Generating Test Cases for Web Services Using Extended Finite State Machine

ChangSup Keum[1], Sungwon Kang[2], In-Young Ko[2],
Jongmoon Baik[2], Young-Il Choi[1]

[1] BcN Research Division,
Electronics and Telecommunications Research Institute
{cskeum, yichoi}@etri.re.kr
[2] School of Engineering,
Information and Communications University
{kangsw, iko, jbaik}@icu.ac.kr

Abstract. Web services utilize a standard communication infrastructure such as XML and SOAP to communicate through the Internet. Even though Web services are becoming more and more widespread as an emerging technology, it is hard to test Web services because they are distributed applications with numerous aspects of runtime behavior that are different from typical applications. This paper presents a new approach to testing Web services based on EFSM (Extended Finite State Machine). WSDL (Web Services Description Language) file alone does not provide dynamic behavior information. This problem can be overcome by augmenting it with a behavior specification of the service. Rather than domain partitioning or perturbation techniques, we choose EFSM because Web services have control flow as well as data flow like communication protocols. By appending this formal model of EFSM to standard WSDL, we can generate a set of test cases which has a better test coverage than other methods. Moreover, a procedure for deriving an EFSM model from WSDL specification is provided to help a service provider augment the EFSM model describing dynamic behaviors of the Web service. To show the efficacy of our approach, we applied our approach to Parlay-X Web services. In this way, we can test Web services with greater confidence in potential fault detection.

1. Introduction

A Web service is any service available on the Internet that uses a standardized XML messaging system and is not tied to a operating system or programming language. In other words, Web service is a collection of components that are wrapped with SOAP (Simple Object Access Protocol) interfaces so they can exchange XML-based (Extensible Markup Language) messages [1]. Using Web Services, companies can integrate existing business applications into new and innovative business applications, publish them as services, discover and subscribe to other services, and exchange information [2].

Some testing techniques that are used to test software components are being extended to Web services. A few papers have presented testing techniques for Web services, but the dynamic discovery and invocation capabilities of Web services bring up many testing issues. Existing Web service testing methods try to take advantage of syntactic aspects of Web service rather than semantic, dynamic, and behavioral information because standard WSDL is not capable of containing such information. Therefore, they focused on testing of single operations rather than testing sequences of operations. Furthermore, they heavily rely on the test engineers' experience.

In this paper, we propose a new approach to test Web services. This idea stems from similarities between communication protocol testing and stateful Web services testing. Web services can be either stateless or stateful. Stateful Web services have several operations which affect the service's state that are used by other operations. Operations in stateless Web service do not change the service's internal states. Each operation in Web services has a request and response message with parameters. It is hard to test such Web services because they are distributed applications with numerous runtime behaviors that are different from typical applications. Service consumers usually have to use black-box testing because specifications are available but design and implementation details of Web services are not available. The specification is written in WSDL (Web Services Description Language). Unfortunately, current WSDL does not contain sufficient information for a consumer to test the available Web services. Although a few technologies exist to verify syntactic aspects of the interactions, it is very difficult to find out whether Web services behave correctly with all possible messages.

Specifically, protocol testing and Web services testing both require to perform some message exchanges and to analyze the result. Furthermore, it is more important to test sequences of messages than to test of single message. Also these two testing methods are basically based on the black-box approach. In black-box testing, specification has a strong influence on testing. Stateful Web services have reactive characteristics similar to communication protocols; therefore specification languages for Web services are favored which precisely define the temporal ordering of interactions. FSM (Finite State Machine) model is often used for defining the temporal order of interaction. However, the FSM model is often too restrictive for defining all aspects of a Web service specification because a Web service has input and output messages with data parameters. In contrast with FSM, EFSM [3] includes additional variables, input and output events including parameters. It consists of transitions which are characterized by a so-called enabling predicate and a transition action. Therefore an extended FSM model seems to be a very promising model for describing Web services behaviors.

We utilize the EFSM model to test Web services. Since current WSDL does not contain sufficient information for a test engineer to test the available Web services, temporal ordering information is added to describe Web services behaviors. EFSM (Extended Finite State Machine) is well suited for describing Web services behavior because it has the control part of the specification represented by pure FSM model and the data part represented by the transition predicates and actions.

There are many benefits to constructing test cases on the basis of a formal model specification such as EFSM. The benefits arise from the ability to precisely describe

and reason about potential faults. In particular, it means that test can be applied uniformly, with greater confidence in their fault detecting potential, and with the possibility of full automation. Using an EFSM formal specification for a Web service, we can generate test cases from the specification automatically if we are equipped with an appropriate tool set such as EFSM analyzer, test case generator, and monitor.

