Constraint-Based Model-Driven Testing of Web Services for Behavior Conformance

Chang-ai Sun[1(✉)], Meng Li[1], Jingting Jia[1], and Jun Han[2]

[1] School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China
casun@ustb.edu.cn

[2] School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia

Abstract. In the current *Web Service Description Language* (*WSDL*), only the interface information of a web service is provided without any indication on its behavior logic. Naturally, it is difficult for the service user and developer to achieve a shared understanding of the service behavior through such a description. A particular challenge is how to make explicit the various behavior assumptions and restrictions of a service (for the user), and make sure that the service implementation conforms to them (for the developer). In order to improve the behavior conformance of services, in this paper we propose a constraint-based model-driven testing approach for web services. In our approach, constraints are introduced in an extended *WSDL*, called *CxWSDL*, to formally and explicitly express the implicit restrictions and assumptions on the behavior of web services, and then the predefined constraints are used to derive test cases in a model-driven manner to test the service implementation's conformance to these behavior constraints from the user's perspective. We have conducted an empirical study with three real-life web services as subject programs, and the experimental results have shown that our approach can effectively validate the service's conformance to the behavior constraints.

Keywords: Web services · Conformance testing
Model-driven testing · Test case generation

1 Introduction

In the context of Service Oriented Architecture, the implementation of web services is separated from their interface description. Service users invoke a web service only based on its interface description written in *WSDL*. Since *WSDL* provides only the signature information for web service invocations, such as types, messages, operations and bindings, a service description in *WSDL* cannot help consumers to understand the way in which the web service should be invoked because it does not indicate any restrictions or assumptions on the behavior of a service.

The behavior expected of a web service is the key to achieve the proper use of the service. A feasible way of avoiding the potential misuse of a service is to enhance the service description with the restrictions and assumptions that underlie its behavior as intended by the service developer. Furthermore, such behavior description can also be used to test the service implementation to ascertain the service's conformance to the expected behavior. However, this kind of behavior-related information is neither formally nor explicitly described in the *WSDL* description.

In this paper, we propose a constraint-based model-driven testing approach to improve the understanding and conformance of web service behavior. We leverage the description of behavior constraints to establish a bridge between service developers and service users. Constraints are used to formally and explicitly describe the implicit behavior restrictions and assumptions on service invocations and to validate the service implementation's conformance to them. The main contributions of this paper are as follows:

1. We summarize a range of common behavior constraints for web services that are useful for potential violation detection.
2. We design an extended *WSDL*, called *CxWSDL*, to incorporate the formal description of behavior constraints.
3. We develop a model-driven testing technique to validate the service implementation's conformance to the behavior constraints. The technique first derives a service behavior model from the constraint-enriched description of a service written in *CxWSDL*. Then, it uses three coverage criteria to generate test sequences from the behavior model, aimed at exercising the service implementation's support for the constraints. Test suites are consequently generated from the test sequences using a constraint solver, and used to test the web service from the user's perspective.
4. We evaluate the effectiveness of the proposed approach with three real-life web services.

The rest of this paper is organized as follows. Section 2 presents an overview of our approach. Section 3 summarizes the common behavior constraints and presents a formal description for them. Section 4 discusses the proposed constraint-based model-driven testing technique. Section 5 reports an empirical evaluation of the proposed approach. Section 6 discusses related work and the paper is concluded in Sect. 7.

2 Approach Overview

Our approach aims to achieve better understanding and conformance of service behavior and has two major aspects. First, we introduce behavior constraints to express the implicit restrictions and assumptions expected of a web service's implementation. In this regard, we extend *WSDL* to enable the explicit description of the various constraints on the invocation of a web service, resulting in *CxWSDL* (***W****eb* ***S****ervices* ***D****escription* ***L****anguage with* ***C****onstraints*). The service description written in *CxWSDL* provides the basis for a shared understanding of the service's behavior constraints between service users and service developers.

Second, we propose a model driven testing technique that first derives a $\boldsymbol{C}onstraint - based\, \boldsymbol{B}ehavior\, \boldsymbol{M}odel$ (CBM) of the service from its extended description in *CxWSDL*, then generates test cases from its *CBM*, and finally validates the service implementation's conformation to the constraints by executing the generated test cases. The proposed testing framework is shown in Fig. 1, which consists of five major components:

(1) *CxWSDL Parsing*, which parses the *CxWSDL* document provided by the web service developer to obtain the operations and constraints document for the web service, the *SOAP* message environment and the *XSD* (*XML Structure-definition Document*) for invoking these operations.
(2) *Behavior Model Construction,* which constructs the web service behavior model according to the operations and constraints document.
(3) *Test Path Generation*, which uses three coverage criteria and the web service behavior model to generate test sequences.
(4) *Test Case Generation*, which outputs an executable test suite by means of a constraint solver, taking as input a decision table provided by the web service developer and the previously generated test sequences.
(5) *Test Case Execution*, which simulates a client by executing the test cases, validates the conformance and violations to the constraints, and generates a test report according to the test results.

