New approach for EFSM-based Passive Testing of Web Services

Abdelghani Benharref, Rachida Dssouli, Mohamed Adel Serhani, Abdeslam En-Nouaary, Roch Glitho

Concordia University
1455 de Maisonneuve West Bd, Montreal, Quebec
H3G 1M8, Canada
{abdel,m_serhan,ennouaar}@ece.concordia.ca,
{dssouli,glitho}@ciise.concordia.ca,

Abstract. Fault management, including fault detection and location, is an important task in management of Web Services. Fault detection can be performed through testing, which can be active or passive. Based on passive observation of interactions between a Web Service and its client, a passive tester tries to detect possible misbehaviors in requests and/or responses. Passive observation is performed in two steps: passive homing and fault detection. In FSM-based observers, the homing consists of state recognition. However, it consists of state recognition and variables initialization in EFSM-based observers. In this paper, we present a novel approach to speed up homing of EFSM-based observers designed for observation of Web Services. Our approach is based on combining observed events and backward walks in the EFSM model to recognize states and appropriately initialize variables. We present different algorithms and illustrate the procedure through an example where faults would not be detected unless backward walks are considered.

Keywords: EFSM-based passive testing, Web Services testing.

1 Introduction

Web services, a rapidly emerging technology, offer a set of mechanisms for program-to-program interactions over the Internet [1]. Managing Web Services is critical because they are being actually used in a wide range of applications. Fault management including fault detection is an important issue in this management.

Active testing and passive testing have been used for fault detection. An active tester applies test cases to the Web Service Under Test (WSUT) and checks its responses. In passive testing, messages received (requests) and sent (responses) by the Web Service Under Observation (WSUO) are observed, and the correct functioning is checked against the WSUT's model. The observation is done by entities known as observers.

Passive testing can complement active testing because it helps detecting faults that have not been detected before deployment. Furthermore, in many cases, it is a better alternative to active testing when the system is already deployed in its final operating

environment. It enables fault detection without subjecting the system to test cases. Test cases consume resources and may even imply taking the WSUT off-line.

Passive testing is conducted in two steps: passive homing (or state recognition) and fault detection. During the first phase, the observer tries to figure out the state where the WSUO is moving actually. This phase is necessary when the observation starts a while after the interaction has started and previous traces are not available.

Few models have been used for model-based observers but most of the published work on passive testing are on control part of systems and are based on Finite State Machine (FSM) model ([2], [3], [4]). Although this model is appropriate for control parts of WSUO, it does not support data flow. Extended FSM (EFSM) is more appropriate for the handling of variables.

The homing procedure in an EFSM-based observer consists of recognizing the actual state of the WSUO in addition to assigning appropriate values to different variables. In the few published papers on EFSM-based passive testing, the homing procedure is either ignored or it depends on the upcoming observed request/responses. In the first case ([5], [6], [7]), the observer must get all the traces to be able to initiate the fault detection process. In the second case ([8], [9]), the observer waits for exchanged messages before moving forward in the homing procedure. Ignoring the homing phase is a very restrictive assumption. Waiting for exchanged messages to continue on the homing procedure may delay the fault detection. Moreover, if there is a significant time gap between requests and responses, the observer spends most of its time waiting.

In this paper, we present a novel approach for homing online EFSM-based observers. Unlike offline observers, online observers analyze the observed traces in real time and report faults as soon as they appear. The observer performs forward walks whenever a new event (request or response) is observed. It performs backward walks in the EFSM model of the WSUO in absence of observed events. The information gathered from the backward and forward walks help speeding up the homing procedure.

The remaining sections of this paper are organized as follows: in the next section, we present related work for passive observation based on FSM and EFSM models. Section 3 presents our new approach using backward and forward walks to speed up the homing procedure and discusses different algorithms illustrated through an example. Section 4 concludes the paper and gives an insight for future works.

2 Related work

Active testing refers to the process of applying a set of requests to the WSUT and verifying its reactions. In this configuration [10], the tester has complete control over the requests and uses selected test sequences to reveal possible faults in the WSUT. Even though it is performed before deployment, active testing is not practical for management once a Web Service is operating in its final environment. Under normal conditions, the tester has no control over requests and/or responses. Passive observation is a potential alternative in this case.

