This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Belli, F., Linschulte, M., Event-Driven Modeling And Testing of Real-Time Web Services, Journal of Service Oriented Computing And Applications, Vol. 4, No. 1, pp. 3-15, Springer London

Event-driven modeling and testing of real-time web services

Fevzi Belli · Michael Linschulte

Abstract A service-oriented architecture (SOA) for web applications is often implemented using web services (WSs) and consists of different operations the executions of which are perceived as events. The order and time-appropriateness of occurrences of these events play a vital role for the proper functioning of a real-time SOA. This paper presents an event-based approach to modeling and testing of functional behavior of WSs by event sequence graphs (ESG). Nodes of an ESG represent events, e.g., "request" or "response", and arcs give the sequence of these events. For representing parameter values, e.g., for time-out of operation calls, ESG are augmented by decision tables. A case study carried out on a commercial web system with SOA validates the approach and analyzes its characteristic issues. The novelty of the approach stems from (i) its simplicity and lucidity in representing complex real-time web applications based on WSs in SOA, and (ii) its modeling that considers also testing and thus enables a comfortable fault management leading to a holistic view.

Keywords Web services · Real-time · Event sequence graph · Model-based testing

1 Introduction and related work

Service-oriented architecture (SOA) is one of the current buzzwords in information technology. SOA allows enterprises to centralize computer-based services and to offer those services over a network. On the basis of a published interface, the service can be used platform-independent inside and outside of the enterprise. The concept of a SOA is often realized by *web services* (WS). WSs are applications that provide a set of operations accessed by other applications through the Internet. An operation of the WS can also be called as "atomic web service". Moreover, an atomic WS can consist of other atomic web services ("composite web service"). The interface of a WS and its correpsonding operations is usually described by WSDL (Web Service Description Language) in accordance with SOAP (Simple Object Access Protocol), which is used as communication protocol. Both standards, SOAP and WSDL, are based on XML.

Interactions with WSs consist of requests and responses, which are perceived by the environment (human users or other systems) as *events*. *Requests* are similar to traditional function calls, but to a WS via network; *responses* are corresponding answers, i.e., reactions of the called WS. In spite of its functional correctness, a request can be useless if it is not delivered "in time", e.g., a request for a stock price is of no interest and thus useless if it is not delivered before the next change. Taking time constraints into account requires time monitoring from request to response; in other words, we need a *real-time* service-oriented architecture (*RTSOA*). Time can be measured either server-sided or client-sided. The time difference between client and server time primarily depends on SOAP calls, which entail intense XML parsing. Also, the size of a response plays an important role: a search operation, triggered by a single query, can return huge amounts of entries.

WSs can roughly be grouped into *stateless* and *stateful* WSs. The view of stateless WSs is that the corresponding web server does not save any context information. All of the context information needs to be included in the WS request. In contrast to this, within stateful WSs context information is

F. Belli · M. Linschulte (✉)
Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, Paderborn, Germany
e-mail: linschu@upb.de

F. Belli
e-mail: belli@upb.de

stored by the web server such that execution of one operation may influence execution and/or result of another operation. Therefore, context information is built up and controlled by the server and not by the client.

This paper presents an event-based approach for modeling and testing the functional behavior of *stateful* WSs where sequences of corresponding operation calls play a vital role. Operations of a WS are modeled by *event sequence graphs* (ESG) where each node represents an event, i.e., request or response. A model is usually used for analyzing the specified functional behavior of the *system under consideration* (SUC), thus for testing whether or not SUC is doing the right things (testing the *desirable* behavior, or *positive testing*). Our ESG view will holistically allow us also the other way around, thus to check whether or not SUC is doing the incorrect, illegal things (testing the *undesirable* behavior, or *negative testing*). This is what we meant "a comfortable fault management" in the abstract. The approach does not, however, consider dynamic binding of web services. This is always difficult to be handled as during design-time cooperating WSs are not determinable and during development time not testable. Therefore, we focus on the given operations of a stateful WS that is to be tested as a black-box. Nevertheless, similar to [6], a model can be defined for (discovered or even undiscovered) cooperating WSs. Generated test cases can be used either to discover a suitable WS or to discover changes made to the WS implementation by periodically running the test cases. Periodically testing of services should be done especially for foreign inbound services as new features might have been added that influence the characteristics of the SUC, e.g., its response time or functional behavior.

Contracts [10] (as introduced in Design by Contract[1]) describing the structure of input data are useful for WSs, but the WSDL-based specification of a WS describes only the syntax of an interface. Thus, there is a lack of semantic information, e.g., a hotel reservation system will need an arrival date and a departure date. It is obvious that the arrival date should lie before departure date. Those additional constraints are not defined in a WSDL-specification. This is the reason why we model the parameters of each operation and their interrelationships, especially the ones concerning time constraints, by *decision tables* (DT) [9].

[1] Design by Contract (DbC) is an object-oriented design technique that was first introduced by Meyer in 1992 [16]. Similar to a legal contract, which is an agreement between a costumer and a supplier about the buy or rent of a service or product and that specifies some conditions to protect costumer and supplier; in DbC, pre-conditions and post-conditions are set for using a piece of software, e.g., an operation. Pre-conditions have to be fulfilled before an operation is executed; post-conditions have to be fulfilled after an operation was executed. The conditions themselves are Boolean expressions, thus they allow usage of the conditions for testing.

The simple and lucid, nevertheless powerful representation capability of DT-augmented ESG, together with their *holistic* testing concept that integrates positive and negative testing of WS-based web applications in RTSOA is the novelty of the approach introduced in this paper. The next chapter discusses the related work. Section 3 introduces the methodology of our approach using a non-trivial example borrowed from the case study of Sect. 4, which does not only validate the approach, but also exploits its characteristic issues. Section 5 concludes the paper, summarizing the approach introduced and future work planned.

2 Comparison with related work

A great deal of research work is dedicated to SOA [5] and some of them to analysis and testing of WSs, as summarized by Baresi et al. [3]. Most of enclosed work deals with composition of WSs and communication among WSs. The reason is the unlimited number of WSs that can be integrated into one application in which a WS itself can even integrate other WSs. An extension to the WS interaction model is the web services Business Process Execution Language (BPEL/WSBPEL). Its goal is to enable the orchestration of WSs, which support business transactions [11]. In this context, Marconi et al. [14] presented a semi-automated development process to support the composition task that is able to generate the appropriate WSDL- and BPEL-Files.

We already mentioned in Sect. 1 that there exists a lack of semantic information in WSDL-Files that complicates black-box testing of WSs (see also [10, 12], etc). Tsai et al. [21] extended WSDL to support testing of WSs that incorporates sequence specifications as well as input–output dependencies. Unfortunately, also this extended WSDL does not consider dependencies among input parameters. There exist few approaches in literature dealing with systematic model-based testing of a single WS. Close to our approach, but not event-based, is stateful WS Testing [8] based on *Stream X-machines*, which is able to generate and execute test cases. Another state-based testing approach is presented by Keum et al. [12]. On the basis of extended finite state machines, test cases for stateful WSs are generated and executed. However, real-time aspects are not considered. Also negative testing is not considered by both approaches. As far as we know, negative testing was not subject of similar approaches. Apart from these novelties, our approach concentrates on observable events as, especially in black-box testing, it is often difficult, even infeasible, to identify specific kinds of states.