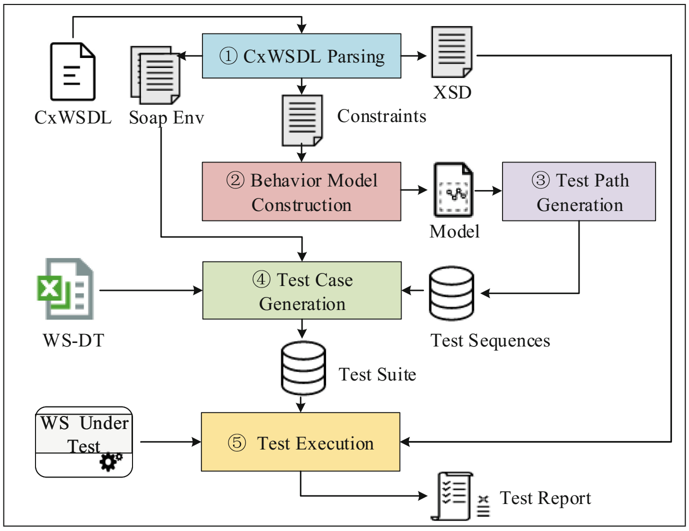

Fig. 1. Framework of model-driven testing of web services

A tool, called *MDGen*, has been developed to provide automated support for the above process. Its details cannot be included due to space limitation.

3 Constraints and Their Formal Description

This section summarizes the different types of behavior constraints and presents a formal description for them.

3.1 Types of Behavior Constraints

In our approach, constraints are used to explicitly express the behavioral assumptions and restrictions behind a service's implementation, which makes it possible to achieve a shared understanding between service developers and service users. That is, the service developer states the assumptions and restrictions in the description of a service via constraints, while a service user understands the service behavior via the stated constraints to achieve proper invocation of the service. From a literature review, we summarize the following common assumptions and restrictions, the misunderstanding of which may result in possible failures of service invocations.

- *Time Constraint* [16], which is necessary for restricting the service availability, especially when a service is being modified or in an inactive or maintenance state. If the access is outside its available period, a service invocation fault may happen due to the unavailability.
- *Region Constraint* [16], which restricts the valid range of *IP* addresses in case some operations of a service can only be accessed in a specific network.
- *Parameter Restriction Constraint* [7,18], which specifies the type and range of an input parameter of an operation.
- *Parameter Relation Constraint* [8,18], which states a relationship between the input parameters of different operations. Even if the input parameters conform to the *WSDL* type restrictions of the operations, the operation invocations may fail due to a violation of such a relationship constraint between the operations.
- *Sequence Constraint* [1,2,4], which can be a *Sequential Constraint* or a *Repeated Invocation Constraint*. The former specifies the order in which operations need to be performed or invoked for the service to function correctly, and the latter specifies whether an operation can be invoked repeatedly.
- *Invocation Constraint* [15], which identifies the other operations called by a given operation. An operation of a web service can involve another operation in performing its tasks, and thus it is important to trace and state such cascading relationships.

3.2 Formal Description of Constraints

We now consider the formal description of the above constraints. Due to the *XSD* type system can be used to define the types in a message and restriction defines the acceptable values for *XML* elements or attributes, the *XSD* restriction is well suited to describing *Parameter Restriction Constraint*. For description of other constraints, we introduce specific description constructs and their grammar is given in the *Extended Backus-Naur Form* (see Fig. 2).

```
<Constraint>        ::=  '{' (('"paraRelation":' <ValuePR> ',' '"ipRegion":' <ValueIR> ',' '"invokeOp":' <ValueIO> ','
                         '"preOp":' <ValuePO> ',' '"Iteration":' <ValueI>) | ('"eTime":' <eDate>)) '}'
<ValuePR>           ::=  '[' ']' | '['<ElementsPR>']'
<ElementsPR>        ::=  <Relationship> | <Relationship> ',' <ElementsPR>
<Relationship>      ::=  '"' <OpName> '.' <OpParameter> <RelationSymbol> <OpName> '.' <OpParameter> '"'
<RelationSymbol>    ::=  '=' | '>' | '<' | '>=' | '<=' | '!='
<ValueIR>           ::=  '"' <IpAdress> '-' <IpAdress> '"'
<IpAdress>          ::=  <IpField>'.' <IpField>'.' <IpField>'.' <IpField>
<IpField>           ::=  ('25'[0-5]|'2'[0-4][0-9]|(('1'[0-9][0-9])|([1-9]?[0-9])))
<ValueIO>           ::=  '[' '] ' | '['<ElementsIO>']'
<ElementsIO>        ::=  '"' <OpName>'"' | '"' <OpName>'"'',' <ElementsIO>
<OpName>            ::=  ([A-Z] | [a-z] | '_' | '$') (([A-Z] | [a-z] | [0-9] | '_' | '$'))*
<OpParameter>       ::=  ([A-Z] | [a-z] | '_' | '$') (([A-Z] | [a-z] | [0-9] | '_' | '$'))*
<ValuePO>           ::=  '"' <Exp>'"'
<Exp>               ::=  ('('<Exp>')*' | '('<Exp>')+' | '('<Exp>')|('<Exp>')' | '('<OpName>')('<OpName>'Response_succ)')*
<ValueI>            ::=  '"true"' | '"false"'
<eDate>             ::=  '"'<DateFormat>'"'
<DateFormat>        ::=  the value of Date Class whose Format is yyyy-MM-dd
```