The remainder of the paper is organized as follows. After reviewing existing Web service testing methods in Section 2, we present a procedure from a WSDL specification to an EFSM model and introduce test case generation algorithm using EFSM in Section 3. An application example is provided to show the efficiency of our method in Section 4. We conclude the paper with a discussion of future work in Section 5.

2. Related Works

In this section, we review various methods for test cases generation for Web services and discuss drawbacks of existing Web service testing.

Heckel and Mariani [4] generate test cases for Web services with individual rules by selecting "likely" inputs. Possible inputs are further restricted by the preconditions of the GT (graph transformation) rules [5]. This suggests the derivation of test cases using a domain-based strategy, known as partition testing [6]. The idea is to select test cases by dividing the input domain into subsets and choosing one or more elements from each domain [7]. The execution of an operation can alter parts of service's state that are used by other operations. GT rules specify state modifications at a conceptual level. By analyzing these rules we can understand dependencies and conflicts between operations without inspecting their actual implementation. In this method, data-flow testing technique is used to test the interaction among production rules if creation of nodes and edge is interpreted as "definition" and deletion as "use" [8]. Conceptually, each operation (rule) can add or remove nodes and edges to or from the conceptual state, and change the values of attributes. Authors expect sequences of operations, which include the creation of an entity and its subsequent uses are likely to expose (state-based) fault.

In short, this method applies existing domain-based testing (partitioning testing) to the GT rules to generate test cases which cover validation of both single operation and sequences. The major problem of this method is that the definition of GT rules does not contain the temporal aspects (control flow) of message interactions. This method only considers data-flow to generate test cases for sequences of operations. This means that [4] has no test criteria for control flow. Furthermore, splitting the input domain into subsets relies on the tester's experience. This could cause non-uniform and biased tests for Web services.

In the paper [9], data perturbation is used as main method for testing Web service components. The testing process operates by modifying request messages, retransmitting messages, and analyzing the response messages for correct behavior. To do this process, value data perturbation modifies values in SOAP messages in terms of the types of the data. Data value perturbation relies on ideas from boundary value testing [10]. Test cases are derived from default boundary values of XML schemas. Tests are

created by replacing each value with each boundary value, in turn, for appropriate type.

Concisely, the authors present a new approach to testing Web services based on data perturbation. Data perturbation uses two methods to test Web services: data value perturbation and interaction perturbation. However, this approach relies strictly on syntactic information about the XML messages, does not use behavior information. They consider only the selection of appropriate input parameter values. The sequences of operations in Web service are not considered. They just focus on testing of single operation of Web service.

Li et al. [11] provide some techniques for various kinds of Web Services testing such as unit testing, functional testing, performance testing, Load/stress testing, security testing and authorization testing. They provide detailed information on the key aspects of Web service testing features related with performance, authorization, and security. Furthermore, they designed an automatic testing tool including SOAP-based log analysis, script generator, recorder, and monitor. However, there is no detailed information on the method of test cases generations in their paper.

In the paper [12], the authors propose a method of extending WSDL to describe dependency information which is useful for Web service testing. They suggest several extensions such as input-output dependency, invocation sequences, hierarchical functional description, and concurrent sequence specification. Similar to [11], there is no test case generation method and experimental data using the extension.

In summary, the existing Web service testing methods try to take advantage of syntactic aspects of Web service rather than behavioral aspects of Web services because standard WSDL does not contain such information. Therefore, they focused on the test of single operations instead of sequences of operation. One of disadvantages using those methods is that they rely on test engineer's experience. This could lead to non-uniform and biased testing. All these problems can be solved by augmenting behavior information to WSDL file. The behavior information holds control and data dependencies of Web service operations because the information is represented as an EFSM formal model. Using the augmented EFSM model, we can generate test cases which cover control and data paths thoroughly. In the next section, we describe our approach in detail.

3. Test cases generation for Web services using EFSM

In this section, we describe our test generation approach for Web services in detail. In Section 3.1, we first give a procedure for deriving an EFSM model from a WSDL specification of a service and illustrate the procedure with a banking Web service example. Once an EFSM model is constructed, test cases can be generated easily using a well-known algorithm as described in Section 3.2.

