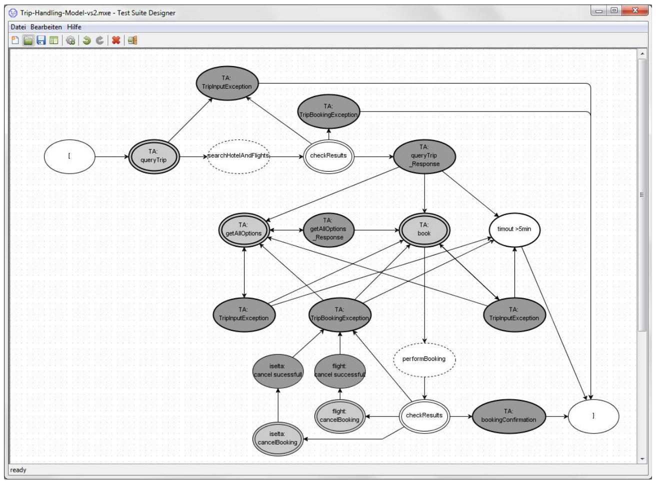
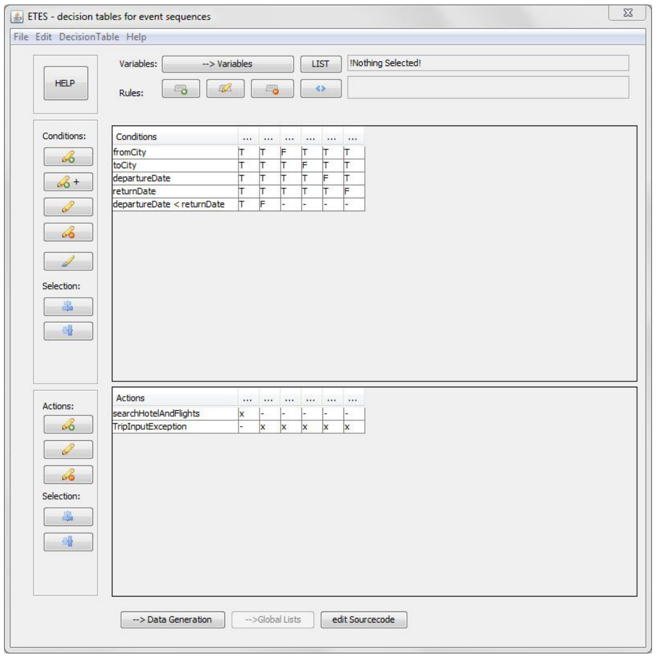


Figure 3. ESG4WSC in Test Suite Designer.



Conditions	...	...	...	...	...	...
fromCity	T	T	F	T	T	T
toCity	T	T	T	F	T	T
departureDate	T	T	T	T	F	T
returnDate	T	T	T	T	T	F
departureDate < returnDate	T	F	-	-	-	-

Actions	...	...	...	...	...	...
searchHotelAndFlights	x	-	-	-	-	-
TripInputException	-	x	x	x	x	x

Figure 4. Decision Tables in Test Suite Designer.

2) Test Execution: We use mule-ESB as the infrastructure software. Initially, all services involved in the composition (including the composite service) are deployed in the ESB, i.e., the entire communication (SOAP messages) passes through the ESB before reaching the destination service. We used an ESB-based mediator service to support the test execution. The mediator is composed of two parts, a web service and an ESB component. The web service contains three main operations:

- `startObservation`: prepares the test execution and identifies the start of a new test case. The services whose messages will be modified are passed as parameters.
- `modifyMessage`: sets what must be the response message for a certain service. This operation is useful to force an event and test a specific scenario.
- `getAllMessages`: retrieves all messages that pass through the ESB after the last *startObservation* calling.