Fig. 2. The grammar of *CxWSDL*

- *Constraint* identifies behavior constraints, expressed in the form of *JSON* attribute-value pairs, including *Parameter Relation Constraints* (i.e. *paraRelation*), *Region Constraints* (i.e. *ipRegion*), *Invocation Constraints* (i.e. *invokeOp*), *Sequential Constraints* (i.e. *preOp*), *Repeated Invocation Constraints* (i.e. *Iteration*), and *Time Constraints* (i.e. *eTime*).
- *ValuePR* states the *Parameter Relation Constraints*, which is a *JSON* array consisting of multiple relationships. A relationship between two parameters consisting of parameter names and relation operators. The relation operators include $=$, $>$, $<$, $>=$, $<=$, and $!=$.
- *ValueIR* states the *Region Constraints*, normally specifying the address range from which an operation can be accessed.
- *ValueIO* states the *Invocation Constraints*, which is a *JSON* array of multiple operation names that identify other operations called by an operation.
- *ValuePO* states the *Sequential Constraints*, which defines the sequential dependencies required for an operation being correctly executed, in a form of regular expressions, supporting repetition (*, +) and alternation (|).

– *ValueI* states whether an operation can be invoked repeatedly, indicated with true or false.
– *eDate* states the available time for a web service in the yyyy-mm-dd form.

To support the deployment of a service whose description is written in *CxWSDL*, we utilize the <documentation> element in *WSDL*, which is a container for human readable documentation. *Time Constraints* are added to the <*documentation*> element under the <*service*> element, and other types of constraints are added to the <*documentation*> element under each <*operation*> element. In this way, an existing container that supports *WSDL*-based web services can be directly used to deploy an extended service with behavior constraints in *CxWSDL* without any modifications.

4 Constraint-Based Model-Driven Testing of Web Services

This section presents our approach to detecting invocation violations to service behavior constraints, which improves the behavior conformance of services in a service-based system.

4.1 Constraint-Based Behavior Model Generation

In order to detect the improper invocations that violate the behavior constraints, we uses *Model Based Testing* (*MBT*) [12] for test case generation and violation detection. In particular, we propose the *Constraint-based Behavior Model* (*CBM*) of a web service based on event sequence graph.

Definition 1 (*CBM*). The *Constraint-based Behavior Model* is defined as a 4-tuple $CBM = <N_s, D, V, E>$, where

– N_s is the name of the model corresponding to a given web service,
– D is the available date of web service,
– V is a finite set of nodes in the CBM, representing the request events (operation invocations) or response events (responding to the request),
– E is a finite set of edges, representing a directed transfer from one node to another, i.e. $E \subset V \times V$.

Definition 2 (*Node*). Let V be the node set of a CBM. Each node v_i in V = $\{ v_0, \ldots , v_n \}$ is represented as a 6-tuple $v_i = <N_d, I_d, C, Pre, Suc, T>$, where

– N_d is the name of v_i,
– I_d is the unique identity of v_i,
– C is the set of constraints of v_i as defined in Sect. 3,
– Pre is the set of *Preceding Nodes* of v_i,
– Suc is the set of *Succeeding Nodes* of v_i,

– T is the type of v_i, where the different node types are: *Start* (i.e. the entry of the *CBM*), *Initial* (i.e. the initialization of a service invocation process), *End* (i.e. the end of the *CBM*), *Request* (i.e. a request event), and *Response* (i.e. a response event).

Definition 3 (*Preceding Node*). Let V be the node set of a CBM. We refer to v_i as a *Preceding Node* of v_j (denoted as $preNode(v_j)$), if and only if the following condition is true: $v_i \in V$, $v_j \in V$, and $(v_i, v_j) \in E$.

Definition 4 (*Succeeding Node*). Let V be the node set of a CBM. We refer to v_i as a *Succeeding Node* of v_j (denoted as $sucNode(v_j)$), if and only if the following condition is true: $v_i \in V$, $v_j \in V$, and $(v_j, v_i) \in E$.

Definition 5 (*Edge*). Let E be the edge set of a CBM. Each edge e_i in $E = \{ e_0, \ldots, e_m \}$ is defined as a 3-tuple $e_i = <N_e, FR, TO>$, where

– N_e is the name of the edge e_i,
– FR refers to the identify of the source node of e_i,
– TO refers to the identity of the target node of e_i.