3.1 Modeling Web Service with EFSM

A WSDL specification is used to describe how to access a Web service and what operations it can perform. However, a WSDL specification does not provide sufficient information for Web service test derivation because it only provides the interface for the service. An EFSM starts from an initial state and moves from one state to another through interactions with its environment. The EFSM model extends the FSM model with variables, statements and conditions. An EFSM is a 6-tuple <*S, s_0, I, O, T, V*>, where *S* is a non-empty set of states, s_0 is the initial state, *I* is an non-empty set of input interactions, O is a non-empty set of output interactions, *T* is a non-empty set of transitions, and *V* is a set of variables. Each element of *T* is a 5-tuple of the form: <source_state, dest_state, input, predicate, compute_block>, where "source state" and "dest state" are states in S corresponding to the starting state and the target state of *t*, respectively; "input" is either an input interaction from *I* or empty; "predicates" is a predicate expressed in terms of variables in *V*, the parameters of the input interaction and some constants, and "compute-block" is a computation block consisting of assignment and output statements. We will only consider deterministic EFSMs that are completely specified. In addition, the initial state is always reachable from any state with a given valid context.

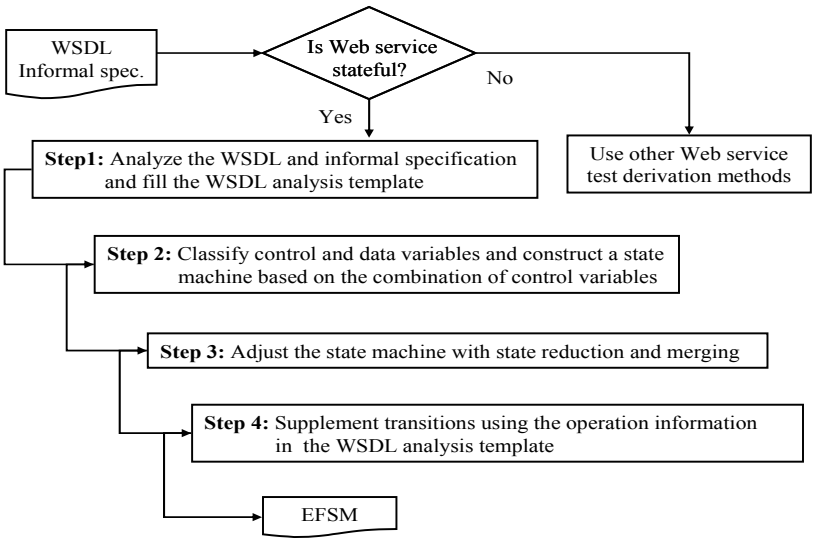


Fig. 1. Procedure for deriving an EFSM model from a WSDL description of a service

Figure 1 presents our procedure for deriving EFSM model from a WSDL specification. First of all, we have to decide whether the Web service to be modeled is stateful or not. A Stateful Web service in general can be modeled as an EFSM. Stateful Web service has several operations which change the service's internal state that

are used by other operations. In that case, the operations may response with different output messages according to the internal state of Web service server. If the Web service is stateless, then we have to use other Web service testing methods such as [4] and [9]. Otherwise, we continue with Steps 1 through 4.

Step 1) We analyze the WSDL specification and the web service specification in informal language and fill the WSDL analysis template shown in Table 1. Each row of Table 1 describes an operation with its name, its parameter types and its return value type together with its pre-condition and post-condition for each operation in WSDL specification.

For example, Table 2 shows the WSDL analysis template filled out for a banking Web service. From WSDL description, we find out that the banking Web service provides four public operations, i.e. *openAccount*, *deposit*, *withdraw*, and *closeAccount*. The operation openAccount expects a single parameter *init* which means an initial deposit, and returns an account number *identifier*. The operation *closeAccount* expects a single parameter *id,* which means account number, and returns the result of operation such as *ResultOK* and *Error*. The operations *deposit* and *withdraw* expect two parameters *id* (identifier) and *v*(value), and return results such as *ResultOK* and *Error*. In Table 2, *value* holds the balance of the bank account created by openAccount operation and *accountId* means account number.

Table 1. WSDL analysis template

operation	pre-condition	post-condition
name: parameter: return:	…	…
…	…	…

Table 2. WSDL analysis template for banking Web service

operation	pre-condition	post-condition
name: openAccount parameter: init return: identifier	init > 0	value' = init accountId > 0
name : deposit parameter: id, v return : res	accountId = id v > 0	value' = value + v accountId > 0
name : withdraw parameter: id, v return : res	accountId = id value >= v	value' = value - v accountId > 0
name : closeAccount parameter: id return : res	accountId = id	accountId = 0 *f* value' = 0

Step 2) To construct EFSM, it is necessary to classify variables in the pre-condition and post-condition of Table 2 into control variables and data variables. Then a combination of different values of the control variables makes a state of the EFSM under construction. For the banking Web service example, there are two control variables *accountId* and *value*. Table 3 presents the classification of variables for banking Web service.