This service can be used directly in the test code (as shown in Figure 6). The second part of the mediator is the ESB component that implements the monitor and aggregator patterns. This component is integrated with mule-ESB and is able to interact with the mediator service. The component records all messages that pass through the ESB and also modify messages according to the service operation `modifyMessage`. Figure 5 summarizes the architecture adopted to execute the tests in this case study.

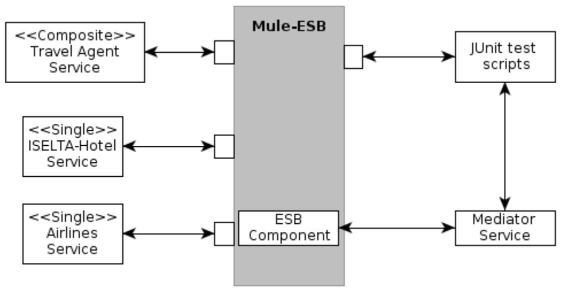

Mule-ESB
<<Composite>> Travel Agent Service
<<Single>> ISELTA-Hotel Service
<<Single>> Airlines Service
ESB Component
JUnit test scripts
Mediator Service

Figure 5. Architecture to execute the tests.

The test cases generated using the ESG4WSC approach were implemented using Java and JUnit. Figure 6 shows how the implementation of a test sequence using JUnit looks like. The object `mediatorService` refers to a stub class to access the Mediator Service; the object `travelAgentService` refers to a stub class to access the WSC under test; the object `suptest` provides some methods to support the tests; and the object `eventChecker` provides methods to check whether the messages correspond to certain events. Lines 6-9 starts the observation using the Mediator service. Lines 11-12 set up the ISELTA-Hotel and Airlines services to answer the specified messages for the first request. Lines 14-25 set up and call the controlled operation `queryTrip` and check the expected output. Finally, Lines 28-33 verify the messages (events) that passed through the ESB, checking their order. Lines 31-32 check a partial order for concurrent events. For example, the event `is:login` should happen before the event `is:soapFault`, but they can interleave with events `fl:search` and `fl:results>=1`.

D. Discussion of Results and Limitations of the Approach

The previous sub-sections stepwise described how to carry out our approach. The case study clearly demonstrated that our test approach is applicable to a non-trivial web-based application. It was able to reveal numerous faults, not only in in the SUT but also in the specification. By applying the ESG4WSC approach, the tester observed failures related to the communication. In the case study, faults related to missing and unexpected messages have been detected for both, specification and SUT. Thus, the approach helps to keep implementation and specification synchronized. Moreover, the specification has been set up first and the implementation and test model creation has been done in parallel by two different persons. Hence, the test engineer does not need to wait for the implementation to set up a model and derive tests since our approach is not based on artifacts like BPEL or WS-CDL as in [5], [7], [8].

We did not distinguish between orchestration and choreography since our approach can be applied in both contexts. The only restriction is that 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, involving setting up some configuration files.

We limited our approach by assuming that ESG4WSCs in a refined event are independent. Thus, we provide a simple way to model some parallelism and, for the example and case study, this was sufficient. It is possible that more complex scenarios can occur in service compositions and the tester might also want to test combinations of message interleaving. Although our approach can be adapted to test these scenarios, we recommend specific models and testing techniques for concurrent programs [20], [21].

In the case study, we used some tools that we have specifically developed to deploy our approach. However, a fully-developed and integrated environment to support our approach was not available. Nevertheless, we believe that most of the JUnit code can be generated automatically, reducing the cost of test concretization.

V. Related Work

Some related work that is necessary for explaining the approach has already been discussed in Section 4. The present section extends the related work referring to further relevant works and compares them with the approach introduced in this paper.