WS-Policy [22] enables to set guidelines for a WS concerning security or Quality of Service. Therefore, real-time aspects could be described by WS-Policy as presented by Ortiz et al. [18]. Mathes et al. [15] described an extension of the WS-Policy standard by temporal aspects. The so-called

WS-Temporal Policy enables a service developer to attach a validity period to the properties described in a WS-Policy, but does not provide the ability to define time constraints for a WS (the title can be misleading).

Heckel et al. [10] present a contract-based approach for black-box testing of WSs. Contracts are modeled by an UML-based notion to facilitate the visualization of contracts. The visualized contracts are then used to construct test cases. Unfortunately, the paper does not describe how test cases and especially test data are generated and executed or if time constraints can be taken into account.

Event-based modeling has gained some attention during the last years. In 2003, Roy Schulte, a Gartner Analyst, declared *event-driven architecture* (EDA) as successor of SOA [17]. EDA promises a reaction to events "when they happen": Whereas a SOA is a "pull" architecture, applications in EDA register for certain business events and are informed immediately when an event occurs ("push" architecture). Hence, EDA is a form of publish and subscribe. David Luckham [13] distinguishes *complex event processing* (CEP) as a more advanced idea from EDA. CEP concentrates on ordering and making use of event information, whereas EDA concentrates on distribution and management of event information.

In this paper, we are not primarily interested in processing of events and their information, but in their variety and "desirability" for the user of the SUC that is to be designed, modeled and tested. We assume that the user is mostly interested in eventually reaching at a final, *desirable* event; thus, the modeling process has to anticipate *undesirable* ones that can prevent him, or her, from reaching of this goal. Accordingly, alternative ways have to be created to reach the goal. An example is given by a WS for credit card check that a bank integrates into different applications and is also publicly available for integration into diverse shopping systems. The desirable final event for both the owner of the credit card and credit card institute is to conclude the business and enjoy the benefits. Nevertheless, undesirable events, as to constraints given by credit limitation, or, more critical, fraud attempts, must be taken into account.

3 Event-driven modeling and testing of web service behavior

The idea of WSs and SOA in general entails providing central operations for recurrent tasks. Now, imagine the likely implementation of a shopping cart: before transactions to the shopping cart are allowed, a valid login is required. If the login is successful, an operation *addItem(itemID)* can be called to put an item into the shopping cart. Finally, a *checkout()*-function can be activated for ordering the selected items. But: What happens if the *addItem()*-operation is called first? This should result in a fault message, and/or warning. The motivation is to test such situations systematically. As stated in Sect. 2, response time of operations may also play a vital role for WSs and can depend on selected parameter values as well as the response itself depends on them. Therefore, a further motivation is to enable systematic modeling and selection of input parameters for testing.

3.1 Holistic view of valid and invalid behavior

This work uses ESG notion [4] for modeling WSs and web applications.

Definition 1 An *event sequence graph* $ESG = (V, E, \Xi, \Gamma)$ is a directed graph where $V \neq \emptyset$ is a finite set of vertices (nodes), $E \subseteq V \times V$ is a finite set of arcs (edges), $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$ and $\gamma \in \Gamma$ called entry nodes and exit nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $\langle \xi, v_0, \ldots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

An *event* is specified as an externally observable phenomenon, e.g., a user stimulus or a system response. An event according to WSs is a *request* (call of an operation; denoted as V_{req}) to the SUC or a *response* (server response message to a "request"; denoted as V_{resp}). Therefore, the set V of an ESG for a WS (ESG4WS) is denoted as $V = V_{req} \cup V_{resp}$. Note that edges $E \in V_{resp} \times V_{resp}$ are not part of an ESG4WS as -technically- one request requires exactly one response to follow or no response. Moreover, a node of the ESG can be refined by another ESG called *structured ESG* (sESG).

As described in Definition 1, ESG are directed graphs; their nodes represent events and edges valid, correct pairs of events. Two pseudo vertices, '['and']', symbolize *entry* and *exit* where any node can be reached by entry node, and any node can reach the exit.

Definition 2 Let V, E be defined as in Definition 1. 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 of length 2 is also called *event pair (EP)*.

Graphically, any sequence of vertices connected by an edge is called a legal ES that consists of EPs. In Fig. 1, EPs are represented by solid arrows connecting two nodes.

Definition 3 Given two functions α(ES) and ω(ES), which return the first and last event of an ES. An ES is *complete* (or, it is called a *complete event sequence, CES*), if $\alpha(ES) = \xi \in \Xi$ is an entry and $\omega(ES) = \gamma \in \Gamma$ is an exit.

As it can be seen in Fig. 1, an ES starting at the pseudo vertex '['and ending with the pseudo vertex ']' is called CES.

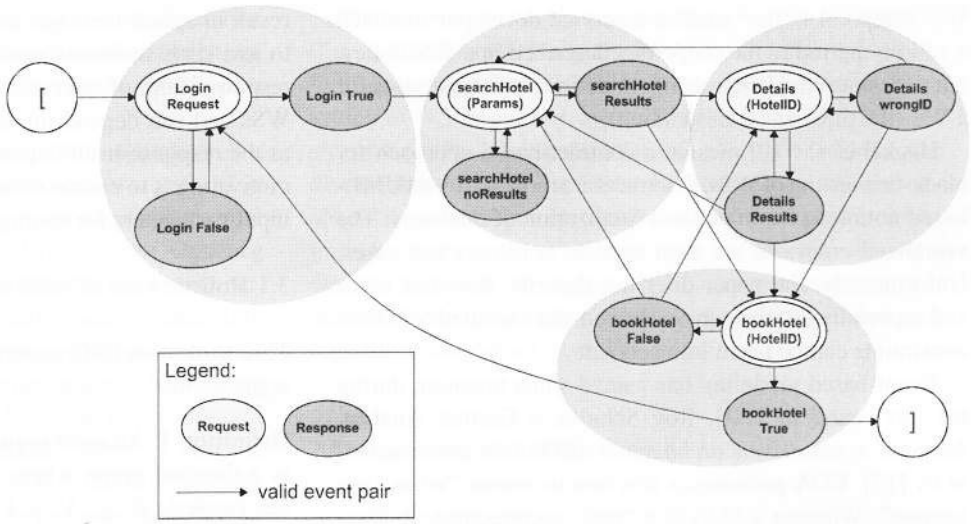


Fig. 1 ESG for the web service "hotel reservation"

CESs are considered to perform *successful* runs through the ESG, i.e., they are expected to arrive at the exit of the ESG that models SUC, in other words they deliver *desirable events*. For (positive) testing of SUC, CESs are used as test inputs.