Fig. 1 shows an observer monitoring interactions between the WSUO and its client during normal operations without disturbing it. Disturbing in this case means no injection of requests messages for testing purposes. If the observed responses are different from what's expected, the WSUO is then declared faulty.

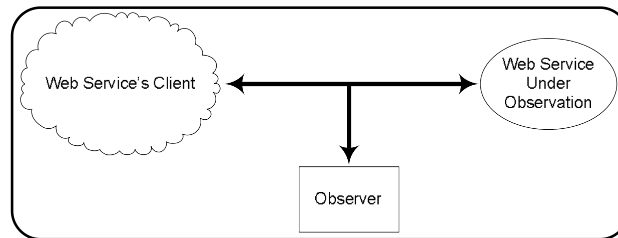

Fig. 1. Passive Testing Architecture

Formal methods, especially Finite State Machine (FSM) have been used in passive observation. Lee et al. in [2] propose algorithms for passive homing and fault detection for FSM-based observers for network management. These algorithms are extended in ([11]; [12]) to deal with fault location in networks and in [13] for avionics telecommunication. They have also been applied to GSM-MAP protocol in [4] and to TCP in [14]. Miller et al. extended the algorithms using Communicating FSM (CFSM) in [3]. While these algorithms work fine, they provide no support for dataflow which requires consideration of EFSM models.

EFSM is an extension of FSM by the following:

§ Interactions have certain parameters, which are typed.

§ The machine has a certain number of local variables, which are typed.

§ Each transition is associated with an enabling predicate. The predicate can be any expression that evaluates to a Boolean (TRUE or FALSE). It depends on parameters of the received input and/or current values of local variables.

§ Whenever a transition is fired, local variables can be updated accordingly and parameters of the output are computed.

Formally, an EFSM is described by a tuple $M = (S, S_0, I, O, T, V, \delta)$ ([10]) where:

§ S is a set of states,

§ $S_0 \in S$ is the initial state,

§ I is a finite set of inputs,

§ O is a finite set of outputs,

§ T is a finite set of transitions

§ V is a finite set of variables

§ δ: S x (I U O) à S is a transition relation

In an EFSM, each transition of T is represented as t: I|P|A|O where:

§ t: label/ID of the transition,

§ S_s: starting state of the transition,

§ I: the input that triggers the transition,

§ P: the enabling predicate (data conditions),

§ A: variables assignments

§ O: the output produced by the transition

§ S_e: ending state of the transition

Using EFSM models allows the detection of input/output faults in addition to faults related to data flow. The latter faults include calling of wrong function, wrong specification of data type, wrong initial values of variables, and referencing undefined or wrong variables.

In the literature on EFSM-based observers, the homing procedure is either ignored or depends fully on observed messages. In the first case, the authors in ([5]; [15]; [7]; [16]) suppose that the observation will start sharply with the interaction between the WSUO and its client. A passive observer based on this assumption will not be able to detect faults if it does not get whole traces. In the second case ([8]; [9]), the observer must wait for exchange of messages before moving forward in the homing procedure.

Since we are interested in online observation of Web Services, ignoring the homing procedure is not an option. We suppose that an EFSM-based online observer can initiate its observation at any time without having access to previously exchanged requests/responses. In the work presented in ([8]; [9]), the observer uses the exchanged messages for state and variables homing. This approach is efficient when the time gap between requests and responses is too short so the observer will be processing traces most of its time. If this time gap is relatively high, the observer spends a significant amount of time waiting for events while valuable information can be gathered by analyzing the EFSM model of the WSUO. The example presented in section 3.6 shows an example where the approaches presented in ([8]; [9]) fail to detect a fault that would have been detected if the observer was performing appropriate analysis of the EFSM machine of the WSUO.

3 EFSM-based observation: forward and backward walks

In client-server communication as in Web Services, it is reasonable to assume that there will be a delay between requests and responses. In fact, the client takes time to formulate and send its request. Once a response is received, the client takes again some time to process the response and decide what to do with it. Moreover, the Web Service requires time to process a request, generate, and send its response.