Table 3. Classification of variables for banking Web service

operation	pre-condition		post-condition	
	control variable	**data variable**	**control variable**	**data variable**
name: openAccount parameter: init return: identifier	-	init	accountId value	init
name : deposit parameter: id, v return : res	accountId	v id	value	-
name : withdraw parameter: id, v return : res	accountId value	v id	value	-
name : CloseAccount parameter: id return : res	accountId	id	accountId value	-

Figure 2 shows an initial version of EFSM for the banking Web service. The states are constructed by combining possible value range of control variables. The variable *accountId and value* have two possible values: range 0 and greater than 0. If the control variables have value 0, it means that it is not initialized yet. When the variable *accountId* is initialized by *openAccout* operation, the variable has a value greater than 0 until it is closed by *closeAccout* operation. After initialization, the variable value keep a balance greater than 0 according to the operation *withdraw* and *deposit*. Therefore, we make four different states with combinations of the two control variables. Then we associate transitions with the appropriate operations by examining the pre-condition and post-condition of an operation.

Step 3) It is desirable to reduce states in the initial version of EFSM model because first often the number of states would be otherwise huge and second there is a possibility that unreachable states may exist. For example, the state with *value* >0 and *accountId* = 0 is an unreachable state. Unreachable states should be deleted for the state reduction. Some states could be merged into one state according to test engineer's judgment. Figure 3 gives an enhanced EFSM obtained by removing an unreachable state and merging two states into a state named *Active*. For human readability, we assign a meaningful name to each state.

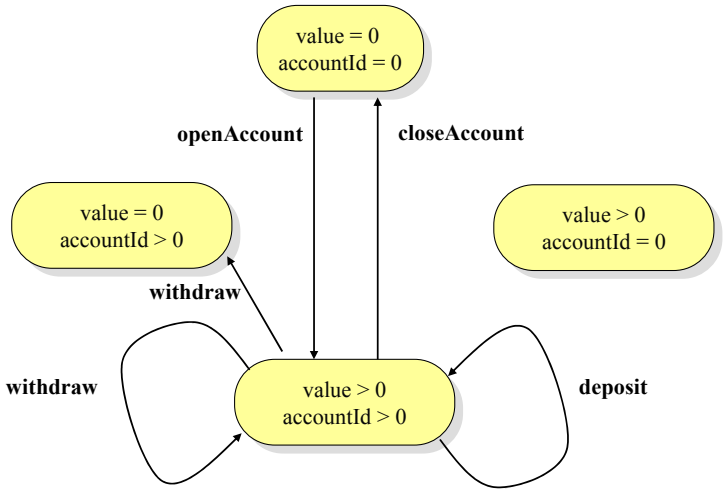


Fig. 2. EFSM construction with control variables

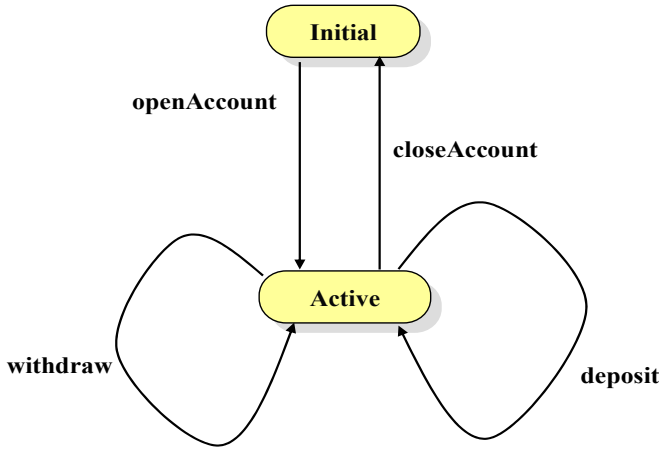


Fig. 3. Enhanced EFSM with state reduction and merging

Step 4) To make a concrete transition in EFSM, operation information in the WSDL is used. An operation has input and output message. Input message is transformed into input event and output message is transformed into output event in the transition. Pre-condition is transformed into guard condition in the transition. Post-condition is transformed into actions in the transition. Figure 4 shows our final EFSM model derived from the WSDL specification for the banking Web service.

