The Web Services Composition Testing Based on Extended Finite State Machine and UML Model

Ching-Seh Wu
Computer Science and Engineering
Oakland University
Rochester, Michigan, United States
cwu@oakland.edu

Chi-Hsin Huang
Computer Science and Engineering
Oakland University
Rochester, Michigan, United States
chuang2@oakland.edu

***Abstract*—Web services are designed as software building blocks for Service Oriented Architecture (SOA). It provides an approach to software development that system and application can be constructed by assembling reusable software building blocks, called services. The industries have adopted web services composition to generate new business applications or mission critical services. One of the most popular integration languages for web services composition is Web Services Business Process Execution Language (WS-BPEL). Although the individual service is usually functional correctly; however, several unexpected faults may occur during execution of composite web service. It is difficult to detect the original failure service because the faults may propagate, accumulate and spread. In this paper, we present a technique of Model-Based Testing (MBT) to enhance testing of interactions among the web services. The technique combines Extended Finite State Machine (EFSM) and UML sequence diagram to generate a test model, called EFSM-SeTM. We also defined various coverage criteria to generate valid test paths from EFSM-SeTM model for a better test coverage of all possible scenarios.**

Keywords—Composite Web Service Testing, Web Services Test Model, WS-BPEL, EFSM, MBT

I. INTRODUCTION

SOA is an architectural manner that designs and develops software in the form of rapid, low-cost, loosely-coupled and easy integrated in heterogeneous environments. Web services are a concrete implementation of the SOA and they have been developed by using open standards such as SOAP, WSDL, and UDDI based on XML. These atomic services are reusable and distributed over the Internet, they can be assembled to construct new and complicated business applications. The web services composition uses a standard integration language, called WS-BPEL, which is generally adopted by industries to specify the execution flow of business process with web services and define how multiple services interact with each other to achieve the business goal. Since composite web service has been created for mission critical services and complex business processes, the composite web service testing is essential to ensure the execution of whole business process is consistent and guarantee a higher service quality and reliability.

For higher service quality and reliability, model-based techniques have been applied to build a test model that describes relevant sequence of the test object [1], [2], [3], [4], [5]. By using techniques of MBT, the test cases can be derived efficiently from the test models and support the testing activity. Tester can easily update the model and regenerate the test suite for changed requirements, avoiding error-prone manual changes.

Andre Takeshi Endo et al. [4] proposed an event- and coverage-based testing approach for web services. They designed the Event Sequence Graphs (ESGs) for test case generation and utilized the JaBUTi structural testing tool for measuring code coverage. The ESGs is able to describe user interactions in a simplified way and JaBUTi implements control-flow and data-flow testing coverage criteria. The testing coverage can be measured for each test case and for all test cases. Although ESGs enable to describe user interactions, it ignores the internal state of a system under test. Andre Takeshi Endo and Adenilso Simao [5] proposed a MBT process for service-oriented applications using state models. They defined three types of artifacts (legacy artifacts, manual artifacts, and generated artifacts) and introduce supporting tools in the proposed testing process. According to identified test scenario from tester, the Mealy's FSM is used to build the state model which represents the behaviors of test scenario. The test cases are derived from the state model and selected criteria. Although the FSM is utilized to build the state model, it doesn't include additional variables, parameters of input and output, enabling precondition and action of transitions. ChangSup Keum et al. [6] and Abdul Salam Kalaji et al. [7] proposed approaches to test web services based on EFSM. Since EFSM extends the FSM with a set of variable, it is able to model both control flow and data flow. Therefore, the transition can have precondition over the input and the machine's variables can have assignments to these variables. By appending EFSM to standard WSDL, they can generate test cases which have a better test coverage. Although they used EFSM to build the test model, it doesn't consider the composite web service.

In this paper, we proposed an EFSM-Sequence Test Model (EFSM-SeTM) which is combined EFSM with sequence diagram for composite web service testing (as shown in Figure 1). The EFSM of a service consists of a set of finite states and transitions among the states. Also EFSM describes its states and the messages it can receive in those states. The sequence diagram is derived from WS-BPEL which describes the

978-0-7695-4985-9/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSSI.2013.46

execution sequence of services and defines the interaction between multiple services. In the sequence diagram, the objects represent the services and the messages represent the requests of services. Therefore, the EFSM-SeTM will include the information of the services execution sequence, services behavior and the services internal state under system test. Based on the EFSM-SeTM, we define coverage criteria to generate test paths which have a better test coverage of all possible scenarios.

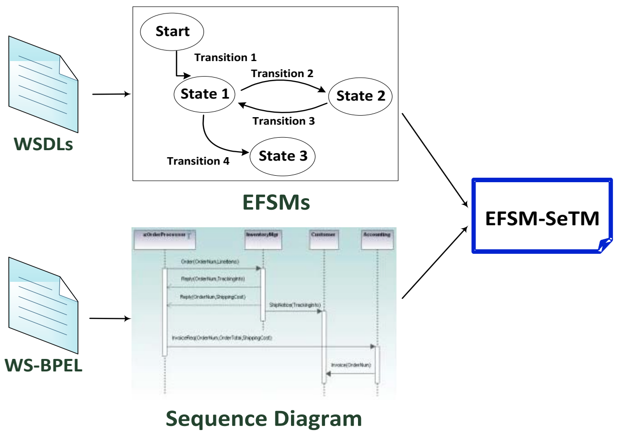

Figure 1. EFSM-SeTM Generation Model

The rest of this paper is organized as follows: Section 2 describes the proposed approach of EFSM-SeTM generation and the Section 3 will introduce the test paths generation and the test path coverage criteria. Section 4 shows the algorithm of the test paths generation. Section 5 concludes the contributions of this paper.