We propose Algorithm 1 to construct a *CBM* from a *CxWSDL* document. It has the following major steps:

– *Initialization (lines 1–5)*: Initialize the *Behavior Model*, G, set its name property and *Time Constraints*, and add the *Start*, *Initial*, and *End* nodes to G.
– *Add nodes into the model (lines 6–15)*: Parse the *CxWSDL* document to identify the set of the operations of the web service under test. For each operation, add the *Request* and *Response* nodes to G and associate each node with the constraint properties.
– *Build sequence relation of nodes (lines 16–25)*: Set the sequence relation of nodes according to the sequence-related constraints.
– *Add edges (lines 26–30)*: For each node in the model, add an edge between the node and each of its *Succeeding Nodes* to the set of edges of G.

Note that the sequence-related constraints determine the behavior model's structure, which will be used to generate the test sequences (see Sect. 4.2). The non-sequence-related constraints are associated with the model's nodes, and will be used to generate test cases (see Sect. 4.3).

4.2 Test Sequence Generation

Test sequences can be generated from the service behavior model and they are classified into two types, namely *Constraint comPliant Sequences* (*CPSs* for short) and *Constraint conFlicting Sequences* (*CFSs* for short).

Definition 6 (*Constraint Compliant Sequence*). Let V and E be the node and edge sets of a CBM, respectively. A sequence of nodes $<v_0, \ldots, v_k>$ is called a *Constraint Compliant Sequence* (*CPS*), if $(v_i, v_i + 1) \in E$ for i = 0, … , $k - 1$, and v_0 is the CBM's *Start* node and v_k is the CBM's *End* node.

Algorithm 1. Behavior Model Construction

```
Input:
    $CxWSDL$ document
Output:
    $G$: $Constraint\text{-}based\ Behavior\ Model$;
1: Parse $CxWSDL$ to get $service\ name$ ($sn$), valid time ($vt$), and $operation$ set ($OpSet$);
2: Initialize $G$, set $G.V \leftarrow \varnothing$ and $G.E \leftarrow \varnothing$;
3: Let $G.S_n \leftarrow sn$ and $G.D \leftarrow vt$;
4: Add a $Start$ node $start$, an $Initial$ node $init$, and an $End$ node $end$ to $G.V$;
5: Let $start \leftarrow preNode(init)$ and $init \leftarrow sucNode(start)$;
6: for each operation $op$ in $OpSet$ do
7:    Add $Request$ node $req$, set its attributes and constraints ;
8:    Add all $Response$ nodes to $resSet$, set their attributes and constraints ;
9:    for each node $res$ in $resSet$ do
10:       $res \leftarrow preNode(req)$, $req \leftarrow sucNode(res)$;
11:       if $Iteration$ = true then
12:          $req \leftarrow preNode(res)$, $res \leftarrow sucNode(req)$;
13:       end if
14:    end for
15: end for
16: for each node $n$ in $G.V$ do
17:    if $n.T = Request$ then
18:       if $preOp$ = null then
19:          $init \leftarrow preNode(n)$, $n \leftarrow sucNode(init)$;
20:       else
21:          Parse the $preOp$ constraint;
22:          Set the preceding and succeeding correlation between nodes;
23:       end if
24:    end if
25: end for
26: for each node $n$ in $G.V$ do
27:    for each $fnode$ in $n.Suc$ do
28:       Add $< n, fnode >$ to $G.E$ and set its attributes;
29:    end for
30: end for
```

Definition 7 (*Constraint Conflicting Sequence*). Let V and E be the node and edge sets of a CBM, respectively. A sequence of nodes $<v_0, \dots, v_k>$ is called a *Constraint Conflicting Sequence* (*CFS*), if there exists a $(v_i, v_i + 1) \notin E$, for i $= 0, \dots, k-1$.

For a large-scale application, there may be many services that collaborate with each other and thus a large number of operations are included in such services. It is impractical or even impossible to test all the possible event sequences or paths. Thus, we define three coverage criteria to control the number of the generated test sequences.

– *Request Node Coverage*, which requires that all nodes whose type is *Request* be covered at least once.
– *Response Node Coverage*, which requires that all nodes whose type is *Response* be covered at least once.
– *Edge Coverage*, which requires that all edges should be covered at least once.

As to the generation of *CPSs*, we employ an open source testing tool, *GraphWalker* [11]. GraphWalker provides a general model traversal strategy supporting the generation of test sequences that execute each of the elements in a given model. As to the generation of *CFSs*, we first parse the *Sequential Constraint* and *Repeated Invocation Constraint*, and then generate the sequence that violates these *Sequence Constraints*. If there is a *Sequence Constraint* for an operation, a *CFS* test sequence is generated for this operation. If the *Repeated Invocation Constraint* is false, we generate a sequence that invokes an operation repeatedly. If an operation has a *Sequential Constraint*, we generate a sequence that invokes the operation without including any preceding operations.