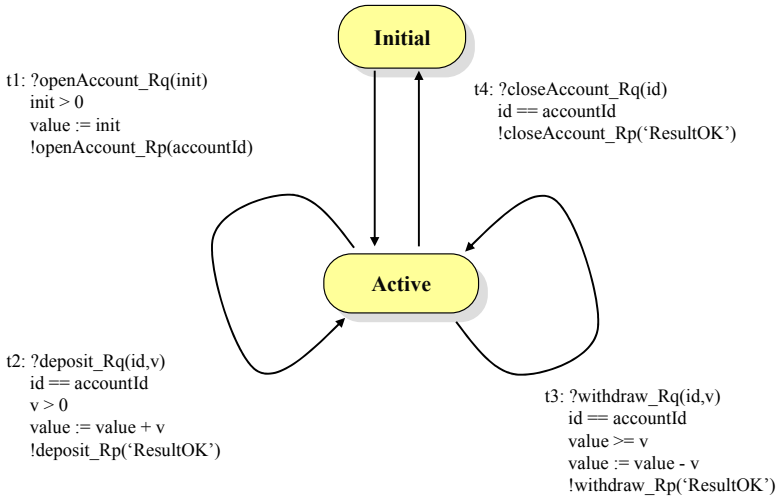


Fig. 4. Final EFSM for banking Web service

3.2 Test Cases Generation Algorithm using EFSM

In the paper [3], the authors provide a comparison of single EFSM-based test generation methods. We choose Bourhfir's algorithm [13] as our test case generation method for Web services because the algorithm considers both control and data flow with better test coverage. The control flow criterion used is UIO (Unique Input Output) sequence [14] and the data flow criterion is "all-definition-uses" criterion [15] where all the paths in the specification containing a definition of a variable and its uses are generated. Moreover, the algorithm uses a technique called cycle analysis to handle executability of test cases.

The detailed algorithm is described in Figure 5. For each state S in the EFSM, the algorithm generates all its executable preambles (a preamble is a path such that its first transition's initial state is the initial state of the system and its last transition's tail state is S) and all its postambles (a postamble is a path such that its first transition's start state is S and its last transition's tail state is the initial state). To generate the "all-definition-uses" paths, the algorithm generates all paths between each definition of a variable and each of its uses and verifies if these paths are executable, i.e., if all the predicates in the paths are true. After the handling executability problem, the algorithms removes the paths which is included in the already existing ones, completes the remaining paths (by adding postambles) and adds paths to cover the transitions which are not covered by the generated test cases.

```
Algorithm Extended FSM Test Generation
Begin
  Generate the dataflow graph G form the EFSM specification
  Choose a value for each input parameter influencing the control flow
  Call Executable-Du-Path-Generation(G) procedure
  Remove the paths that are included in already existing ones
  Add a postamble to each du-path to form a complete path
  Make it executable for each complete path using cycle analysis
  Add paths to cover the uncovered transitions
  Generate its input/output sequence using symbolic evaluation
End.

Procedure Executable-Du-Path-Generation(flowgraph G)
  Begin
  Generate the shortest executable preamble for each transition
  For each transition T in G
    For each variable v which has an A-Use in T
      For each transition U which has a P-Use or a C-Use of v
        Find-All-Paths(T,U)
      EndFor
    EndFor
  EndFor
End;
```

Fig. 5. Test case generation algorithm using EFSM

The following definitions that appeared in the paper [3] were used in the algorithm:

- A transition has an assignment-use (A-Use) of variable x, if x appears at the left-hand side of an assignment statement in the transition.
- When a variable x appears in the input list of a transition, the transition is said to have an input-use (I-Use) of variable x.
- A variable x is a definition (referred to as def), if x has an A-use or I-use.
- When a variable x appears in the predicate expression of a transition (Provided Clause), the transition has a predicate-use or P-Use of variable x.
- A transition is said to have a computational-use or C-use of variable x, if x occurs in an output primitive or an assignment statement at the right-hand side.
- A path $(t_1,t_2,\ldots,t_k,t_n)$ is said to a def-clear-path with respect to (w.r.t) a variable x if $t_2,\ldots,t_k$ do not contain defs of x.
- A path $(t_1,\ldots,t_k)$ is a Du-path (definition-uses) w.r.t a variable x, if $x \in$ def (t_1) and either $x \in$ c-use(t_k) or $x \in$ p-use(t_k), and $(t_1,\ldots,t_k)$ is a def-clear-path w.r.t x from t_1 to t_k.

In Table 4 shows a part of test cases and test sequences without input parameters for the EFSM in Figure 5.