II. The Proposed Approach

A. *Framework of Test Model and Test Paths Generation*

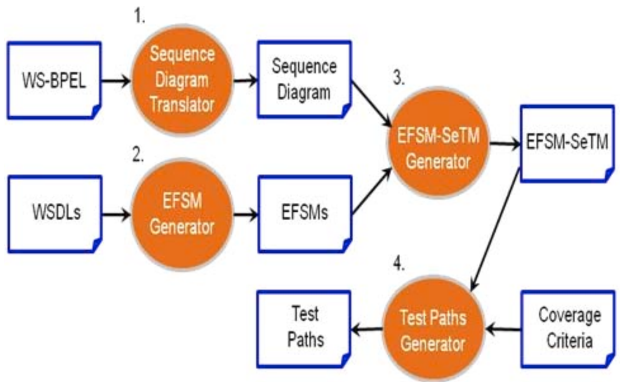

Figure 2. Framework of Test Model and Test Paths Generation

In this section, we will introduce a graph-based test model to enhance testing of interactions among the web services. This idea stems from Shaukat Ali's et al. [8], ChangSup Keum's et al. [6] and M.Emilia Cambronero's et al. [9] approaches. The interactions and the dynamic behaviors of the services should be examined for all possible states of the services involved. This is particular importance to web services composition testing. The general processes to create the test model and the test paths are described as shown in Figure 2. The first step, WS-BPEL file is translating into UML sequence diagram. The second step, we derive the EFSM models from each WSDL file of the web services. After EFSMs and sequence diagram generated, we combine EFSM models and sequence diagram to create the EFSM-SeTM as the third step. Based on the EFSM-SeTM and selected coverage criterion, the test paths generator is able to create the test paths. The proposed test model and test paths generation consist of the following steps:

1. Sequence Diagram Translator: The translator is responsible for deriving the sequence diagram from WS-BPEL. Two techniques can be used for translation processes, XMI (XML Metadata Interchange) and XSLT (Extensible Stylesheet Language Transformations). The XMI is a standard as an interchange format for UML models (such as sequence diagram). The XSLT is a language for transforming XML file into another XML file. We can use XSLT to translate WS-BPEL into XMI as the first step. The second step, sequence diagram can be retrieved from XMI file which is translated from WS-BPLE file.
2. EFSM Generator: The EFSM generator is to create EFSM models from WSDL files. From ChangSup Keum's [6], they present a technique to form the WSDL specification into the EFSM model. By adding the information of preconditions and operations to EFSM transactions, EFSM is well suited for describing the web services behavior. The detail of the EFSM generation will be described in section II-D.
3. EFSM-SeTM Generator: The EFSM-SeTM generator is to create EFSM-SeTM from sequence diagram and EFSM models. The EFSM-SeTM is the final test model, called Extended Finite State Machine-Sequence Test Model, which adds the information of services internal state into sequence diagram. The details of the EFSM-SeTM generation will be described in section II-E.
4. Test Paths Generator: From EFSM-SeTM, we define several kinds of coverage criteria based on all possible scenarios that cover each state transition of the web services. After we defined coverage criteria, the test paths can be derived from the EFSM-SeTM according to selected coverage criterion. The detail of the test paths generation will be introduced in section III.

B. *Defining EFSM-SeTM*

The EFSM-SeTM is utilized to generate test specifications for web services composition testing. In this section, we describe how to construct EFSM-SeTM from a given UML sequence diagram and EFSM models. In order to provide more precise rules or the well-formedness test model, each structural component of the EFSM-SeTM is specified as below:

The Vertex in EFSM-SeTM - There are two kinds of vertices in EFSM-SeTM; stateful vertex and stateless vertex. The stateful vertex corresponds to a stateful web service participating in the sequence diagram. The state invariants of stateful web service are specified and given a meaning to each state during the EFSM design phase. A stateful web service can receive a message in one or many states and show behavior for the same message in different states. A stateless vertex corresponds to a stateless web service and only represents a single vertex in the EFSM-SeTM graph.

The Edges in EFSM-SeTM - In the test model, there are two kinds of edges; message edge and transition edge. The message edge represents an operation call between services and each message edge may have a condition or iteration. The transition edge represents a state transition of web service and it connects to two different states of the same web service. The state transfers from a start state to an end state when it receives an operation call. Each operation call may trigger a state transition to occur. The transitions of the EFSM exhibit distinct states for the same operation. Therefore, one message edge in EFSM-SeTM may correspond to multiple transition edges which are representing a conditional transition. Each of these transitions is generally controlled by mutually exclusive conditions. The internal content of a vertex represents the web service name and state of the instance. The attributes of the message edge include sequence number, operation, receiver and sender. The attribute of the transition edge include operation, accepting state, and sending state.