To speed up the homing procedure, the observer should make a concise use of the information contained within the EFSM model in addition to the information carried by observed events. The homing algorithm can perform backward walks in the EFSM model to guess what transitions the WSUO fired before getting into its actual state. By analyzing the set of predicates and variable definitions on these transitions, the observer can reduce the set of possible states and/or the set of possible values of variables. Performing both backward and forward walks provides a set of possible execution trees: the forward process adds execution sequences to the root of trees, and the backward process adds execution sequences to the leaf states of trees.

During the homing procedure, the observer manipulates the following entities:

§ Set of Possible States (**SPS**): this is the set of possible states with regards to what has been observed and processed up to now. At the beginning, all states are possible.

§ Tree of Possible Previous States for state s (**TPPS(s)**): this tree contains the possible paths that could lead to each state s in the SPS. During the homing procedure, there is a TPPS for each state in the SPS.
§ Set of Possible Variable Values for variable v (**SPVV(v)**): this is the set of all possible values that variable v can have with regards to what has been received and processed before. It consists of a list of specific values or ranges. At the beginning, all values in the definition's domain of variable v are possible.
§ Set of Known Variables (**SKV**): the set of known variables. A variable is said to be known if it is assigned a specific value. In this case, SPVV(v) contains one element, i.e. |SPVV(v)| = 1.
§ Set of Unknown Variables (**SUV**): the set of variables not yet known.

The next three sub-sections present in detail the processes of analyzing observed requests and responses and performing backward walks within an EFSM-based observer using both backward and forward walks.

3.1 The homing controller algorithm

While the observer is going through the homing procedure (Algorithm 1), it has 3 possible options:

1. process a request that has been received by the WSUO (line 11),
2. process a response that has been sent by the WSUO (line 17), or
3. perform a one-step backward walk (line 20). In this case, the algorithm considers the event e that triggers the loop as empty.

Processing observed events has priority and the backward walk is performed if and only if there are no observed events waiting for processing. This procedure is repeated until:

§ a fault is detected (unexpected input/output which results in an empty set of possible states and/or contradictory values of variables), or
§ the set of possible states has one item (|SPS| = 1) **and** the set of unknown variables is empty (SUV = Ø).

The complexity of Algorithm 1 depends on the number of events required to successfully achieve the homing procedure (line 4) and the complexity of processInput (line 11), processOutput (line 17), and performBackWalk (line 20). Lets denote the number of cycles to achieve the homing by n, and the complexities of processInput, processOutput and performBackWalk by O(BF_PI), O(BF_PO), O(BF_BW) respectively. The complexity O(H) of the homing algorithm is given in Equation 1 and will be developed through the following sections when individual complexities will be computed.

$$O(H) = n.O(BF_PI) + n.O(BF_PO) + n.O(BF_BW) \quad \text{Equation } \mathbf{1}$$

```
   SPS := S // At startup, all states are possible
   SUV := V // At startup, all variables are unknown
   Expected_Event ← "Any"
   Data: event e
4  repeat
      e ← observed event
      switch (e) do
         case (e is an input)
            if (Expected_Event == "Output") then
               return "Fault: Output expected not Input"
            else
11             processInput(e);          // Complexity:  O(BF_PI)
               Expected_Event ← "Output";
         case (e is an output)
            if (Expected_Event == "Input") then
               return "Fault: Input expected not Output"
            else
17             processOutput(e) ;        // Complexity:  O(BF_PO)
               Expected_Event ← "Input";
         otherwise
20          performBackWalk ;            // Complexity:  O(BF_BW)
   until (|SPS| == 1)AND(|SUV| == 0)
```

Algorithm 1. Homing controller

3.2 Processing observed requests

When the observer witnesses an input, if the observer was expecting an output, a fault ("Output expected rather than Input") is generated. Otherwise, it removes all the states in the set of possible states that don't accept the input, and the states that accept the input but the predicate of the corresponding transition is evaluated to FALSE. For each of the remaining possible transitions, the input parameters are assigned (if applicable) to appropriate state variables. Then, the predicate condition is decomposed into elementary expressions (operands of AND/OR/XOR combinations). For each state variable, the set of possible values/ranges is updated using the elementary conditions. If this set contains a unique value, this latter is assigned to the corresponding variable; this variable is then removed from the set of unknown