SOA testing has been studied intensively in the last years [4], [22], with a particular effort on formal testing approaches (see [23]). Following, we review research that use models to test service compositions.

```
//Complete Event Sequence 02
//[ TA:queryTrip, [ [is:login, is:SOAPFault] || [fl:search, fl:results>=1] ], TA:TripInputException ]
@Test
public void testCase_02_01()
{
  ArrayList<String> services = new ArrayList<String>();
  services.add("iselta");
  services.add("Airlines");
  mediatorService.startObservation(services);

  mediatorService.modifyMessage("iselta", 1, suptest.loadMessage("iselta-login-fault.xml"));
  mediatorService.modifyMessage("Airlines", 1, suptest.loadMessage("airline-search-greater01.xml"));

  TripSearchData tripSearchData = new TripSearchData();
  tripSearchData.setFromCity("Paderborn");
  tripSearchData.setToCity("Sao Carlos");
  tripSearchData.setDepartureDate( suptest.toXMLGregorian(2011, 07, 13) );
  tripSearchData.setReturnDate( suptest.toXMLGregorian(2011, 07, 19) );

  try {
    travelAgentService.queryTrip( tripSearchData );
    fail();
  } catch (Exception e) {
    assertTrue(e instanceof TripInputException_Exception);
  }

  //check messages
  ArrayList<String> messages = mediatorService.getAllMessages();
  assertEquals(6, messages.size());
  assertTrue( eventChecker.isTA_queryTrip( messages.get(0)) );
  assertTrue( suptest.checkOrder(eventChecker, messages, 1, 4, {"IS_login","IS_soapFault"}) );
  assertTrue( suptest.checkOrder(eventChecker, messages, 1, 4, {"FL_search", "FL_results"}) );
  assertTrue( eventChecker.isTA_TripInputException( messages.get(5)) );
}
```

Figure 6. JUnit code for implementing a test sequence.

Benharref et al. [6] propose a multi-observer architecture to detect and locate faults in composite web services. The proposed architecture is composed of 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, based on collecting and analyzing traces. The concept of observers is similar to the mediator service and can also be implemented using an ESB. However, our approach aims to actively test the service composition. A strategy similar to Benharref et al. can be implemented using ESG4WSCs and the Mediator service.

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 (XRGs). Based on LTS, the LTS-based Choreography model (C-LTS) is proposed with XRGs attached in transitions that represent service invocations. New types of definition-use associations are proposed and test adequacy criteria are presented. Our work is focused on test case generation, which is not approached by Mei et al. Thus, both approaches are complementary since the coverage criteria can be used to give more information about the test suite generated by our 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] present a mapping from BPEL to STS. Test cases are generated using symbolic execution, and applied to the SUT using online testing. Hou et al. [7] model a BPEL program using message sequence graphs and generate message sequences. Our work differs from [7], [8] concerning the available artifacts. We do not assume that the composition was developed using BPEL and the tester has access to this artifact. Although our approach requires more effort to develop the test model, the SUT 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 [10], 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. Wieczorek et al. work is close to our approach with two main differences: (i) We rely on the ESG4WSCs that are more abstract models compared to MCMs which form a domain-specific language created to design service choreography. (ii) Our approach also performs the tests using the ESB-based mediator service, which can be adapted to MCM approach.

VI. Conclusion and Future Work

In this paper, we presented an event-oriented approach, named ESG4WSC (ESG for Web Service Composition), for model-based testing (MBT) of web service compositions. We also conducted a case study and described the Enterprise

Service Bus-based Mediator service to support the test execution.

ESG4WSC brings the benefits of MBT to web service compositions. 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 case study results give some evidence that our approach scales well (25 test cases were sufficient to detect 14 faults). Moreover, we have been able to detect faults not only in the SUT, but also in the specification for a non-trivial service-oriented application.

The approach can broadly be applied, without any constraints. However, its strength stems from its potential to fit well for cases where BPEL or WS-CDL specifications are not available. Thus, it allows to simultaneously performing the steps for implementation and testing of the SUT. Other approaches strictly require the availability of the artifacts, such as BPEL codes, WS-CDL specifications, etc. Moreover, our experiments based on the prototype test tools convince of the feasibility of the approach automation.