C. Sequence Diagram Translator

For sequence diagram translator, we translate the WS-BPEL file into a sequence diagram. In Cambronero et al. [9], they developed a U4WC tool with XMI and XSLT to translate sequence diagram into WS-BPEL. The XMI [10] is an OMG (Object Management Group) standard for exchanging metadata information and the most common use of XMI is as an interchange format for UML models. The XSLT [11] is developed by the W3C to transform XML file into another XML file and hence it is able to transform XMI into WS-BPEL.

In other words, the WS-BPEL can be easy translated into XMI file by using XSLT as well. The XMI is the most common interchange format for UML models. From XMI, we can retrieve the UML sequence diagram. In this section we show two standards for translation of the WS-BPEL XML format into UML sequence diagram. This translation has been developed with XSLT (XML Stylesheets Language for Transformation) which is a language for transforming XML into another XML documents. In the market we can find some tools that allow us to model the UML diagrams and to obtain the corresponding XMI (XML Metadata Interchange) document, as IBM Rational Rose tool or Oracle JDeveloper. The WS-BPEL document generated by one of these tools can be easily translated into the XMI documents by using XSLT. From these processes, we can retrieve the UML sequence diagram from WS-BPEL document, by means of XSLT, following the translation rules explained in this section.

D. EFSM Generator

ChangSup Keum et al. [6] describe the way to create the EFSM model from WSDL specification. We use the same steps as Keum's approach to construct EFSM. Detailed information can be referenced in [6]. The EFSM model is an extended FSM model by adding variables, statements and conditions into FSM. It starts from an initial state; one state moves to another state according to the interactions of operations. The EFSM can be represented as a 6-tuple < S, s_0, I, O, T, V >; where "S" is the finite set of states, "s_0" is the initial state, "I" is the finite set of inputs, "O" is the finite set of outputs, "T" is the finite set of transitions, and the "V" is the finite set of variables. Each element t ∈ T can be represented as a 5-tuple < start_state, end_state, input, predicate, compute_block >, where "start_state" is the start state of t, "end_state" is end state of t; "input" is either an input or empty of t, "predicate" is a predicate expression in terms of variables in "V", the parameters of the input interaction and some constants, and "op" is a computation block consisting of assignment and output statements.

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. Here, the Ticket web service is used as an example. The following four steps show the construction of EFSM:

Step 1: To analyze the WSDL specification and the web service specification in informal language and fill the WSDL analysis template. For example, in Table 1, the WSDL analysis template filled out for Ticket web service. From WSDL description, the Ticket web service provides three operations (register, buyticket, and cancel). The "register" operation expects "Name" and "Email" ("Name" as customer name and "Email" as customer email) as two parameters and returns an "Identifier" as account number (AC_Id). The "buyticket" operation expects "Id" and "Qty." ("Id" is the account number and "Qty." is the quantity of buying ticket) and returns a result such as "success" or "fail". The "cancel" operation expects "Id" and returns a result such as "cancelled" or "fail".

Operation	Pre-condition	Post-condition
Name: register **Parameter**: Name, Email **Return**: Identifer	-	AC_Id > 0 AC_Name = Name
Name: buyticket **Parameter**: Id, Qty. **Return**: result	AC_Id = Id A_ticket ≧ Qty.	AC_Id > 0 A_ticket' = A_ticket – Qty.
Name: cancel **Parameter**: Id **Return**: result	AC_Id = Id	AC_Id > 0 A_ticket' = A_ticket + Qty.

Table 1. WSDL analysis template for Ticket web service

Step 2: To classify variables in the pre-condition and post-condition into control variables and data variables. Then the states of the EFSM under construction were made by the combination of different values of the control variables. In Table 2, it presents the classification of variables for Ticket web service and there are two control variables ("AC_Id" and "A_ticket"). A_ticket stands for available ticket.

Operation	Pre-condition		Post-condition	
	Control variable	Data variable	Control variable	Data variable
Name: register **Parameter**: Name, Email **Return**: identifer	-	Name Email	AC_Id	Name Email
Name: buyticket **Parameter**: Id, Qty. **Return**: result	AC_Id A_ticket	Id Qty.	AC_Id A_ticket	-
Name: cancel **Parameter**: Id **Return**: result	AC_Id	Id	AC_Id A_ticket	-

Table 2. Classification of variables for Ticket web service

In Figure 3, it shows an initial version of EFSM for the Ticket web service. The states are constructed by combining possible value range of control variables. The variable AC_Id and A_ticket have two possible values: range 0 and greater than 0. When the variable AC_Id is initialized by "register" operation, the AC_Id has a value greater than 0. Otherwise, the AC_Id is not initialized and it has a value 0. After initialization, the value of A_ticket is either 0 or greater than 0 according to the operation "buyticket" and "cancel". 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.

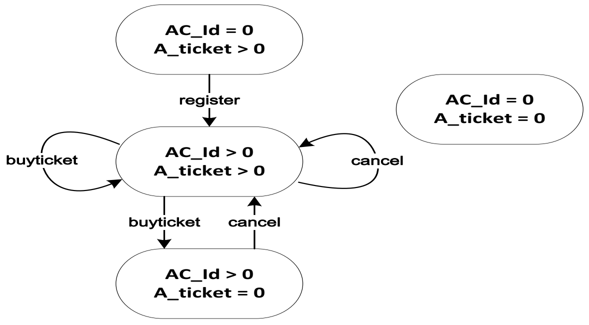

Figure 3. EFSM construction with control variables

Step 3: Since first the number of states would be huge and second there may have a possibility that unreachable states may exist, the redundant states will be removed in the initial version of EFSM model. In Figure 3, the state with AC_Id = 0 and A_ticket = 0 is an unreachable state and the unreachable states should be deleted. Some state could be merged into one state according to tester's judgment. In Figure 4, the enhanced EFSM obtained by removing the unreachable state. For readability and understandability, we assign a meaningful name to each state.