variables and added to the set of known variables. The transition's assignments part is processed, then updating the sets of known/unknown variables accordingly (Algorithm 2). The observer expects now the next observable event to be an output.

```
   Input: event e
   Data: boolean possibleState
1  foreach (S ∈ SPS) do
       possibleState = false
3      foreach Transition t so that ((t.S_s == S) AND (t.I == e) AND
       (t.P ≠ FALSE)) do
           possibleState = true
           assign appropriate variables the values of the parameters of e
6          update SPVV, SKV, and SUV
7          decompose the predicate into elementary conditions
8          foreach (elementary condition) do
9              update the SPVV, SKV, and SUV
               if (contradictory values/ranges) then
                   return "Contradictory values/ranges"
       if (possibleState == false) then
           remove S from SPS
           if (SPS == ∅) then
               return "Fault detected before homing is complete"
```

Algorithm 2. Processing observed requests

The complexity of Algorithm 2 is affected by the maximum number of states in the SPS (line 1), maximum number of transitions at each state in the SPS (line 3), and the complexity of updating the SPVV, SKV, and SUV. In fact, we can assume that a predicate will have very few elementary conditions, then decomposing the predicate (line 7) and using the elementary conditions to update the variables (line 8) does not affect the complexity of the whole algorithm. If the number of variables is V, the complexity of updating the SPVV, SKV, and SUV is in the order of O(V) since the procedure should go through all the variables. The complexity of Algorithm 2 is depicted in Equation 2 where S_{max} is the maximum number of states in the SPS, and T_{max} is the maximum number of transitions that a state in the SPS can have.

$$O(BF_PI)=O(S_{max}.T_{max}.V) \qquad \text{Equation 2}$$

3.3 Processing observed responses

In case the event is a response (output), if the observer was expecting an input, a fault ("Input expected rather than Output") is generated. Otherwise, the observer removes all the states in the set of possible states that don't have transitions that produce the output. If a state has two (or more) possible transitions, the TPPS is cloned as many as possible (number of possible transitions) so that each clone represents a possible transition. The assignment part of the transition is processed and variables are updated. The set of possible states holds the ending states of all the possible transitions. In the context of SOAP communication between a Web Service and its client, the response (message) holds basically one parameter. Whenever an output message is observed, a variable becomes known, or at least a new condition on variable values is augmented unless the message carries no parameter or the variable is already known. The observer expects now the next observable event to be an input.

Let's now determine the complexity of Algorithm 3. If we denote the maximum number of nodes (i.e states) in a TPPS tree by P_{max}, cloning a TPPS tree (line 5) is in the order of $O(P_{max})$. Moreover, the complexity of removing a TPPS tree (lines 14 and 19) is also in the order of $O(P_{max})$. Lines 8 and 9 do not affect the complexity since the number of assignments in a transition is somehow low compared, for instance, to P_{max}. The complexity of Algorithm 3 then can be written as:

$$O(BF_PO) = O(S_{max}.T_{max}.(P_{max} + V)) \qquad \text{Equation } \mathbf{3}$$

3.4 Performing backward walk

While the observer is waiting for a new event (either request or response), it can perform a 1-step backward walk in the EFSM model to guess the path that could bring the Web Service to its actual state. From each state in the set of possible states, the observer builds a tree of probable-previously visited states and fired transitions. Every time a transition could lead to the actual state or one of its possible previous states, the variables constraints in the enabling condition is added as a set of successive elementary conditions connected with logical operators OR and AND: constraints of two successive transitions are connected with AND, while constraints on two transitions ending at the same state are connected with OR.

```
   Data: event e
   Data: boolean possibleState
1  foreach (S ∈ SPS) do
   |  possibleState = false
3  |  foreach Transition t so that ((t.S_s == S) AND (t.I == e) AND
   |  (t.P ≠ FALSE) AND (t can produce e)) do
   |  |  possibleState = true
5  |  |  clone the corresponding TPPS
   |  |  t.S_e becomes the root of the cloned TPPS
   |  |  S becomes its child
8  |  |  process the transition's assignment part
9  |  |  assign appropriate variables values of the parameter (if any) of e
10 |  |  update the SPVV, SKV and SUV
   |  |  if (contradictory values/ranges) then
   |  |  |_ return "Contradictory values/ranges"
   |  |_
13 |  remove S from the SPS
14 |  remove the original TPPS ;   /* no longer useful, cloned (and
   |  updated) trees will be used */
   |_
   if (possibleState == false) then
16 |  remove S from SPS
   |  if (SPS == ∅) then
   |  |_ return "Fault detected before homing is complete"
19 |_ remove the corresponding TPPS
```