We propose Algorithm 2 for generating test sequences from a service's *CBM*, which has five major steps:

– *Initialization (lines 1–2)*: Initialize the Constraint Compliant Sequence set (Tss), the Constraint Conflicting Sequence set ($cTss$), the initial test sequence set ($initTss$), and the set of elements ($eleCoverSet$), and set the coverage criterion.
– *Set Coverage Criterion (lines 3–12)*: Based on the selected coverage criterion, traverse G to obtain the set of elements ($eleCoverSet$) to be covered.
– *Generate CPSs (lines 13–16)*: For each element ele in $eleCoverSet$, use *GraphWalker* to generate initial *CPSs* test sequences ($initTss$).
– *Remove Redundant Sequences (lines 17–25)*: For $initTss$, delete the redundant sequences and obtain the final *CPS* test sequences tss.
– *Generate CFSs (lines 26–36)*: Generate the *CFS* test sequence set $cTss$.

4.3 Test Case Generation

We first use the constraint solver tool *Z3* to generate the combinations of input parameter values that satisfy the constraints involved in each test sequence, then incorporate such parameter values into corresponding *SOAP* messages, and finally generate the executable test cases.

The executable test cases on each test sequence are derived from the behavior model and the decision table for the operations of a web service. The Decision Table (DT) is a triple $DT = <C, E, R>$, where the *Conditions* part (C) specifies a set of constraints on the input parameters that can be evaluated to true or false, the *Events* part (E) contains a set of response events related to the *Response* type nodes, the *Rules* part (R) denotes a specific value of any combination of the conditions and their corresponding execution events. For a parking fee service *PFC*, for example, R_3 in Table 1 means that if the *login_License* input parameter to operation *login* satisfies a regular expression (i.e., MATCH (*login_License*,

Algorithm 2. Test Sequence Generation

```
Input:
    $G$: *Constraint-based Behavior Model*;
    $gf$: a graphml file;
Output:
    $Tss$: *a CPS set*;
    $cTss$: *a CFS Set*;
 1: Let $initTss \leftarrow \varnothing$, $Tss \leftarrow \varnothing$, $cTss \leftarrow \varnothing$, and $eleCoverSet \leftarrow \varnothing$;
 2: Set a Coverage Criterion $cc$;
 3: if $cc$ = Request Node Coverage then
 4:     Parse $G$ to get ReqNodeSet whose element is a Request node;
 5:     $eleCoverSet \leftarrow ReqNodeSet$;
 6: else if $cc$ = Response Node Coverage then
 7:     Parse $G$ to get ResNodeSet whose element is a Response node;
 8:     $eleCoverSet \leftarrow ResNodeSet$;
 9: else
10:     Parse $G$ to get EdgeSet whose element is an Edge;
11:     $eleCoverSet \leftarrow EdgeSet$;
12: end if
13: for each element $ele$ in $eleCoverSet$ do
14:     Generate test sequence $ts$ which covers $ele$;
15:     Add $ts$ to $initTss$;
16: end for
17: while $eleCoverSet$ != $\varnothing$ do
18:     Get the maximum length $ts$ in $initTss$;
19:     Get the element set $eleTCoverSet$ which $ts$ covers;
20:     if $eleTCoverSet$ != $\varnothing$ then
21:         $eleCoverSet \leftarrow eleCoverSet$ - $eleTCoverSet$;
22:         $initTss \leftarrow initTss$ - $ts$;
23:         $Tss \leftarrow Tss + ts$;
24:     end if
25: end while
26: Parse $G$ to get ReqNodeSet whose element is a Request node;
27: for each node $n$ in $ReqNodeSet$ do
28:     if $n.Iteration$ = false then
29:         Generate test sequence $ts$ which repeated calls to $n$;
30:         Add $ts$ to $cTss$;
31:     end if
32:     if $n.preOp$ != null then
33:         Generate test sequence $ts$ which directly calls to $n$;
34:         Add $ts$ to $cTss$;
35:     end if
36: end for
```

[BJ][A-Y][0-9]{5}) and the *login_loginTime* input parameter is between 0 and 24, then the response event is *loginResponse_succ*. Thus, each rule of the *DT* defines a pre-condition of a *Response* node. Table 1 shows an example *DT* for the operation *login* of *PFC*, which has three rules, namely R_1, R_2, and R_3.

Table 1. Decision Table for a login operation

		Rules		
		R_1	R_2	R_3
Conditions	MATCH(*login_License*, [BJ][A-Y][0-9]{5}) == true	F	T	T
	0 <= *login_loginTime* <= 24	T	F	T
Events	*loginResponse_succ*			$\surd$
	loginResponse_fail	$\surd$	$\surd$	

We traverse all the nodes of a test sequence to get their associated constraints. For the *Request* node, we obtain the related input parameter name and type from *CBM* and convert these constraints into variable definitions of *Z3*. For the *Response* node, we first parse the decision table for the node, select the appropriate rule where the event is the target node. Then, we convert those conditions to assert commands of *Z3*. For example, the *loginResponse_fail* node has two rules (i.e. R_1 and R_2 in Table 1). We then run the constraint solver script to get the solution (combinations of input parameter values) that satisfies the constraints mentioned above. Finally, we combine the parameter value combinations with the test sequences to form the executable test cases.