The ESG depicted in Fig. 1 exemplifies a WS implementing a hotel reservation process; an extended version will be used also in the case study of Sect. 4. In this figure, requests and responses that belong together are grouped in a circle. Note that the operation *bookHotel()* returns a result list if and only if an hotel was found that fulfills the search criteria. Furthermore, a successful login is required before a search process for a hotel can be started.

A node of the ESG representing an operation call with input parameter is refined by a DT [7,9,19], i.e., input parameters and their structure are modeled by DTs. Thus, refined nodes are augmented by DT and in the corresponding ESG they are double-circled. DTs allow the modeling of valid system behavior depending on input parameters the variety of which is to be analyzed.

Definition 4 A (simple/binary) Decision Table DT $= \{C, A, R\}$ represents actions that depend on certain constraints where:

- $C \neq \emptyset$ is the set of constraints/conditions (propositions) that can be true or false
- $A \neq \emptyset$ is the set of actions
- $R \neq \emptyset$ 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 action

Table 1 shows a rudimentary DT for handling login requests. The top left cell identifies the operation represented by the DT. The *constraint part* specifies *a)* valid domains to input parameters ($\sum^*$defines the input alphabet) and *b)* additional constraints (in italics). These constraints are the pre-conditions of a contract and have to be resolvable to true or false. A dash '-' ("don't care") indicates that a constraint does not need to be considered in a rule. The action part specifies valid system responses with respect to constraint sets, called *rules* (see column "R1", "R2", ...). Thus, actions are the post-conditions of a contract, more specifically, the expected system responses according to specific parameter values. Hence, every rule of the DT defines a *contract*.

Table 1 DT modeling "LoginRequest"

	LoginRequest(usr, pwd)	Rules				
		R1	R2	R3	R4	R5
	usr $\in \sum^*$	T	T	T	F	F
Constr.	pwd$\in \sum^*$	T	T	F	T	F
	(usr, pwd) $\in$Login	*T*	*F*	–	–	–
Act.	LoginResponse(True)<200 ms	X				
	LoginResponse(False)<200 ms		X	X	X	X

Definition 5 Let R be a set of rules as defined in Definition 4. Then, a *rule* $R_i \in R$ is defined as $R_i = (C_{True}, C_{False}, A_x)$ where:

- $C_{True} \subseteq C$ is the set of constraints that have to be resolved to true
- $C_{False} \subseteq C \backslash C_{True}$ is the set of constraints that have to be resolved to false
- $A_x \subseteq A$ is the set of actions 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

System responses' description can be extended by time constraints where a response is expected within a certain

time range. Table 1 illustrates an example of a DT that represents the operation "LoginRequest" of Fig. 1. As an example, selecting data according to **R1** means that a response is expected within 200 ms. If a response takes longer, the corresponding test fails. Note that time bounds considered in this paper are fixed ones. This sounds very simple as the model introduced is that simple (which is our goal). But this brings up also limitations for our model, e.g., continuous time bounds cannot be considered. Therefore, other models, e.g., timed automata [2], could be considered if the necessity arises to analyze more complex systems and processes. Nevertheless, within the assumption that one knows the underlying database, he, or she, is able to define also dynamic time constraints. As an example, imagine that there are 10 results given by a search operation, then the given time constraint can be multiplied by 10. This aspect is concluded in the future work pointed out in the conclusions later on. Conclusions will also mention the need for checking the achieved responses. This, in turn, necessitates knowledge on the database, which will enable dynamic time bounds.

The objective of the approach is to identify single WS operations that hurt a given time constraint. In general, also the total time for each test case can be calculated as the sum of the individual time bounds of each operation call within this test case.

It can be expected that third parties have access to the WSDL-File, which describes the interface of the WS. Unfortunately, the WSDL does not describe valid or invalid orders in which WS's operations are supposed to be called. Therefore, testing of invalid sequences of events is also of substantial interest.

Definition 6 Let V_{req} be the set of requests and V_{resp} be the set of responses with $V = V_{req} \cup V_{resp}$. For an $ESG = (V, E, \Xi, \Gamma)$, its *completion* is defined as $\widehat{ESG} = (V, \widehat{E}, \widehat{\Xi}, \Gamma)$ with $\widehat{E} = V \times V \setminus V_{resp} \times V_{resp}$ and $\widehat{\Xi} = V_{req}$.

Definition 7 The *inverse* (or complementary) ESG are then defined as
$\overline{ESG} = (V, \overline{E}, \overline{\Xi}, \Gamma)$ with $\overline{E} = \widehat{E} \setminus E$ and $\overline{\Xi} = \widehat{\Xi} \setminus \Xi$.

To test the undesirable behavior, the ESG4WS is to be *completed* by additional edges, represented as follows (dashed lines in Fig. 2):

- Add edges from response vertices to request vertices where no edges exist ("Response" → "Request").
- Add edges from pseudo vertex "[" to request vertices where no edges exist ("[" → "Request").
- Add edges from request vertices to request vertices, where no edges exists ("Request" → "Request").

Definition 8 Any EP of the $\overline{ESG}$ is a *faulty event pair (FEP)* for ESG.

The additional edges symbolize FEPs as they should lead to a negative system response, i.e., executing such EPs the SUC is expected to react with a warning. For testing the system reaction upon an unexpected, undesirable event, FEPs are to be extended to *faulty complete event sequences* (FCES).

Definition 9 Let $ES = \langle v_0, \ldots, v_k \rangle$ be an event sequence of length $k + 1$ of an ESG and $FEP = \langle v_k, v_m \rangle$ a faulty event pair of the corresponding $\overline{ESG}$. The concatenation of the ES and FEP then forms a *faulty event sequence* $FES = \langle v_0, \ldots, v_k, v_m \rangle$.

Definition 10 A FES is *complete* (or, it is called a *faulty complete event sequence, FCES*) if $\alpha(FES) = \xi \in \Xi$ is an entry. The ES as part of a FCES is called a *starter*.

A FCES starts with the pseudo vertex '[' and ends with the FEP under consideration. FEPs containing refined "Request"-nodes have to be thoroughly resolved with data. Every rule of the DT should carefully be checked as the result could depend on selected values. The ESG given in Fig. 1 is extended by dashed edges, which are FEPs. Executing those FEPs should result in a SOAP fault (symbolized by vertex "Fault"). Wherever another system response arises (perhaps one of the regular ones), the corresponding tests are failed ones (negative testing). The described way of testing both desirable and undesirable behavior by means of positive and negative testing using the same model enables a *holistic testing*.

3.2 Fault model and test process

The model developed in Sect. 3.1 defines by means of its completion a set of fault models and enables a meaningful coverage criterion: All EPs of the ESG given are to be covered by CESs of minimal total length, and/or of minimal number of CESs. Graphically, this means a coverage of all edges in an ESG. This problem is a derivation of the *Chinese Postman Problem* (CPP) [1] that has $O(n^3)$ runtime complexity in our case [4].