Table 4. Test cases for the banking Web service

No	Test Cases	Input/Output Sequence
1	t1, t4	?openAcount_Rq!openAccount_Rp ?closeAccountRq !closeAccount_Rp
2	t1,t2,t4	?openAcount_Rq!openAccount_Rp ?deposit_Rq!deposit_Rp ?closeAccountRq !closeAccount_Rp
3	t1,t3,t4	?openAcount_Rq!openAccount_Rp ?withdraw_Rq!withdraw_Rp ?closeAccountRq !closeAccount_Rp
4	t1,t3,t2,t4	?openAcount_Rq!openAccount_Rp ?withdraw_Rq!withdraw_Rp ?deposit_Rq!deposit_Rp ?closeAccountRq !closeAccount_Rp
5	t1, t2, t3, t4	?openAcount_Rq!openAccount_Rp ?deposit_Rq!deposit_Rp ?withdraw_Rq!withdraw_Rp ?closeAccountRq !closeAccount_Rp

4. Application to Parlay-X Web services

To show that our method can be effectively used for nontrivial real world problems, we applied it to Parlay-X Web services [16]. Parlay-X is a Web Services framework for telecommunications domain. The architecture of the framework in which Parlay-X Web services operate is shown in Figure 6. A Parlay-X Web service, *Third Party Call*, is used to create and manage a call initiated by an application. The overall scope of this Web service is to provide functions to application developers to create a call in a simple way. Using the *Third Party Call* Web service, application developers can invoke call handling functions without detailed telecommunication knowledge. The *Third Party Call* Web service provides four operations: *MakeCall*, *GetCallInformation*, *EndCall*, and *CancelCall*.

For comparison, we generated test cases for the *Third Party Call* Web service with three different methods, i.e. the method of Heckel et al [4], the method of Offtut et al [9] and finally our method. For the method of Heckel et al [4], we defined a domain based on GT production rules. Eight production rules for the four operations were found. After that, we found attributes for each production rule. Test cases are generated by fixing a boundary value for at least one of them and randomly generating the other two values. In addition, we generated test cases using incorrect inputs for each rule. The sequences of operations are generated by analyzing dependencies and conflicts of operations. Finally, 36 test cases were generated using this method. For the method of Offtut et al [9], 40 test cases were generated through the analysis of boundary values of message parameters.

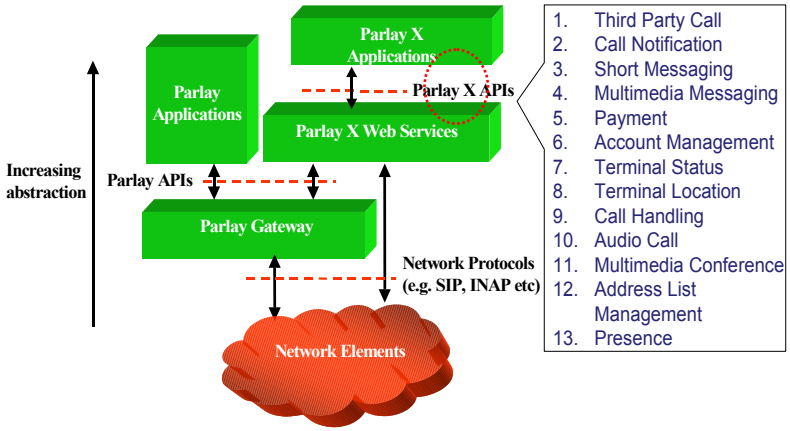
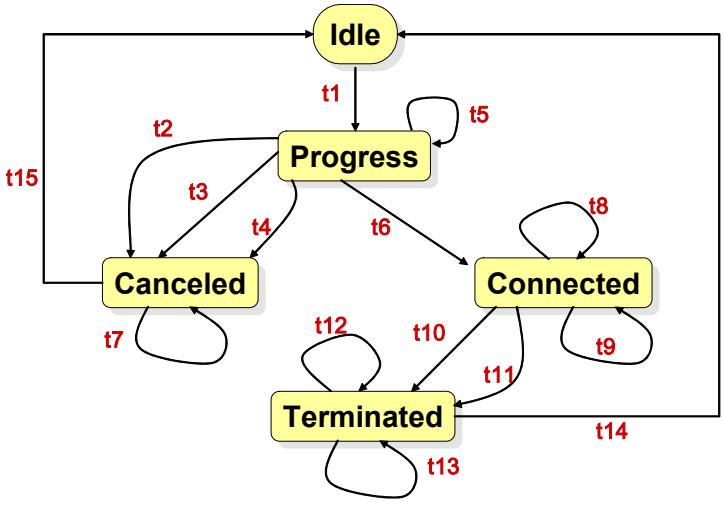


Fig. 6. Architecture of Parlay-X Web services

To generate test cases using our method, we followed the procedure described in Section 3.1. First, we analyzed the WSDL specification of *Third Party Call* and the informal specification of the *Third Party Call* Web service. For Step 2, three control variables were identified by analyzing the WSDL analysis template. Then we constructed an EFSM based on these three control variables and the four operations. The final EFSM shown in Figure 7 has five states and fifteen transitions. Using the EFSM and the algorithm described in Section 3.2, 95 test cases were generated for *Third Party Call.* Table 5 shows some of the test cases for *Third Party Call* Parlay-X Web service.