Algorithm 3. Processing observed responses

Algorithm 4 has three embedded loops. The first loop (line 1) is bounded by the number of TPPS trees; that is, the number of states in the SPS (S_{max}). The second loop (line 2) goes through all leaf states of a TPPS, which is at the worst case P_{max}. The third loop (line 3) explores all the states in the EFSM that can lead to a particular state in a TPPS. Lets denote the number of states in an EFSM by S_{EFSM}. Propagating a constraint through the root of a TPPS (line 4) is in the order of $O(P_{max}.V)$ since the procedure has to process all states and update the SPVV at each state. The complexity of Algorithm 4 can be written as:

$$P(BF_BW) = O(S_{max}.S_{EFSM}.P^2_{max}.V) \qquad \text{Equation } \mathbf{4}$$

```
Input: EFSM
1 foreach (TPPS) do
2     foreach (Leaf state S of TPPS) do
3         foreach (state S' in the EFSM that leads to S) do
4             Propagate the constraints of the corresponding transition
              toward the root of TPPS;
              /* Lets consider that propagation cannot go beyound
                 state Sp                                            */
              if (Sp is the root of TPPS) then
                  /* this path is possible→ consider it in the
                     TPPS                                            */
                  add S' as child of S;
7             update SPVV, SKV, and SUV;
              if (contradictory values/ranges) then
                  return "Contradictory values/ranges";
```

Algorithm 4. Performing Backward walk

From Equation **1**, Equation 2, Equation **3**, and Equation **4**, the overall complexity for homing an observer using Algorithm 1 can be developed as follows:

$$
\begin{aligned}
O(H) = O(&n.S_{max}.T_{max}.V + n.S_{max}.T_{max}.(P_{max} + V) + \\
&n.S_{max}.S_{EFSM}.P^2{}_{max}.V) \\
= \; &n.S_{max}.T_{max}.(P_{max} + V) + n.S_{max}.S_{EFSM}.P^2{}_{max}.V)
\end{aligned}
$$

3.5 Discussion

Although backward walks-based observers require a little bit more resources than an observer without backward walks, this overload is acceptable. First of all, backward walks are performed whenever there is no trace to analyze so the observer does not use additional processing time. It just uses the slots initially allocated to trace analysis. Second, limiting the backward to a unique step at a time reduces the duration of cycles of Algorithm 4 and does not delay processing of eventual available traces.

As for convergence of Algorithm 1, it is not possible to decide if the observer will converge or not. This is the case for both brands of observers: with backward and without backward. This limitation is out of the scope of the homing approach used but fully tied to the fact that the observer has no control on exchanged events. The Web Service and its client can continuously exchange messages that do not bring useful information to reduce the SPS and the SPVV.

However, the backward approach can be compared to the approach without backward, for the same WSUO and observed traces, as follows:

Property 1: if an observer without backward walks converges, an observer with backward walks converges too.

Property 2: if an observer without backward walks requires *n* cycles to converge, and an observer with backward walks requires *m* cycles to converge, then $m \leq n$.

The next sub-section presents a proof of property 2 which can be considered also as proof for property 1.

Proof

The homing algorithm converges when the SPS has one element and the SUV is empty. The SUV is empty when, for each variable v in V, SPVV(v) contains a unique element.

As discussed above, analysis of traces adds states as roots of TPPS and backward walks adds states as leaves of TPPS. Whenever a trace can generate two different execution paths, the corresponding TPPS is cloned. This will build TPPS trees where the root has a unique child. In such trees, all constraints propagation from backward walks will propagate using AND operator between the root and its child. This propagation tries to reduce the SPVV; in the worst case the SPVV is neither reduced nor extended.