The generation of data out of DTs leads to the *Constraint Satisfaction Problem* (CSP) that has a polynomial runtime complexity. According to Russell, Norvig [20], 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 non-empty 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". Each rule of the DT under consideration represents a CSP. Algorithm 1 sketches the overall test process. Lines 1–6 deal with generation of test sequences, lines 7–18 describe their execution. As given in Algorithm 1, faults are identified if execution of a CES does not reach the final event. This is the case if another response as the expected one occurs or time constraints are

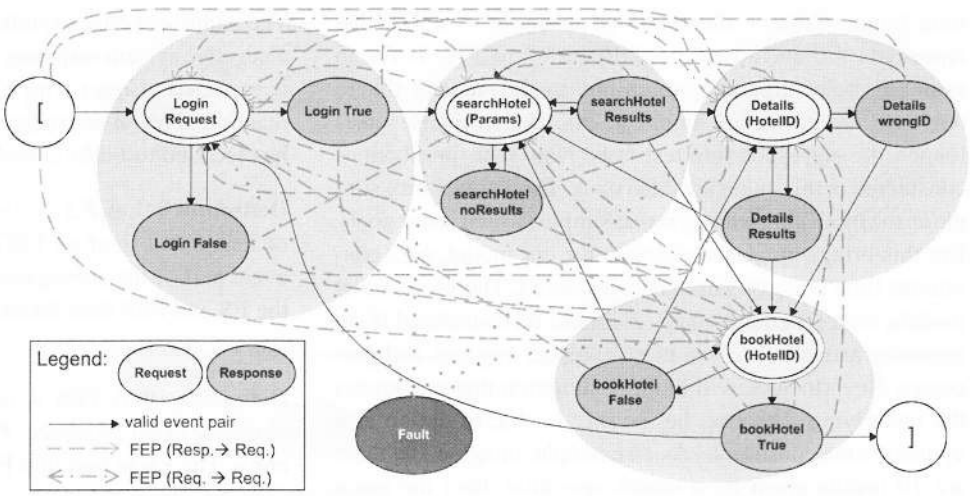


Fig. 2 Completed ESG according to Fig. 1

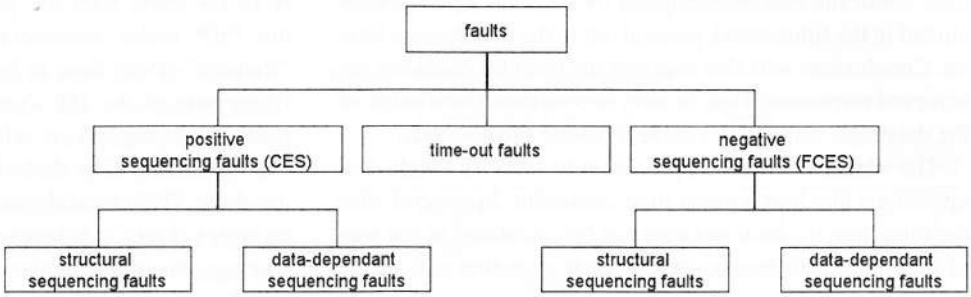


Fig. 3 Fault classes

violated. Faults with respect to FCESs are identified if the sequence does not lead to a fault response, maybe because a valid response under regular circumstances has occurred.

The detected faults are classified in *positive* and *negative sequencing faults* if they are detected by CESs or FCESs, respectively. Both categories can be refined into *structural sequencing faults* and *data-dependent sequencing faults*. Structural sequencing faults arise if parameters have not caused this undesirable event; otherwise, we have data-dependent sequencing faults. Where time limits of system responses are exceeded, faults are categorized as *time-out faults*. Figure 3 gives a hierarchical overview of defined fault classes.

Note that the explosion of state space is a general problem of generating test cases on the basis of graphs. We addressed this problem with (a) considering events instead of states, (b) a hierarchical collection of *structured ESG* (sESG), where a node can represent another ESG that is to be tested all alone and (c) the constraints represented in DTs.

3.3 Tool support

To avoid tedious, time-consuming and error-prone manual work, a tool (see Fig. 4) for generation of event sequences has been built that implements a solution to the CPP-Problem mentioned in Sect. 3.2. Based on an ESG given, the tool constructs a minimal set of CESs. The graph can be created and edited directly using the tool. Moreover, a graph saved in graphml-format can be imported. Before CESs are generated, the tool checks whether the graph given represents a valid ESG.

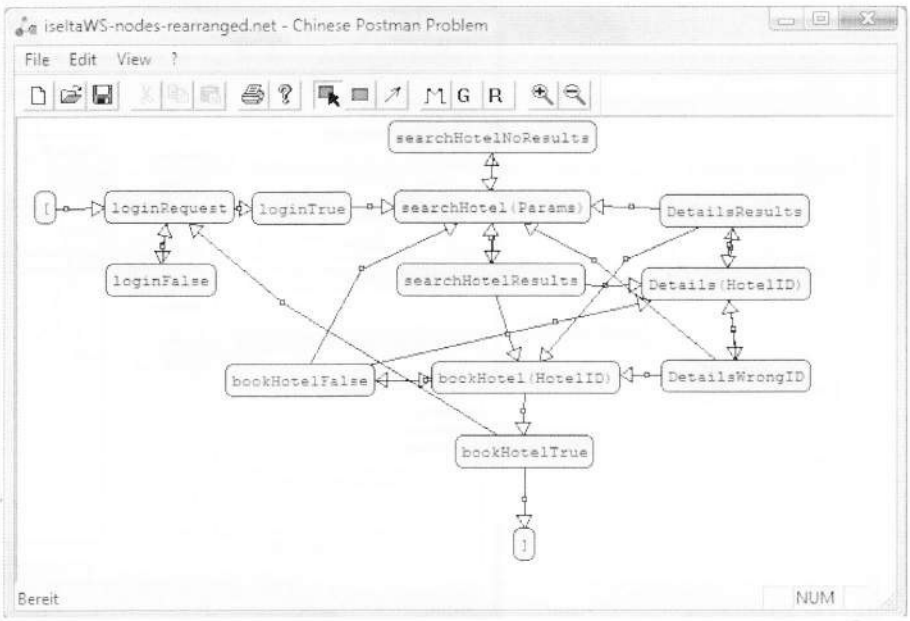
Another tedious work while testing is the generation of test data out of DTs. Therefore, a tool we developed and called "ETES" generates data out of DTs (see Fig. 5). ETES [23] evaluates constraints given, finds possible solutions w.r.t. the CSP mentioned in Sect. 3.2. and transfers all or subsets of solutions into a XML-File. Moreover, it supports the user by a sample data generation method for domains. It is even possible to import sample data given in csv-Files. This enables to use well-known test data generation tools, e.g., found in http://www.generatedata.com where csv-Files can be exported.