The above process only considers which element should be covered, without taking into account the state of the transferred data when the element is executed. Therefore, we propose a set of test suite generation strategies based on these coverage criteria with or without considering the node state in the test sequence, namely: *ReqN-S* and *ReqN-NS* representing *Request Node Coverage* with and without considering the state of node, respectively; *ResN-S* and *ResN-NS* representing *Response Node Coverage* with and without considering the state of node, respectively; *E-S* and *E-NS* representing *Edge Coverage* with and without considering the state of node, respectively.

4.4 Test Execution

We execute the service under test with the generated test cases wrapped in *SOAP* messages. In our experiment, these SOAP messages are coded into a client script. During the execution, we monitor the invocations of the service operations from the client script and determine whether an invocation violates the constraints. If a violation is detected, we record the type of constraint violated by the test case. Finally, we check whether such violations are as intended by the test cases.

5 Evaluation

5.1 Research Questions

In this study, we aim to answer the following research questions:

RQ1 Can *CxWSDL* effectively describe all the presented behavior constraints?
To answer this question, we examine possible underlying restrictions and assumption in the experimental web services and evaluate whether *CxWSDL* is able to describe them.

RQ2 Can the proposed behavior constraint-based testing technique validate the behavior conformance of web services from a user perspective?
To answer this question, we evaluate whether our approach can effectively detect service invocations that violate the behavior constraints during execution as intended by the test cases, by comparing the expected and actual invocation violations.

RQ3 What is the difference between the test suites generated using different coverage criteria in terms of violation detection effectiveness?
To answer this question, we evaluate whether the test suites generated using different coverage criteria show different detection effectiveness of invocation violations.

5.2 Subject Programs

We choose three web services to evaluate the effectiveness of our technique. *Parking Fee Calculation* (*PFC*) calculates the parking fee according to the vehicle type (e.g. *motorcycle, van, coupe*), the parking day (whether weekend or workday), the parking time, and whether using a discount coupon. *Expense Reimbursement System* (*EXP*) assists the sales director of a firm in determining the fee to be charged to each senior sales manager or sales manager for any excessive mileage in the use of the company car, and in processing reimbursement requests regarding various kinds of expenses such as airfare, hotel accommodation, meals, and phone calls. *PostalMethods* (*PostalWS*) provides the service of mailing documents such as letters, invoices, notices, and contracts.

PostalWS is a real-world web service provided by *PostalMethods.com* (http://www.postalmethods.com/), while *PFC* and *EXP* are two web services developed based on real-world business specifications. In order to illustrate the diversity of constraints, we derived another variant for each of *PFC* and *EXP*, denoted as PFC^2 and EXP^2, respectively. PFC^2 considers an additional *Time Constraint*, and EXP^2 excludes the *Region Constraint*.

5.3 Result and Analysis

Following the process of specifying behavior constraints in *CxWSDL*, deriving behavior model, generating and executing test cases using *MDGen*, we have tested each of the subject web services and collected experimental data relevant to the three research questions. Due to space limitation, further details of the experiments can not be included in the paper.

Table 2. Summary of violation detection effectiveness

Services	Coverage strategy											
	ReqN-S		*ReqN-NS*		*ResN-S*		*ResN-NS*		*E-S*		*E-NS*	
	V	*TS*	*V*	*TS*	*V*	*TS*	*V*	*TS*	*V*	*TS*	*V*	*TS*
PFC	73	109	3	4	82	118	5	6	82	118	5	6
PFC$^{\mathbf{2}}$	109	109	4	4	118	118	6	6	118	118	6	6
EXP	21	21	3	3	41	41	6	6	209	209	6	6
EXP$^{\mathbf{2}}$	1	21	0	3	21	41	3	6	25	209	3	6
PosatlWS	7	26	5	12	85	102	13	20	85	152	13	23

(1) **Expressive power of *CxWSDL* for behavior constraints description.** We have analyzed each operation for *PFC*, PFC^2, *EXP*, EXP^2 and *PostalWS*, obtained the behavioral constraints of these experimental services. The different types of behavior constraints for the different experimental services are shown in the second column in Table 3.
We can see that the subject services cover all the six types of behavioral constraints discussed in Sect. 3. Furthermore, all these constraints are described in *CxWSDL* documents, which can be deployed and accessed in the same way as a WSDL document. In summary, the result shows that *CxWSDL* can adequately express the service behavior constraints proposed in this paper.