Step 4: Using the operation information in the WSDL to make a concrete transition in EFSM. An operation has input and output message. The pre-condition is transformed into guard condition and the post-condition is transformed into actions in the transition. In Figure 5, it shows the final EFSM derived from the WSDL specification for the Ticket web service.

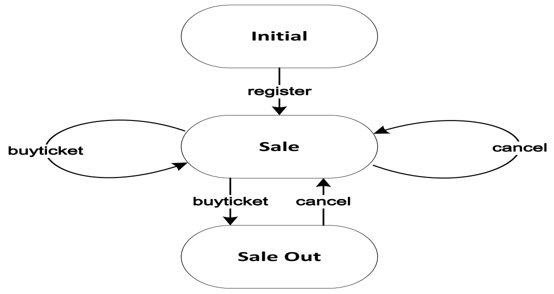

Figure 4. Enhanced EFSM with state reduction

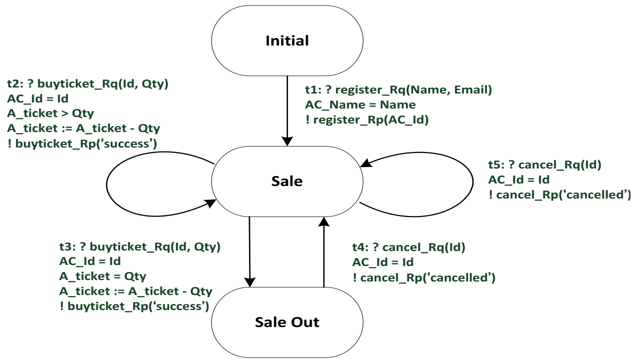

Figure 5. Final EFSM for Ticket web service

E. EFSM-SeTM Generator

The EFSM-SeTM generator is to create EFSM-SeTM from sequence diagram and EFSM models. Shaukat Ali et al. [8] proposed a technique to create a test model from state-chart and collaboration diagrams for Object-Oriented software integration testing. We apply the same steps as Ali's approach to construct EFSM-SeTM from the EFSM and sequence diagram. Detailed information can be referenced in [8]. Here, the BookShop composite web service is used as an example. The BookShop composite web service is composed of five web services which are the Bookstore service, Inventory service, Payment service, Delivery service, and Message service. The customer uses the BookShop composite service to buy a book which he likes. The sequence diagram (shown in Figure 6) shows the procedures of buying a book, as well as structural and behavioral aspects of the BookShop service. To construct the EFSM-SeTM model of the BookShop service, we start from the sequence diagram. The first operation (search) allows customer to search the book which he is interested in. After customer found a book, the second operation (checkStock) checks the inventory whether the book is in stock or not. If the book is in stock, the customer can make a decision to buy the book as the third operation (buy) and go to make a payment for the book as the fourth operation (makePayment). After customer make a payment, the fifth operation (ship) will ship the book from Inventory and the book is ready to deliver. The sixth operation (deliver) is to deliver the book and the seventh operation (confirmMessage) is to send a confirmation message to customer.

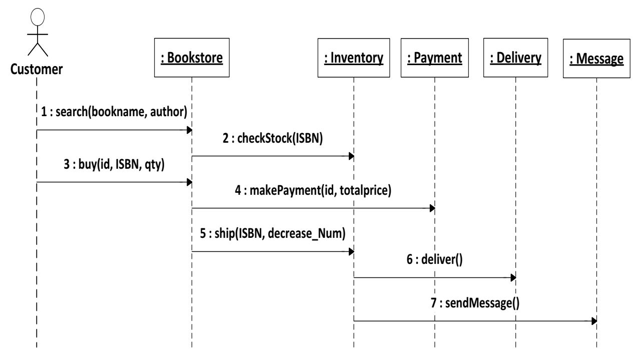

Figure 6. Sequence Diagram for Bookstore composite web service

The EFSM of Bookstore, Inventory and Payment web services are shown in Figure 7, 8 and 9 respectively. We know that the Bookstore, Inventory and Payment are stateful web services and each stateful service corresponds to an object in sequence diagram. The EFSM of Bookstore service defines two states: the "Sale" state for four operations (openAcct, buy, refund and search) and the "sale out" state for three operations (order, search and buy). The Inventory service defines two states: the "In Stock" state for four operations (newGoods, increaseStock, checkStock and shipment) and the "Out of Stock" state for two operations (shipment and checkStock). The Payment service defines two states as well: the "Not Pay" state for two operations (confirmOrder and cancelOrder) and the "Paid" state for makePayment operation.