4 Case study

4.1 System under consideration

The SUC of the case study is a large commercial web portal with 53,000 LOC (lines of code) called ISELTA™ (*Isik's System for Enterprise-Level Web-Centric Tourist Applications*; visit http://www.iselta.com). ISELTA™ enables travel and tourist enterprises, e.g., hotel owners, recreation/fitness centers, to create their individual search & service offering

Fig. 4 CPP-Tool



Algorithm 1 Test process

```
generate a minimal set of CESs containing all EPs of the given ESG (CPP)
generate a set of FCESs containing all FEPs of the given ESG
FOREACH CES and FCES with decision tables DO
    generate data-expanded CES/FCESs on the basis of DTs (CSP)
FOREACH CES and FCES with sESG DO
    generate multiple CES/FCES by replacing the structured nodes with the
        CESs of the sESG [4]
FOREACH CES DO
    execute CES
    IF final event is not reachable OR time constraint was violated
        remark test as failed
    ELSE
        remark test as passed
FOREACH FCES DO
    execute FCES
    IF FCES leads to a Fault-Response
        remark test as passed
    ELSE
        remark test as failed
```

masks (Fig. 6). These masks can be embedded in the existing homepage of the hotels as an interface between customers and system. Potential customers can then use those masks to select and book services, e.g., hotel rooms.

A special service is given by the implementation of hotel reservation as a WS. This WS enables enterprises to embed ISELTA into own applications. Among others, following operations are available:

- *login(username, password)*: operation for login, returns true or false
- *search(params)*: operation for searching an offer, returns a result list
- *details(HotelID)*: returns detailed information
- *bookHotel(HotelID)*: enables reservation of an hotel, returns true or false
- *viewBooking(BookingID)*: enables to take a look on an existing booking, returns booking details
- *cancelBooking(BookingID)*: enables to cancel an existing booking, returns true or false
- *getAllBookings()*: returns all bookings done by a user

Searching and booking of services are carried out in four successive steps: First, a user has to log in using his or her data. Second, search criteria as to arrival date and departure date, service/comfort level of the hotel (three stars, four stars, etc.), or type of catering has to be submitted. As a third step, detailed information on an object found can be inquired. Finally, the user can book an offer.

Fig. 5 ETES-Tool

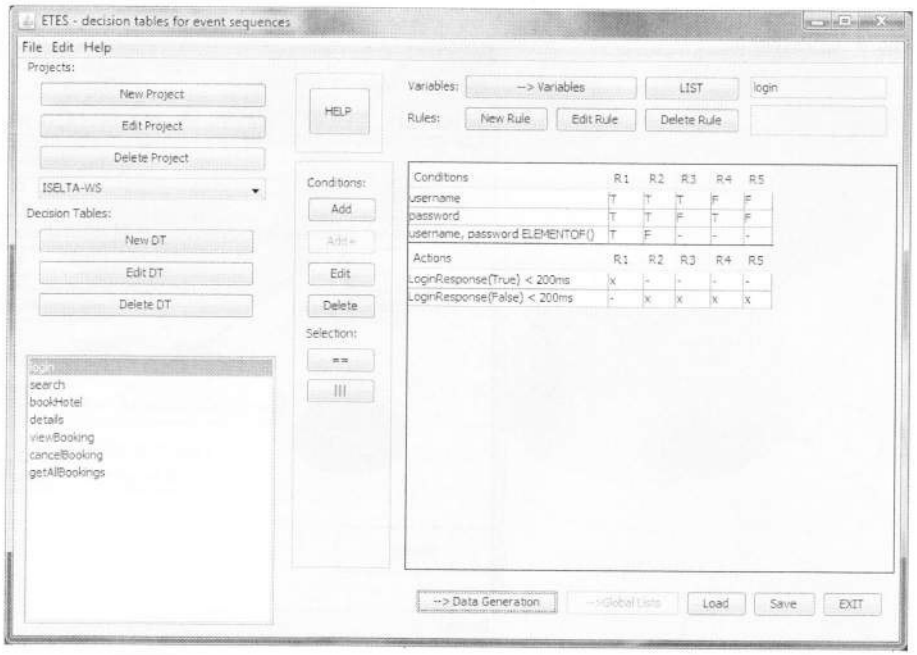
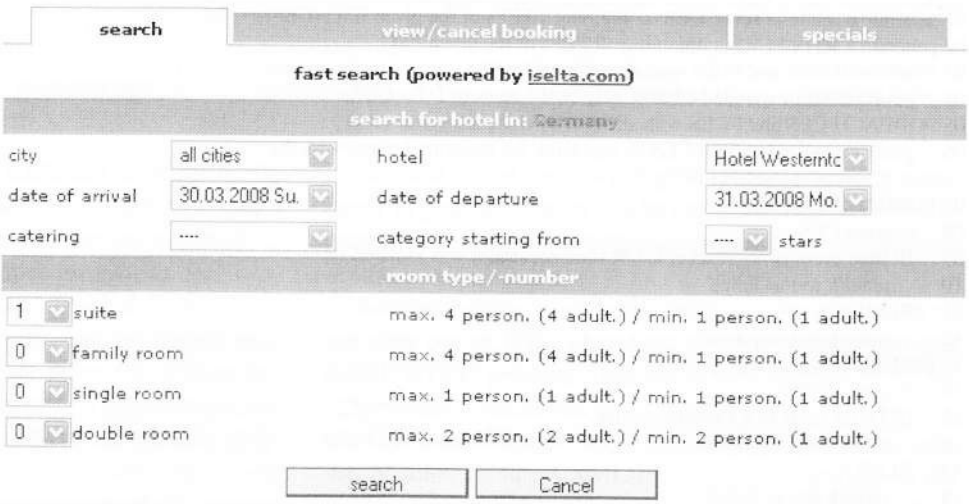


Fig. 6 Screenshot of ISELTA-WA



Moreover, the user can request additional information to achieve further results. Operation *getAllBookings()* enables to get all bookings done by the user. Detailed booking information can be requested using *viewBooking()*. Last but not least, the user can choose to cancel a booking.

4.2 Architecture of ISELTA

Figure 7 sketches the architecture of ISELTA and the shared resources of ISELTA web service (ISELTA-WS) and ISELTA web application (ISELTA-WA). ISELTA-WA includes Logic Files for checking data correctness before passing them to the classes. Furthermore, those Logic Files select and

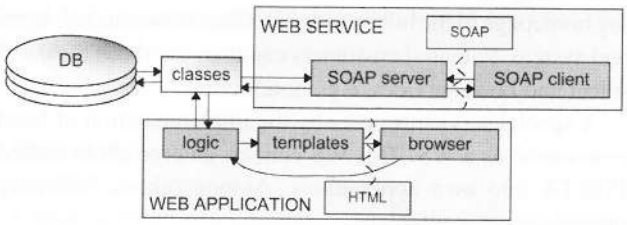


Fig. 7 Architecture of ISELTA

include HTML-templates on the basis of constitution of data. ISELTA-WS is built up on the same classes and database. Thus, ISELTA-WA and ISELTA-WS have the same basis. This allows to use ISELTA either by its in-built web