Transition	Input/Output/Computation
t1	?MakeCall_Rq(cgNum,cdNum) callId := GenerateCallId() !MakeCall_Rp(callId) status := Initial
t2	?CancelCall_Rq(id) id == callId status := Canceled set timer
t3	?EndCall_Rq(id) id == callId status := Canceled set timer
t4	?NoAnswer id == callId errCode := SVC0001 !ServiceError(id, errCode) status := Canceled set timer
t5	?GetCallInformation_Rq(id) id == callId !GetcallInformation_Rp(status)
t6	?CallConnected status := Connected
t7	?GetCallInformation_Rq(id) id == callId !GetcallInformation_Rp(status)
t8	?GetCallInformation_Rq(id) id == callId !GetcallInformation_Rp(status)
t9	?CancelCall_Rq(id) id == callId errCode := SVC0260 !ServiceError(id, errCode)
t10	?CallTerminated status := Terminated set timer
t11	?EndCall_Rq(id) id == callId status := Terminated set timer
t12	?GetCallInformation_Rq(id) id == callId !GetcallInformation_Rp(status)
t13	?EndCall_Rq(id) id == callId errCode := SVC0261 !ServiceError(id, errCode)
t14	expire_timer
t15	expire_timer

Fig. 7. EFSM model for the third party call Web service

Table 5. Test cases for Parlay-X Web service *Third Party Call*

No	Test cases
1	?MakeCall !CallId, ?GetCallInformation !CallStatus ?CallConnected ?CancelCall !ServiceError ?GetCallInformation !CallStatus ?CallTerminated ?TimeOut
2	?MakeCall !CallId ?CallConnected ?CancelCall !ServiceError ?CallTerminated ?TimeOut
3	?MakeCall !CallId ?GetCallInformation !CallStatus ?CallConnected ?CancelCall !ServiceError ?CallTerminated ?TimeOut
4	?MakeCall !CallId ?CallConnected ?GetCallInformation !CallStatus ?CancelCall !ServiceError ?CallTerminated ?TimeOut

5	?MakeCall !CallId, ?GetCallInformation !CallStatus ?CallConnected ?GetCallInformation !CallStatus ?CancelCall !ServiceError ?CallTerminated ?TimeOut

A test suite is a set of test cases and is said to satisfy a coverage criterion if for every entity defined by coverage criterion, there is a test case in the test suite that exercises the entity. Each method used in our experiment had its own test coverage criterion. The comparison of test coverage criterion for three methods is summarized in Table 6.

Table 6. Comparison of test criteria

	Data flow criterion	Control flow criterion
Method of Heckel et al [4]	all-definitions-uses	-
Method of Offtut et al [9]	-	-
Our method	all-definitions-uses	UIO sequence

The method [9] had no test coverage criterion, but we could generate test cases easily through examining types of message parameters. There is a trade-off in choosing test coverage criteria. The program could be more thoroughly tested with the stronger criterion. However, usually the cost incurred by test cases generation and testing is negligible compared to the cost incurred by the presence of faults in programs.

Test cases and results of different methods are summarized in Table 7. As we expected, our method located more faults than the other methods even though it spent more time for executing a test case. Our method spent more time than other method because test cases generated using our method consist of the complex sequences of operations but almost all test cases generated using other method is made of a single operation. To show the efficacy of our method, the number of test cases and the accumulated number of faults detected are analyzed in Figure 8. As shown in Figure 10, our method detected many faults in the early phase of testing. Our methods detected many errors that occurred during executing complex sequences of operations. For example, the operation *GetCallInformation* worked well in the initial state and the progress state, but the operation caused an error when it executed in the connected state. The method [4] located some faults related with boundary value and incorrect input values in the case of testing for single operations. However, the sequences of operations derived from the method [4] were not effective for locating faults. Even if the method [4] expected the data flow coverage criterion “all-definitions-uses” for generated test cases, the generated test cases using relations of conflicts and casual dependencies between productions rules did not find out any faults which were located by our method. During testing using the method [9], it was difficult to find faults because faults rarely occurred when we executed single operations with different boundary values. Only two faults related with message parameter value with maximum length were founded.