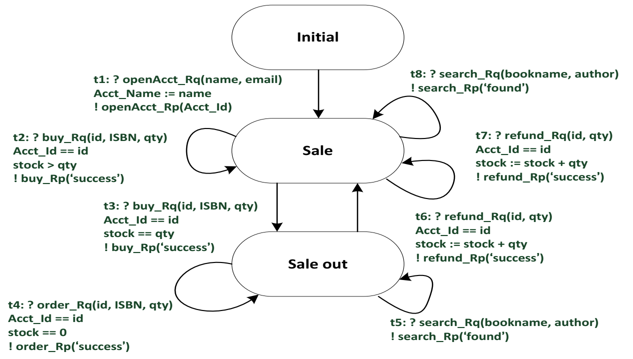

Figure 7. EFSM for Bookstore web service

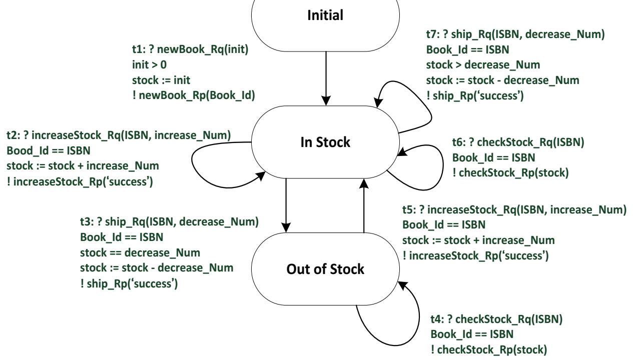
Initial
t1: ? newBook_Rq(init)
init > 0
stock := init
! newBook_Rp(Book_Id)
t7: ? ship_Rq(ISBN, decrease_Num)
Book_Id == ISBN
stock > decrease_Num
stock := stock - decrease_Num
! ship_Rp('success')
t2: ? increaseStock_Rq(ISBN, increase_Num)
Bood_Id == ISBN
stock := stock + increase_Num
! increaseStock_Rp('success')
In Stock
t6: ? checkStock_Rq(ISBN)
Book_Id == ISBN
! checkStock_Rp(stock)
t3: ? ship_Rq(ISBN, decrease_Num)
Book_Id == ISBN
stock == decrease_Num
stock := stock - decrease_Num
! ship_Rp('success')
t5: ? increaseStock_Rq(ISBN, increase_Num)
Book_Id == ISBN
stock := stock + increase_Num
! increaseStock_Rp('success')
Out of Stock
t4: ? checkStock_Rq(ISBN)
Book_Id == ISBN
! checkStock_Rp(stock)

Figure 8. EFSM for Inventory web service

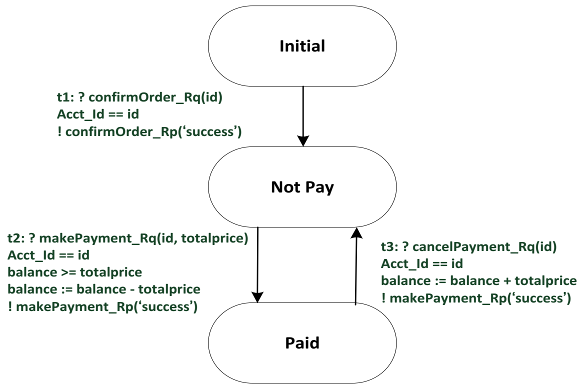
Initial
t1: ? confirmOrder_Rq(id)
Acct_Id == id
! confirmOrder_Rp('success')
Not Pay
t2: ? makePayment_Rq(id, totalprice)
Acct_Id == id
balance >= totalprice
balance := balance - totalprice
! makePayment_Rp('success')
t3: ? cancelPayment_Rq(id)
Acct_Id == id
balance := balance + totalprice
! makePayment_Rp('success')
Paid

Figure 9. EFSM for Payment web service

There are multiple vertices in the EFSM-SeTM are created for the object in sequence diagram. Each object corresponds to a stateful web service or a stateless web service. The vertex of object of stateful web service represents various states and the vertex of object of stateless web service only has one state. Vertices act as placeholders for objects of web services and they have labels of the form X@S (for the stateful web service). The X represents the service name and S is the state identifier as represented in EFSM. For the stateless web service, the labels of vertices are the form X@X. To construct the EFSM-SeTM of the BookShop the process begins from the null vertex which is dummy vertex. Following the sequence diagram, the first message (search) corresponds to the "Bookstore" service which is a stateful service, and it has two instance labels as Bookstore@Sale and Bookstore@Sale out. Bookstore@Sale and Bookstore@Sale out are added after the null vertex. The second message (checkStock) corresponds to the "Inventory" service which is a stateful service; and Inventroy@In Stock and Inventory@Out of Stock are added after two instances of the "Bookstore" service. The third message (buy) also corresponds to the "Bookstore" service; Bookstore@Sale and Payment@paid are added after the null vertex. The fourth message (makePayment) corresponds to the "Payment" service which is a stateful service; and Payment@Not Pay and Payment@Paid are added after two instances of the "Bookstore" service. The fifth message (ship) corresponds to the "Inventory" service. The Payment@Not Pay and Payment@Paid are added after two instances of the "Bookstore" service. The sixth message (deliver) is a stateless service which corresponds to Delivery service; and the Delivery@Delivery is added after two instances of the "Inventory" service. The seventh message (sendMessage) is also a stateless service which corresponds to Message service; and the Message@Message is added after two instances of the "Inventory" service as well. The EFSM-SeTM of the Book Shop composite web service is generated from sequence diagram and EFSM and it is shown in Figure 10.