(2) **Behavior conformance.** After generating the behavior model and test sequences for each subject service, we generate test cases using six test case generation strategies. Each strategy and the number of test cases in the associated test suites are shown in Table 2. The test suites contain both constraint-conforming and constraint-violation test cases, where *V* refers to the number of test cases that detected violations and *TS* refers to the total number of test cases. In our experiment, once the *CxWSDL* document is obtained, it is easy to generate the *CBM* and test suite automatically using *MDGen*. The results show that our approach can detect all the improper invocations and can correctly locate the violations as determined by the types of service constraints being violated, as shown in Table 3.

(3) **Effectiveness of different coverage criteria.** Table 3 shows the violation detection effectiveness of the different coverage strategies. The number of related test cases generated using different coverage criteria are given in the third to eighth columns. The results show that the *Response Node* and *Edge* coverage criteria can cover more types of behavior constraints than the *Request Node* coverage criterion.

Table 3. Distribution of detected violation by different test case generation strategies

Services	Constraints	Coverage strategy					
		ReqN-S	*ReqN-NS*	*ResN-S*	*ResN-NS*	*E-S*	*E-NS*
PFC	*paraRestriction*	0	0	7	1	7	1
	preOp	36	1	36	1	36	1
	Iteration	37	2	37	2	37	2
	paraRelation	0	0	2	1	2	1
PFC²	*eTime*	108	3	110	4	110	4
	paraRestriction	0	0	7	1	7	1
	Iteration	1	1	1	1	1	1
EXP	*paraRestriction*	0	0	20	3	20	3
	invokeOp	12	1	3	1	9	1
	ipRegion	9	2	18	2	180	2
EXP²	*paraRestriction*	0	0	20	3	20	3
	invokeOp	1	0	1	0	5	0
PosatlWS	*paraRestriction*	0	0	52	7	52	7
	preOp	7	4	7	4	7	4
	Iteration	0	1	2	1	2	1
	paraRelation	0	0	24	1	24	1

6 Related Work

Many research efforts have been made to address the challenging issues of web services testing. We describe closely related work from the perspective of extensions to *WSDL* and model-based testing techniques.

6.1 Extensions to *WSDL*

A service description contains basic information as well as additional information, such as exceptions, operational semantics, and contractual conditions. Researchers have proposed extensions to *WSDL* with various purposes, such as testing and behavioral modeling.

For testing web services, Tsai et al. [15] proposed four types of extensions to *WSDL* (input-output dependency, invocation sequence, hierarchical functional description, and concurrent sequence specification) to support the description of dependencies. Similarly, Sneed et al. [14] extended *WSDL* with the pre-condition assertions, and Jiang et al. [8] extended *WSDL* using *Design-by-Contract* for precisely locating faults when the web service does not meet its requirements.

For modeling service behaviors, Sheng et al. [13] extended *WSDL* with *Semantic Markup for Web Service* (*OWL-S*) and *Web Service Semantics* (*WSDL-S*) to support the description of service behaviors. Bertolino et al. [3]

extended *WSDL* with *Protocol State Machine* to describe the prescribed ordering of operation invocations. Heckel et al. [6] extended *WSDL* with graph transformation rules to support the modeling of both the service's behavior and the client's requirements.

In this work, we have extended *WSDL* with constraints to support the description of restrictions or assumptions of service behaviors, and a formal language is provided for expressing common constraints. Such an extension provides the basis for testing the conformance of web services to their behavior constraints from a user perspective.

6.2 Model-Driven Testing of Web Services

Various models have been proposed for testing web services or their composites, such as *Finite State Machine* (*FSM*) [5,9,10], *Event Sequence Graph* (*ESG*) [1,4], and *Unified Modeling Language* (*UML*) [17,19].

Keum et al. [9] proposed to model web service behaviors with *Extended Finite State Machine* (*EFSM*) and generate test cases from the *EFSM* model to achieve a better test coverage. Endo et al. [5] proposed a model-based testing process for service-oriented applications, and *FSM* was used to model and support test case generation. Similarly, Kiran et al. [10] proposed an *FSM* model-based approach to testing composite services, which focuses on the test coverage required for testing the component services individually and their compositions.

Endo et al. [4] proposed an integrated testing strategy for web services, which first used *ESG* to model web services under test, then generated test cases from the *ESG* model, finally conducted a coverage analysis after the test case execution. Belli et al. [1] proposed a model-based approach to testing composite services, in which message exchanges in a web service were viewed as events modeled using *ESG*. These techniques mainly focus on structural testing of web services or their compositions without considering internal constraints on the invoked services.

Wu et al. [17] proposed a combination of *EFSM* and *UML* sequence diagram, called *EFSM-SeTM*, from which various coverage criteria are defined to test all possible scenarios. Similarly, Zhang et al. [19] proposed an extended *UML* activity diagram to model the behavior of *BPEL* service compositions, and defined coverage criteria on the model. These techniques focus on coverage testing of composite services, while ignoring behavior conformance of component web services.