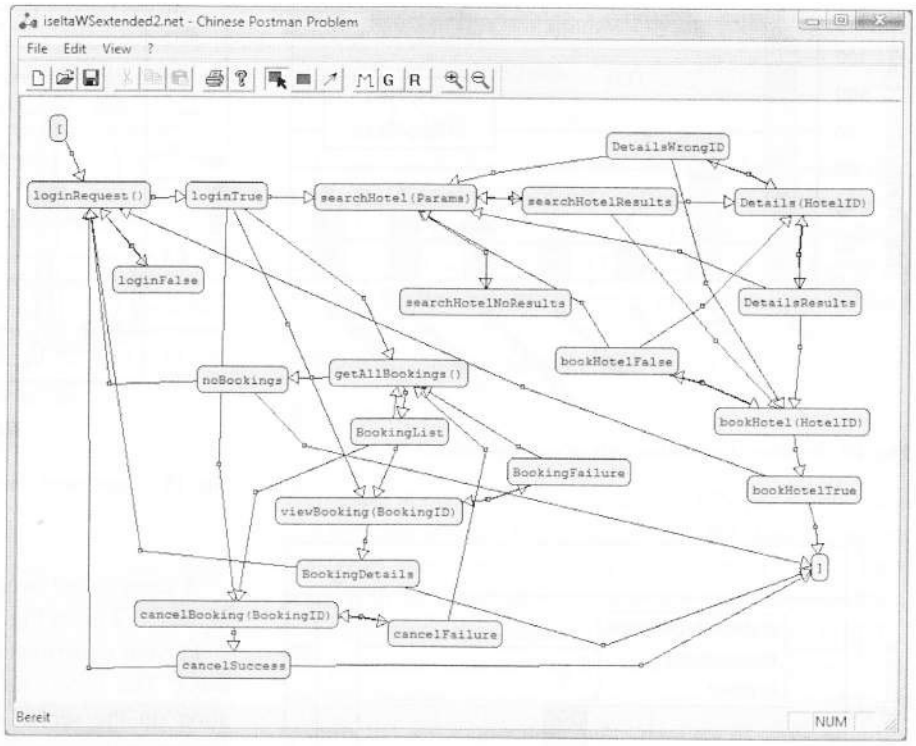


Fig. 8 ESG for ISELTA-WS testing

Table 2 Test sequence example

1. login("false","user")
2. login →false <200 ms
3. login("max", "pwd")
4. login →true <200 ms
5. searchHotel("08-08-2008", "12-08-2008", "Paderborn", "SR=1", "all", "all")
6. searchHotel → ResultList <1,000 ms
7. details("2")
8. details →detailsResults <200 ms
9. bookHotel("2")
10. bookHotel →true <200 ms

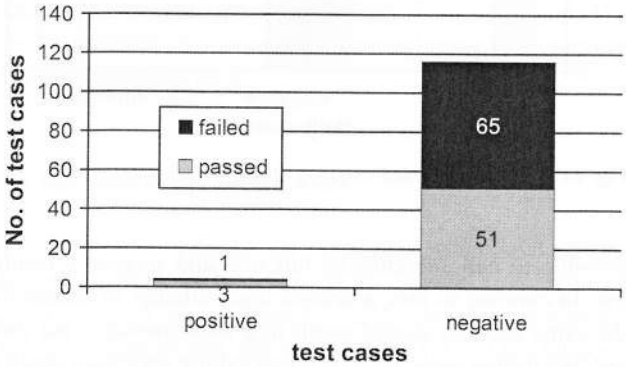


Fig. 9 Number of test cases

interface or by its web service interface that especially enables to integrate ISELTA in foreign systems.

4.3 Experiments & results

Figure 8 depicts the corresponding ESG used in our case study. Refinement of this ESG and corresponding DTs, test cases are generated and tests are run by methods described in Sects. 3.1 and 3.2. Table 2 contains an example for a test sequence with input data added. Those test sequences have been implemented in PHP as web services client, and results are written to a csv-File.

Figure 9 shows an overview of all test cases grouped in sets of positive and negative test cases as well as failed and passed test cases. In total, 120 test cases have been executed, 66 of them (or 55%) failed and revealed a fault. The unproportional higher number of negative test cases depends on the fact that each FEP has to be tested by an own FCES as a successful execution of a FEP leads the system into an undefined state (w.r.t. the model considered). In contrast, several EPs can be tested by one CES. Figure 10 summarizes the number of operation calls of each operation during test execution. As the *login* operation is the initial operation for calling other operations, it has been executed most. Besides, for testing FEPs with two consecutive requests each operation was called six times in test cases, where only a request was sent to the server.

Testing ISELTA-WS revealed six severe faults. For security reasons, a successful booking or cancelation of a booking should have logged out the user, but did not, i.e., it was

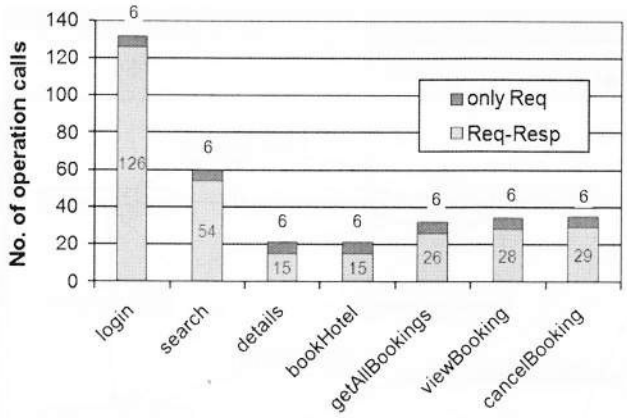


Fig. 10 Number of operation calls of each operation

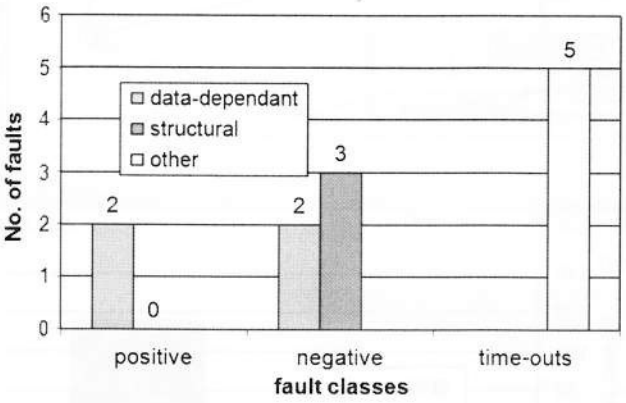


Fig. 11 Faults categorized into fault classes

possible to call *searchHotel*-function and receive a result list. In contrast to this, a second login attempt to access to the same account should result in a fault message, but did not. We further detected a fault that calling operation *details* requires storage of search parameters in session cache. Operation *searchHotel* should reset parameters in cache before executing the search query. We detected also some SOAP faults where a response contained the German characters 'ä','ö','ü'. Hence, those characters should be replaced by their HTML pendants. The fifth fault occurred by calling *searchHotel*. A call of *details* led to wrong details information. All these faults are uncovered by our negative test cases. The sixth fault enabled to cancel elapsed bookings. Moreover, cancelation of already canceled bookings did not result in a fault message like "already canceled".