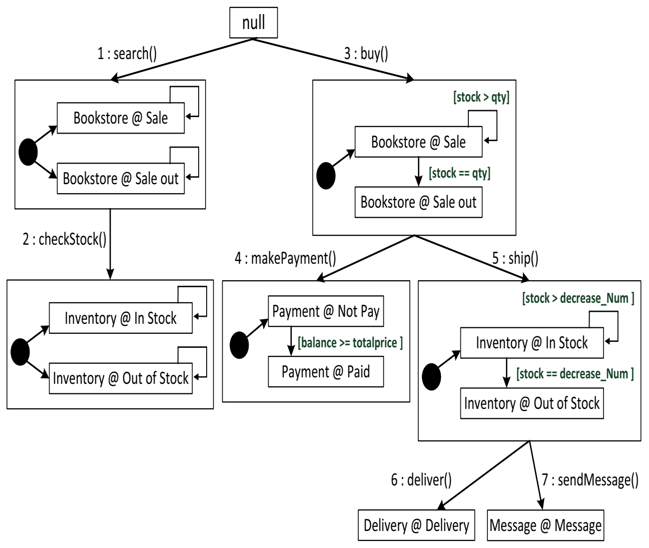

Figure 10. EFSM-SeTm for Book Shop Service

III. TEST PATH GENERATION

The test paths are derived by traveling the EFSM_SeTM. Each test path represents interactions among services in appropriate states. By traveling a whole EFSM_SeTM, all test paths will test all possible interactions with valid states in a particular sequence. The test path starts with the null vertex and builds by the order of the entire message sequence in the UML sequence diagram. The total number of test paths is decided by the state transition paths of stateful web services. Each message path may go through the stateful services and stateless services. Each state transition path of the stateful service is from a source state to a target state when it receives a particular message. The stateful service needs to travel each state transition path; otherwise the stateless service will be just a single vertex in EFSM-SeTM. However, they were not all feasible test paths when we generated the test paths. For example, in Figure 10, message 1 operates the Bookstore in the "Sale out" state. The test path should not include message 2 to operate the Inventory in the "In Stock" state, because it is obviously infeasible. Therefore, the infeasible path must be detected manually.

Figure 10 shows the sequence numbers and names of the messages along the edges, but doesn't include the internal structure and detailed information about the message. In test paths, messages are identified by their names, sequence numbers, sender service name and receiver service name. The general form of a message expression based on [5] is as below:

*SequenceNO : * [iteration] [Condition] message_name $ sender service_name @ receiver service_name @ state_identifier → [Guard] resultant_state*

The message expression for a non-modal class is as below:

*SequenceNo : * [iteration] [Condition] message_name $ sender service_name @ receiver service_name @receiver service_name*

The message expression describes a message call (message_name) on an object from a specified service (sender service_name) to the other specified service (receiver service_name) in a specified state (state_identifier). In the expression, the symbol ":" is used to separate the sequence number of message and the message expression. The "*" denotes repetition. The term before the "$" is the message name and after "$" shows the sender service name and the receiver service name, and they are separated by the first symbol "@" . The term after the second symbol "@", it indicates the resultant state of the object. The condition and iteration which show in message expression will skip for unconditional message.

A. *Definition of Coverage Criteria*

The number of test paths generated from EFSM_SeTM can provide an intuitive overhead of the testing level. However, in a large scale system, there are a large number of services that collaborate together and a large number of states included in such services. The number of test paths generated from such systems would grow exponentially. In other words, due to the cost, it is impractical, or even impossible, to test all the paths. Therefore, based on EFSM_SeTM, we define five coverage criteria that use minimum overhead to execute a reasonable level of testing.

- **All-Path coverage**

This criterion covered all paths in EFSM_SeTM and it is suitable for a completed testing in a small system. The total number of test paths in All-Path coverage can be calculated by taking a product of the numbers of state transition paths in each service. The state transition path is an internal transformation from the source service state to the target service state when receiving a particular message. Furthermore, these state transition paths are selected from EFSM based on a particular message in sequence diagram. However, based on some constraints with messages, some test paths are infeasible when traversing the EFSM_SeTM. Therefore, only the feasible test paths should be counted for the coverage criterion.

Service name	Message	Number of state	Number of transition
Bookstore	search()	2	2
Inventory	checkStock()	2	2
Bookstore	buy()	2	2
Payment	makePayment()	2	1
Inventory	ship()	2	2
Delivery	deliver()	1	1
Message	sendMessage()	1	1

Table 1. State Transitions of Book Shop Composite Web Service

Table 1 shows the number of states and transitions for each of the services. As mention above, the total number of test paths in All-Path coverage is the product of the numbers of state transition paths. Therefore, there are 2 * 2 * 2 * 1 * 2 * 1* 1 = 16 test paths of the particular case in table 1. Nevertheless, this criterion will result in a prohibitive number of test paths in modern systems. Some simple path coverage criteria which are subsets of All-Path coverage should be based on the testing requirement and testing budget.

- **All-State coverage**

This criterion subsumes All-Message coverage and ensures that every state of a service is executed at least once. The number of test paths in this criterion is determined by the maximum number of states to the all services. For example, in the Table 1 of EFSM_SeTM model, the maximum number of state to all of the services is two. Therefore, the total test paths are two. The All-State coverage criterion can be used to check if all services can function well in different states.