Encouraged by the successful application of our approach to testing of a commercial project, we plan to extend our work to include unexpected behavior of the SUT (*negative testing*). Additionally, we investigate how to evolve ESG4WSCs to a state machine to consider not only events, but to take also states into account and to cope with the complex behavior of SUTs in the practice. This requires a considerably increase of the automation power of our tools, which are also subjects of our present research. The automation will also help with evaluating the approach in a larger case study.

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).

References

[1] M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen, "The enterprise service bus: making service-oriented architecture real," *IBM Systems Journal*, vol. 44, pp. 781–797, 2005.

[2] N. Josuttis, *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[3] M. P. Papazoglou and W.-J. Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The International Journal on Very Large Databases (VLDB)*, vol. 16, no. 3, pp. 389–415, July 2007.

[4] G. Canfora and M. Di Penta, "Service-oriented architectures testing: A survey," in *Software Engineering: International Summer Schools (ISSSE)*, 2009, pp. 78–105.

[5] L. Mei, W. K. Chan, and T. H. Tse, "Data flow testing of service choreography," in *Symposium on the Foundations of Software Engineering (FSE)*, 2009, pp. 151–160.

[6] A. Benharref, R. Dssouli, R. Glitho, and M. A. Serhani, "Towards the testing of composed web services in 3rd generation networks," in *TESTCOM*, 2006, pp. 118–133.

[7] S.-S. Hou, L. Zhang, Q. Lan, H. Mei, and J.-S. Sun, "Generating effective test sequences for BPEL testing," in *International Conference on Quality Software (QSIC)*, 2009, pp. 331–340.

[8] L. Bentakouk, P. Poizat, and F. Zaïdi, "A formal framework for service orchestration testing based on symbolic transition systems," in *International Conference on Testing of Software and Communication Systems (TESTCOM)*, 2009, pp. 16–32.

[9] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker, "Applying model checking to generate model-based integration tests from choreography models," in *TESTCOM*, 2009, pp. 179–194.

[10] S. Wieczorek, A. Stefanescu, and A. Roth, "Model-driven service integration testing - a case study," in *International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2010, pp. 292–297.

[11] W. Grieskamp, N. Kicillof, K. Stobie, and V. A. Braberman, "Model-based quality assurance of protocol documentation: tools and methodology," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 55–71, 2011.

[12] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study," *Software Testing, Verification & Reliability*, vol. 16, no. 1, pp. 3–32, 2006.

[13] F. Belli and M. Linschulte, "Event-driven modeling and testing of real-time web services," *Service Oriented Computing and Applications Journal*, vol. 4, no. 1, pp. 3–15, 2010.

[14] D. Jordan et al., "OASIS web services business process execution language (WSBPEL) v2.0," 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/

[15] N. Kavantzas et al., "Web services choreography description language version 1.0," 2005. [Online]. Available: http://www.w3.org/TR/ws-cdl-10/

[16] A. Arkin et al., "Web service choreography interface (WSCI) 1.0," 2002. [Online]. Available: http://www.w3.org/TR/wsci/

[17] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, *The Art of Software Testing*. John Wiley & Sons, 2004.

[18] F. Belli and C. J. Budnik, "Minimal spanning set for coverage testing of interactive systems," in *International Colloquium on Theoretical Aspects and Computing (ICTAC)*, 2004, pp. 220–234.

[19] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 2003, ch. Constraints Satisfaction Problems, pp. 137–160.

[20] C. Hoare, *Communicating Sequential Processes*, 2004.

[21] Y. Lei and R. Carver, "Reachability testing of concurrent programs," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 382 –403, june 2006.

[22] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing web services: A survey," Department of Computer Science, King's College London, Tech. Rep. TR-10-01, January 2010.

[23] A. T. Endo and A. S. Simao, "A systematic review on formal testing approaches for web services," in *Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, 2010, pp. 89–98.

[24] F. Belli, N. Güler, and M. Linschulte, "Are longer test sequences always better? - a reliability theoretical analysis," in *Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, 2010, pp. 78–85.

[25] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2006.