Note that ISELTA-WA has been tested in prior case studies. As ISELTA-WS and ISELTA-WA have the same basis, faults that have their origin in classes should also be revealed by testing ISELTA-WS, e.g., testing ISELTA-WA revealed a fault that a password check had not differentiated between upper and lower cases. This fault could be confirmed.

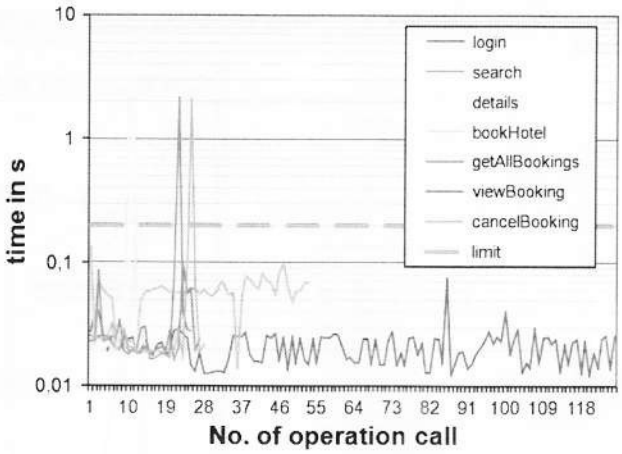


Fig. 12 Client-sided time of all operations

A categorized overview into the fault classes mentioned in Sect. 3.2 gives Fig. 11.

For our experiments, we measured time in two different ways: The *server-sided time* is measured within an operation on the server and reflects the computational effort for generating a response; the *client-sided time* represents the time between request and response on the client. Note that the client-sided time includes the server-sided time plus transfer-time for request and response via network. As we choose black-box testing of our SUC, the assumption is that we test WSs from the view of the client side. Therefore, our model considers the SUC from the client side. Measuring times on both sides enables to detect the reason for time discrepancies, which can be on the server and, hence, in the implementation itself, or due to network load. To eliminate misleading network effects, we tested locally on one computer. Nevertheless, our experiments measured both, so it would have been possible to detect such differences, but we do not model different times.

Figure 12 depicts the client-sided time of all operations. Obviously, some operations violate the time constraint of 200 ms, e.g., operation *bookHotel*.

Figure 13 shows the server-sided and client-sided time of *bookHotel*. The difference between server time and client time was nearly constant and reflects the circumstance that we tested locally but there are two calls of the operation that result in an extraordinary long response time (>2 s). As the server-sided time is already breaking the time constraint, there seems to be a problem with the calculation of a result in some cases. A deeper look into these calls reveals that the time is extraordinary long in the case of a successful booking. There are different tasks that have to be executed in this case, e.g., sending mails or faxes, reduction in available rooms. For fault correction, the programmer has to check the different tasks and correct the operation so that the time constraint will hold.

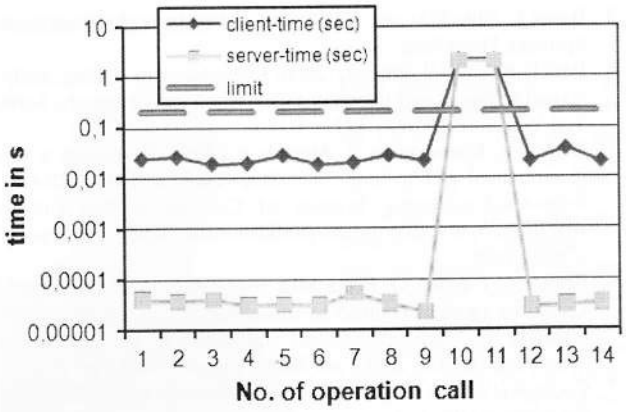


Fig. 13 Request and response times for operation"bookHotel"

To sum up, the case study shows that especially negative testing using FCESs has a very high potential to reveal substantial faults and thus must not be left out. In our case study, time constraints have been violated five times by four different operations and show the importance of considering time constraints especially for RTSOA. Worthwhile to be mentioned is also the fact that measuring time only once means not much and is thus insufficient; operations should be executed several times and with different responses to check several factors of them as it can bee seen in our case study.

Table 3 gives an overview of our positive test cases and their coverage of nodes and valid EPs of our ESG model given in Fig. 8. This ESG consists of 21 nodes and 48 edges, which represent valid EPs. Thus, test case # 4 covered 12 of 21 nodes and 26 of 48 edges. Note that a single node or edge might have possibly been executed more than once while exercising test case # 4. Here, a covered node or edge is only counted once even if it is covered several times within a test case. As CESs form our test cases, the length of a test case is simply the length of its corresponding CES. For prioritizing the test cases, i.e., to decide, which test cases have to be exercised first, check EPs that are covered most within a test case. For the previous example, this is test case # 4 as it covers 54% of all EPs. A second look reveals that this test case is much longer than the other ones, i.e., several EPs are executed more than one time within this test case. The test *effectiveness ef* per single event can be calculated by the following formula:

$$ef = \frac{\frac{EPs\ covered\ in\ test\ case}{Total\ EPs} \times 100}{Length\ of\ test\ case}$$

It can easily be shown that the calculated test effectiveness is always between $0 < ef <= \frac{100}{Total EPs}$ since $Length\ of\ test\ case >= EPs\ covered\ in\ test\ case$. In our case, this is $0 < ef < 2.083$. To subsume, the higher the number, the less EPs are executed more than once within a given test case.

For test case # 4, the test effectiveness is:

$$\frac{\frac{26}{48} \times 100}{74} = 0.7320$$

The result shows that the coverage of EPs per single event of a test case is much higher in test case # 1 than in test case # 4. Thus, considering the two aspects could help in large projects to decide which test cases have to be exercised first.

Extra attention is necessary to the fact that testing ISELTA-WS produced some real bookings while target service environment is used as a test frame. Therefore, while testing a system, the test engineer should take care of those environmental issues. We recommend service providers to build a second, functional equivalent WS interface for testing. This enables service consumers to test correctness of own services before publishing them.

5 Conclusions and future work

This paper introduced an approach for modeling and testing of WSs in real-time SOA. Events as to requests and responses are modeled by ESG, whereby corresponding input parameters are modeled by DTs. This allows a simple, nevertheless powerful means for modeling and analysis of the system under consideration (SUC). The option to test also whether SUC reacts correctly in unexpected situations completes the approach, which we call negative testing. CESs are generated from ESG supported by a tool that evaluates constraints given in augmenting decision tables (DT) and generates data on the basis of those constraints. Although our approach concentrates on stateful WSs where sequences of operation calls play a vital

Table 3 Positive test cases and their coverage of nodes and valid EPs

	Total	Test case 1	Test case 2	Test case 3	Test case 4
No. of nodes covered	21	4 (19%)	7 (33.3%)	7 (33.3%)	12 (57.1%)
No. of edges covered	48	6 (12.5%)	12 (25%)	11 (22.9%)	26 (54.2%)
Length of test case	–	7	24	18	74
Effectiveness per EP	–	~1.7857	~1.0417	~1.2731	~0.7320

role, the modeling of input parameters by DTs can also be applied to stateless WSs enabling a systematic evaluation and testing of input parameter constraints for a single WS operation.