The 1st test path:
1: search$null@Bookstore@Sale → Sale
2: checkStock$Bookstore@Inventory@In Stock → In Stock
3: buy$null@Bookstore@Sale → [stock > qty] Sale
4: makePayment$Bookstore@Payment@Not Pay → [balance >= totalprice] Paid
5: ship$Bookstore@Inventory@In Stock → [stock > decrease_Num] In Stock
6: deliver$Inventory@Delivery@Delivery
7: sendMessage$Inventory@Message@Message

The 2nd test path:
1: search$null@Bookstore@Sale out → Sale out
2: checkStock$Bookstore@Inventory@Out of Stock → Out of Stock

- **All-Transition coverage**

This criterion subsumes All-State coverage criterion and guaranties that each state transition in a service is tested at least once. The maximum number of state transitions to all services will become the number of test paths in this criterion. For instance, in the Table 1 of EFSM_SeTM model, it shows that the maximum number of state transition to all of the services is two. Therefore, the total number of test paths is two. The All-Transition coverage criterion can reveal invalid transitions between services states.

The 1st test path:
1: search$null@Bookstore@Sale → Sale
2: checkStock$Bookstore@Inventory@In Stock → In Stock
3: buy$null@Bookstore@Sale → [stock > qty] Sale
4: makePayment$Bookstore@Payment@Not Pay → [balance >= totalprice] Paid
5: ship$Bookstore@Inventory@In Stock → [stock > decrease_Num] In Stock
6: deliver$Inventory@Delivery@Delivery
7: sendMessage$Inventory@Message@Message

The 2nd test path:
1: search$null@Bookstore@Sale → Sale
2: checkStock$Bookstore@Inventory@In Stock → In Stock
3: buy$null@Bookstore@Sale → [stock == qty] Sale Out
4: makePayment$Bookstore@Payment@Not Pay →[balance >= totalprice] Paid
5: ship$Bookstore@Inventory@In Stock → [stock == decrease_Num] Out of Stock
6: deliver$Inventory@Delivery@Delivery
7: sendMessage$Inventory@Message@Message

- **All-Message coverage**

The All-Message coverage criterion is the minimal coverage criterion when the number of test paths in this criterion is always one. This criterion guaranties that all the sequence of messages from the initial vertex to the end vertex of EFSM_SeTM in the sequence diagram are followed once. The function of this coverage criterion is to prove the correctness of the interactions between services regardless of the object state transitions. The test path can be selected randomly from All-Path coverage.

- **Predicate coverage**

Predicate expressions appear in the EFSMs as path conditions and guards. As such, the EFSM-SeTM contains these predicates as well. The Predicate coverage criterion guaranties the state transitions which have a predicate in a service is tested at least once. The maximum number of predicate to all services will become the number of test paths in this criterion. For instance, in the EFSM_SeTM model, it shows that the maximum number of predicate to all of the services is two. Therefore, the total number of test paths is two. The Predicate coverage criterion can reveal invalid transitions depending on the predicate between services states.

The 1st test path:
1: search$null@Bookstore@Sale → Sale
2: checkStock$Bookstore@Inventory@In Stock → In Stock
3: buy$null@Bookstore@Sale → [stock > qty] Sale
4: makePayment$Bookstore@Payment@Not Pay → [balance >= totalprice] Paid
5: ship$Bookstore@Inventory@In Stock → [stock > decrease_Num] In Stock
6: deliver$Inventory@Delivery@Delivery
7: sendMessage$Inventory@Message@Message

The 2nd test path:
1: search$null@Bookstore@Sale → Sale
2: checkStock$Bookstore@Inventory@In Stock → In Stock
3: buy$null@Bookstore@Sale → [stock == qty] Sale Out
4: makePayment$Bookstore@Payment@Not Pay → [balance >= totalprice] Paid
5: ship$Bookstore@Inventory@In Stock → [stock == decrease_Num] Out of Stock
6: deliver$Inventory@Delivery@Delivery
7: sendMessage$Inventory@Message@Message