Table 7. Test cases and results

	Method of Heckel et al [4]	Method of Offutt et al [9]	Our method
Number of test cases generated	36	40	95
Number of faults found	5	2	18
total execution time (sec.)	90	80	859
average execution time (sec.)	2.5	2	9

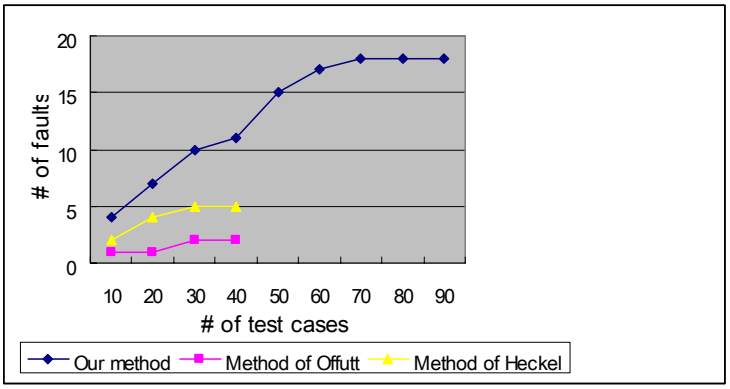


Fig. 8. Number of test cases and number of faults found

5. Conclusion

In this paper, we presented a new test cases generation method for Web services. The key idea is to augment a WSDL specification with an EFSM model that precisely describes the dynamic behavior of the service specified in the WSDL specification. Generally speaking, modeling an EFSM for a Web service is not a trivial task. To make this task easy and systematic, we suggested a procedure to derive an EFSM model from WSDL description of a service.

In summary, the main contributions of this paper are as follows: First, this paper introduces a new Web service testing method that augments WSDL specification with an EFSM formal model and applies a formal technique to Web service test generation. Second, using the EFSM based approach, we can generate a set of test cases with a very high test coverage which covers both control flow and data flow. Third, we applied our method to an industry level example and showed the efficacy of our method in terms of test coverage and fault detection.

One of drawbacks of our approach is the overhead to generate test cases based on an EFSM. Even if we suggest a procedure to derive an EFSM model from a WSDL specification, it may require additional jobs besides Figure 1 to complete a fully described EFSM in case of very complicated WSDL files. The algorithm described in Section 3.3 is also a heavy-weight algorithm. Without any automatic tool for generating test cases using EFSM, it is a very tedious task to generate test cases manually.

In this paper, we focused on testing of a Web service with single EFSM derived from a WSDL specification. For future work, we plan to extend our method to treat more complex situations such as test cases generation for compositions of Web services.

References

1. E. Cerami, Web Services Essentials, O'Reilly, 2002.
2. D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, Web Services Architecture. W3C working group note, W3C, 2004.
3. C. Bourhfir, E.Aboulhamid, F.Khendek, and R.Dssouli, "Test cases selection from SDL specifications," Computer Networks 35(6), pp.693-708, 2001.
4. R. Heckel and L. Mariani, "Automatic Conformance Testing of Web Services," FASE 2005, LNCS 3442, pp. 34 – 48, 2005.
5. P. Baldan, B.Konig, and I.Sturmer, "Generating test cases for code generators by unfolding graph transformation systems," Proc. 2nd Intl. Conference on Graph Transformation, Rome, Italy, 2004.
6. L. White and E. Cohen, "A domain strategy for computer program testing." IEEE Transactions on Software Engineering 6, pp. 247–257, 1980.
7. E. Weyuker and B. Jeng, "Analyzing partition testing strategies," IEEE Transactions on Software Engineering 17, pp. 703–711, 1991.
8. S. Rapps, and E. Wejuker, "Data flow analysis techniques for program test data selection," 6th Intl. Conference on Software Engineering. pp. 272–278, 1982.
9. J. Offutt and W. Xu, "Generating Test Cases for Web Services Using Data Perturbation," ACM SIGSOFT SEN, 2004.
10. B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990.
11. Y. Li, M. Li, and J. Yu, "Web Service Testing, the Methodology, and the Implementation of the Automation-Testing Tool," GCC2003, LNCS 3032, pp.940-947, 2004.
12. W.T.Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, "Extending WSDL to Facilitate Web Services Testing," HASE 2002, 2002.
13. C. Bourhfir, R. Dssouli, E.Aboulhamid, and N.Rico, "Automatic executable test case generation for EFSM specified protocols," IWTCS'97, pp.75-90, 1997.
14. K.Sabnani and A.Dahbura, "A new Technique for Generating Protocol Tests," ACM Comput. Commun. 15(4), 1985.
15. Weyuker, E.J. and Rapps, S., "Selecting Software Test Data using Data Flow Information", IEEE Transactions on Software Engineering, April, 1985.
16. Parlay X Working Group, Parlay-X White Paper, http://www.parlay.org, 2002.