Our future work aims at further reduction in the manual test effort by a better self-adaptability of the model due to likely changes in applications. Therefore, we plan to bridge the gap between mapping of CESs and data generation out of DT, which currently has to be done by hand. This mapping took approximately one extra hour manual effort while performing the experiments of our case study. Furthermore, combining both tools will enable to generate an executable test script, which makes the maintenance of test scripts much easier by simply regenerating them. Additionally, dealing with data leads to the necessity of checking results, handling the problems as to, referring to a typical question of the case study: "Were all of the hotels found correctly or is still some missing?" Therefore, we plan the extension of data checking. For this, we need to know the content of the database, which will, in turn, enable dynamic time bounds. If we know the expected results of an operation beforehand, e.g., a search operation, we are then able to set more dynamic time constraints. The time bounds considered in this paper are fixed ones but can easily be adjusted by hand within a specific test case where we know the exact response. Hence, our goal is to reach an automatic calculation of dynamic time bounds. Within this paper, our goal was to identify single WS operations that hurt a given time constraint. Alternatively, we are able to calculate a total time constraint for each test case, which would be the sum of the single time bounds of each operation call within this test case. This is just another point of view.

Finally, we recommend service providers to publish a functional equivalent interface for testing their service, which is enriched by information on server execution times. A testing interface is especially necessary where a service has a decisive influence on its environment as to binding bookings. This supports service consumers during implementation and test of own applications. Moreover, a testing interface enables external users to reveal changes in the service if tests are run periodically. Assuming that those testing interfaces would be described in a standardized manner within a WSDL-File, they even help to find adequate services where services are bound dynamically.

References

1. Aho A, Dahbura A, Lee D, Uyar M (1991) An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. pp 1604–1615
2. Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235
3. Baresi L, Nitto ED (eds) (2007) Test and analysis of web services. Springer, Heidelberg
4. Belli F, Budnik CJ, White L (2006) Event-based modelling, analysis and testing of user interactions: approach and case study. Softw Test Verif Reliab 16(1):3–32
5. Bianco P, Kotermanski R, Merson P (2007) Evaluating a service-oriented architecture. Tech. Rep. CMU/SEI-2007-TR-015, Software Engineering Institute of Carnegie Mellon University, http://www.sei.cmu.edu/publications/documents/07.reports/07tr015.html
6. Bruno M, Canfora G, Penta MD, Esposito G, Mazza V (2005) Using test cases as contract to ensure service compliance across releases. In: Third international conference in service-oriented computing (ICSOC 2005), vol 3826. Springer, Berlin, pp 87–100
7. Codasyl (1982) A modern appraisal of decision tables. Report of the decision table task group, pp 230–232
8. Dranidis D, Kourtesis D, Ramollari E (2007) Formal verification of web service behavioural conformance through testing. Ann Math, Comput Teleinformatics 1(5):36–43
9. Garcia AM, Verhelle M, Vanthienen J (2000) An overview of decision table literature 1982–2000. In: Fifth international conference on artificial intelligence and emerging technologies in accounting, finance and tax
10. Heckel R, Lohmann M (2005) Towards contract-based testing of web services. Electr Notes Theor Comput Sci 116:145–156
11. Jordan D, Evdemon J (2007) Web services business process execution language version 2.0. Tech. rep., OASIS, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
12. Keum C, Kang S, Ko IY, Baik J, Choi YI (2006) Generating test cases for web services using extended finite state machine. In: TestCom, pp 103–117
13. Luckham DC (2001) The power of events: an introduction to complex event processing in distributed enterprise systems. Addison-Wesley Longman Publishing Co., Inc, Boston
14. Marconi A, Pistore M, Traverso P (2007) Process-level composition of web services: a semi-automated iterative approach. Ann Math, Comput Teleinformatics 1(5):11–25
15. Mathes M, Heinzl S, Freisleben B (2008) Ws-temporalpolicy: a ws-policy extension for describing service properties with time constraints. Annu Int Comput Softw Appl Conf 0:1180–1186
16. Meyer B (1997) Object-oriented software construction. 2. Prentice-Hall Inc, Upper Saddle River
17. Michelson B (Feb 2006) Event-driven architecture overview. Patricia Seybold Group
18. Ortiz G, Leymann F (2006) Combining ws-policy and aspect-oriented programming. In: Telecommunications, 2006 AICT-ICIW '06 International conference on internet and web applications and services/advanced international conference on pp 143–148
19. Pollac LS, Hicks H, Harrison W (1971) Decision tables: theory and practice. Wiley, New York
20. Russell S, Norvig P (2003) Artificial intelligence: a modern approach, 2nd edn, chap Constraints satisfaction problems. pp 137–160
21. Tsai WT, Paul R, Wang Y, Fan C, Wang D (2002) Extending wsdl to facilitate web services testing. In: HASE '02: Proceedings of the 7th IEEE international symposium on high assurance systems engineering (HASE'02), IEEE Computer Society, Washington, DC, USA, p 171
22. Vedamuthu A, Orchard D, Hirsch F, Hondo M, Yendluri P, Boubez T, Yalcinalp U (2007) Web services policy. Tech. rep., World Wide Web Consortium (W3C), http://www.w3.org/TR/ws-policy
23. Zirnsak R (2009) Master thesis (working title): input data generation for positive and negative testing. Tech. Rep. 2009/1, University of Paderborn

Author Biographies

Fevzi Belli completed his PhD in formal methods for verifying software systems and self-correction features in formal languages at Berlin Technical University in 1978. He spent several years as a software engineer in Munich, writing programs to test other programs. After a brief stint as a senior scientist at the SHAPE Technical Center of NATO at The Hague, he became in 1983 a professor at the University of Applied Sciences in Bremerhaven; in 1989, he changed to the University of Paderborn. Prior to his appointment with the University of Paderborn, Dr. Belli was also a faculty member of the University of Maryland, College Park, European Division, for many years. During 2002 and 2003, he was founding chair of the Computer Science Department at the University of Economics in Izmir, Turkey. He has an interest and experience in software reliability/fault tolerance, model-based testing, and test automation. He was general chair and program chair of and invited speaker at several IEEE and ACM conferences on those topics. Visit http://adt.et.upb.de/personal.php?id=1 for publications.

Michael Linschulte is a PhD student and research assistant at University of Paderborn where he studied Business Computing Systems before. His research interests include analysis and testing of web applications as well as web services by means of model-based testing. Further, he is interested in test automation, software reliability and fault tolerance.