IV. The Algorithm of Test Path Generation

In this section, we show an algorithm to describe how the test paths of All-Path coverage criterion are generated from EFSM-SeTM. The algorithms for the others coverage criteria will base on the algorithm of the All-Path coverage criterion. The algorithm (shown in Figure 11) takes EFSM-SeTM of the Book Shop composite web service as an input and the output is a set of test paths. The algorithm covered all test paths in EFSM-SeTM and it is able to complete test in composite web service. Lines at 13 to 34 execute every stateful service and stateless service in EFSM-SeTM.

```
Algorithm   AllPathCoverage(TM): TPSseq
Input       TM: EFSM-SeTM
Output      TPSseq: a sequence of test paths
Declare     TPSseq: a sequence of test paths
            MESseq: a sequence of message edges in TM
            medg: a message edge in TM
            tedg: a transition edge in TM
            tpm: test sub path corresponding to a message edge of type string
            tpt: test sub path corresponding to a transition edge of the current message edge of
                 type string
            TMEseq: a sequence of transition edges corresponding to a message Edge

1.  begin
2.    TPSseq ← { }
3.    MESseq ← TM.edge.message
4.    for all medg ∈ MESseq do
5.        tpm ← medg.sequenceNumber+" :" +medg.constraint+medg.associatedOperation+" $" +
                medg.senderService+" @" +medg.receiverService+" @"
6.        if | TPSseq | = 0
7.          TPSseq → insertAt (1, tpm)
8.        else
9.          for ( i = 1 to | TPSseq | ) do
10.           TPSseq.insertAt( i, TPSseq → at(i) → concat(tpm) )
11.         end for
12.       end if
13.       if medg.vertex.oclIsTypeOf(StatefulVertex)
14.         TMEseq ← medg.transition
15.         n ← | TPSseq |
16.         for ( j = 1 to | TMEseq | - 1 ) do
17.           for ( i = 1 to n ) do
18.             TPSseq → append( TPSseq → at(i) )
19.           end for
20.         end for
21.         n ← 0
22.         for ( i = 1 to | TMEseq | ) do
23.           for ( j = 1 to | TPSseq | / | TMEseq | ) do
24.             tedg ← TMEseq → at(i)
25.             tpt ← tedg.sourceState + " →" + tedg.guard + tedg.targetState
26.             TPSseq → insertAt ( (n+j), TPSseq → at(n+j) → concat(tpt) )
27.           end for
28.           n ← n+j
29.         end for
30.       else
31.         for ( i = 1 to | TPSseq | ) do
32.           TPSseq → insertAt ( i, TPSseq → at(i) → concat(medg.receiverClass) )
33.         end for
34.       end if
35.   end for
36.   return TPSseq
37. end
```

Figure 11. Algorithm for All-Path Coverage Criteria

V. CONCLUSION

In this paper, our approach focuses on creating a test model from the UML sequence diagram and EFSM. We derived the test paths from the test model for composite web service testing. To develop the test model, we have generated EFSM-SeTM from sequence diagram and EFSMs for composite web service testing. The EFSM-SeTM represents states of objects and interaction information in the test model. For verifying our approach, the Book Shop composite web service is illustrated as an example. We have also presented several coverage criteria for test path generation and the algorithm is to automatically generate the test paths. In the paper, we show how to develop EFSM-SeTM test model, how to define the coverage criteria and create the algorithm of the test path generation. There are several works in the future, including defining metrics for testing, test data generation, creating test cases from EFSM-SeTM, developing the system to do the testing, and using the industrial system for case studies to carefully analyze the cost-benefit of the proposed testing strategy.

REFERENCES

[1] F. Belli and M. Linschulte, "Event-driven modeling and testing of web services," in IEEE International Computer Software and Applications (COMPSAC), pp. 1168–1173, August 2008.

[2] Y. Zheng, J. Zhou, and P. Krause, "An automatic test case generation framework for web services," Journal of Software, vol. 2, no. 3, pp. 64–77, September 2007.

[3] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "Model-based generation of testbeds for web services," in 20th IFIP International Conference on Testing of Communicating Systems (TESTCOM), vol. 5047, pp. 266–282, June 2008.

[4] A.T. Endo, M. Linschulte, A. da Silva Simão, and S. do Rocio Senger de Souza, "Event- and Coverage-Based Testing of Web Services," in 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C), pp. 62-69, June 2010.

[5] A.T. Endo and A. Simao, "Model-Based Testing of Service-Oriented Applications via State Models," in 2011 IEEE International Conference on Services Computing (SCC) , pp. 432-439, July 2011.

[6] C. S. Keum, S. Kang, I. Y. Ko, J. Baik and Y. I. Choi, "Generating Test Cases for Web Services Using Extended Finite State Machine," TestCom 2006, pp. 103-117, May 2006.

[7] A. S. Kalaji, R. M. Hierons and S. Swift, "An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models," Information and Software Technology, Vol. 53, Issue 12, pp. 1297-1318, December 2011.

[8] S. Ali, L. C. Briand, M. J. Rehman, H. Asghar, M. Z. Z. Iqbal and A. Nadeem, "A state-based approach to integration testing based on UML models," Information and Software Technology, Vol. 49, Issues 11–12, pp. 1087-1106, November 2007.

[9] M.E. Cambronero, G. Diaz, J.J. Pardo and V. Valero, "Using UML Diagrams to Model Real-Time Web Services," ICIW '07. Second International Conference on Internet and Web Applications and Services, pp. 24-29, May 2007

[10] XML Metadata Interchange (XMI) Version 2.4.1. OMG, 2011. http://www.omg.org/spec/XMI/2.4.1/

[11] Michael Kay. XSL Transformations (XSLT) Version 2.0. W3C, 2007. http://www.w3.org/TR/xslt20/.

[12] C. S. Wu, W. C. Chang, S. Kim, and C. H. Huang, "Generating State-Based Polymorphism Interaction Graph from UML Diagrams for Object-Oriented Testing," In Proceedings of the International MultiConference of Engineers and Computer Scientists, pp. 726-731, March 2011.

[13] C. S. Wu, and I. Khoury, "Tree-based Search Algorithm for Web Service Composition in SaaS," The International Conference on Information Technology: New Generation (ITNG), pp.132-138, April 2012

[14] C. S. Wu and Y. T. Lee, "Automatic SaaS Test Cases Generation based on SOA in the Cloud Service," IEEE Int'l Conf. on Cloud Computing Technology and Science (CloudCom), pp. 349-354, December 2012.