In this work, we have proposed a model-driven approach to testing web services' conformance via behavior constraints from a user's perspective. The service behavior is modeled using *ESG* derived from constraints expressed in *CxWSDL*, and test cases are generated from the behavior model with respect to coverage criteria. Unlike the existing model-based testing approaches that mainly focus on test coverage of web services or their compositions, our approach focuses on the behavior conformance of web services, and connects the description of behavior constraints to service executions with executable test cases.

7 Conclusion

In this paper, we have proposed a constraint-based model-driven testing approach for testing the behavior conformance of web services. Our approach leverages constraints to provide more accurate descriptions of the behavior logic of web services and consequently enhances the testing of services through such behavior-based test case generation and execution. Experimental results have shown that our approach can effectively generate test cases and detect the service invocations that violate the service behavior constraints.

In future work, we plan to consider further types of constraints and carry out evaluations with more complex real-life web services.

Acknowledgment. This work is supported by the Beijing Natural Science Foundation of China under Grant No. 4162040, the National Natural Science Foundation of China under Grant No. 61872039, the Aeronautical Science Foundation of China under Grant No. 2016ZD74004, the Fundamental Research Funds for the Central Universities under Grant No. FRF-GF-17-B29, and China Postdoctoral Science Foundation under Grant No. 2017M620617.

References

1. Belli, F., Endo, A.T., Linschulte, M., Simao, A.: A holistic approach to model-based testing of web service compositions. Softw.: Pract. Exp. **44**(2), 201–234 (2014)
2. Belli, F., Linschulte, M.: Event-driven modeling and testing of web services. In: Proceedings of the 32nd IEEE International Computer Software and Applications Conference, pp. 1168–1173. IEEE CS (2008)
3. Bertolino, A., Polini, A.: The audition framework for testing web services interoperability. In: Proceedings of the 31st International Conference on Software Engineering and Advanced Applications, pp. 134–142. IEEE CS (2005)
4. Endo, A.T., Linschulte, M., Simao, A.D.S., Souza, S.R.S.: Event-and coverage-based testing of web services. In: Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion, pp. 62–69. IEEE CS (2010)
5. Endo, A.T., Simao, A.: Model-based testing of service-oriented applications via state models. In: Proceedings of the 8th IEEE International Conference on Services Computing, pp. 432–439. IEEE CS (2011)
6. Heckel, R., Mariani, L.: Automatic conformance testing of web services. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 34–48. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31984-9_4
7. Hou, K.J., Bai, X.Y., Lu, H., Li, S.F., Zhou, L.Z.: Web service test data generation using interface semantic contract. J. Softw. **24**(9), 2020–2041 (2013). (in Chinese)
8. Jiang, Y., Xin, G.M., Shan, J.H., Xie, B.: Research on a testing technology based on design-by-contract. J. Softw. **15**, 130–137 (2004). (in Chinese)
9. Keum, C.S., Kang, S., Ko, I.-Y., Baik, J., Choi, Y.-I.: Generating test cases for web services using extended finite state machine. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 103–117. Springer, Heidelberg (2006). https://doi.org/10.1007/11754008_7

10. Kiran, M., Simons, A.J.H.: Model-based testing for composite web services in cloud brokerage scenarios. In: Ortiz, G., Tran, C. (eds.) ESOCC 2014. CCIS, vol. 508, pp. 190–205. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14886-1_18
11. Kristian, K.: Graphwalker (2017). http://graphwalker.github.io
12. Micskei, Z.: Model-based testing (MBT) (2017). http://mit.bme.hu/~micskeiz/pages/mbt.html
13. Sheng, Q.Z., Maamar, Z., Yao, L., Szabo, C., Bourne, S.: Behavior modeling and automated verification of web services. Inf. Sci. **258**(3), 416–433 (2014)
14. Sneed, H.M., Huang, S.: WSDLTest - a tool for testing web services. In: Proceedings of the 8th IEEE International Workshop on Web Site Evolution, pp. 14–21. IEEE CS (2006)
15. Tsai, W.T., Paul, R., Wang, Y., Fan, C., Wang, D.: Extending WSDL to facilitate web services testing. In: Proceedings of the 7th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2002), pp. 171–172. IEEE CS (2002)
16. Wang, P.W., Ding, Z.J., Jiang, C.J., Zhou, M.C.: Constraint-aware approach to web service composition. IEEE Trans. Syst. Man Cybern. Syst. **44**(6), 770–784 (2017)
17. Wu, C.S., Huang, C.H.: The web services composition testing based on extended finite state machine and UML model. In: Proceedings of the 5th International Conference on Service Science and Innovation, pp. 215–222. IEEE CS (2013)
18. Xu, L., Chen, L., Xu, B.W.: Testing web services based on user requirements. J. Softw. **36**(6), 1029–1040 (2011)
19. Zhang, G., Mei, R., Zhang, J.: A business process of web services testing method based on UML 2.0 activity diagram. In: Proceedings of the Workshop on Intelligent Information Technology Application, pp. 59–65. IEEE CS (2007)