In Fig. 2, at a cycle *i*, a TPPS has S_i as root, S_j is its child, and $SPVV_i(v)$ is the set of possible values of variable v at S_i as computed from a previously observed trace. Suppose that during cycle *i+1*, the backward walk adds two leaves to S_j: S_l and S_k. In Fig. 2, the labels on transitions represent the SPVV that result from the predicate of the transitions.

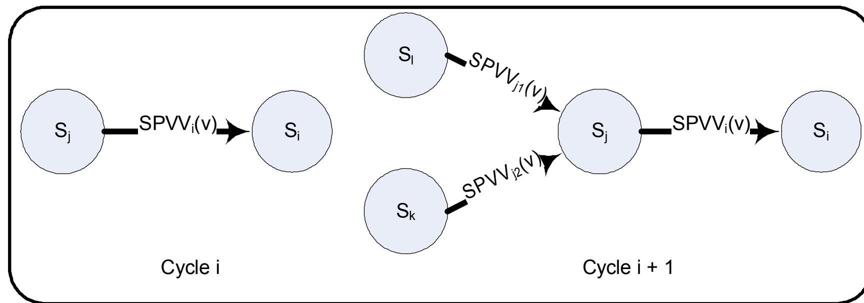

Fig. 2. SPVV and constraints propagation

Propagation of constraints from S_l and S_k to S_j and then to S_i modifies SPVV(v) as follows: $SPVV_{i+1}(v) = SPVV_i(v) \cap ((SPVV_l(v) \cup SPVV_k(v))$. There are three cases:

1. $SPVV_i(v) \subseteq (SPVV_l(v) \cup SPVV_k(v))$: in this case, the $SPVV_{i+1}(v)$ is equal to $SPVV_i(v)$. The backward walks do not bring useful information to reduce $SPVV_i(v)$. If subsequent backward walks do the same, the number of required cycles for homing remains unchanged: *m=n*.
2. $SPVV_{i+1}(v) = \varnothing$: this indicates that the variable, at S_i after cycle i+1 can not have any value from its definition domain. The observer detects a fault immediately without waiting for the next observed event which results in *m* strictly less than *n* ($m<n$).
3. $SPVV_{i+1}(v) \subset SPVV_i(v)$: in this case, the SPVV(v) is reduced. If following backward walks, associated to trace analysis, reduce further the SPVV(v), the homing with backward is likely to require less than *n* cycles ($m<n$) or at most *n* cycles ($m=n$).

The following example illustrates the first case where backward walks reduce the number of required cycles ($m<n$) and allows detection of faults that can not be detected without backward walks. The execution of the homing procedure is detailed hereafter in a step by step scenario.

3.6 Example

Let's consider the portion of an EFSM of a Web Service illustrated in Fig. 3 where variables u, x, y, and z are integers. Events I1(15), O(13), and I2(0) are observed respectively. Each transition is represented as t:I|P|A|O where t is the label of the transition, I its input, P its predicate, A is the set of assignments, and O is the output. A predicate of a transition is evaluated to TRUE/FALSE if its condition is true/false, otherwise it is said INCONCLUSIVE if the predicate can not be evaluated. The latter case occurs if some of the variables in the predicate are not yet known.

Observation without backward walks

After observing I1(15), transitions t1, t2, and t4 can be fired but not t3 or t5. However, since the input parameter is bigger than 0, the predicate of t4 is evaluated to FALSE. Only transitions t1 and t2 should be considered since the variables y and z are, up to now, unknown and the predicates are evaluated to INCONCLUSIVE. This reduces the set of possible states to S1 and S2. If t1 is executed then $x := 15$, $y > 15$, and $z := 15 - y$, if t2 is executed then $y := 15$, $z < 15$, $x := 15 - z$.

When O1(13) is observed, the value of the output parameter (13) indicates that transition t2 has been executed. Later on, when event I2(0) is observed, since the variable u is unknown, the predicate ($x > u$) is evaluated to INCONCLUSIVE, which enables the transition. So, the sequence I1(15), O1(13), I2(0) executes properly.

However, the sequence I1(15), O1(13), I2(0) is a faulty sequence and the fault would be detected if backward walks have been considered as discussed in the next section.

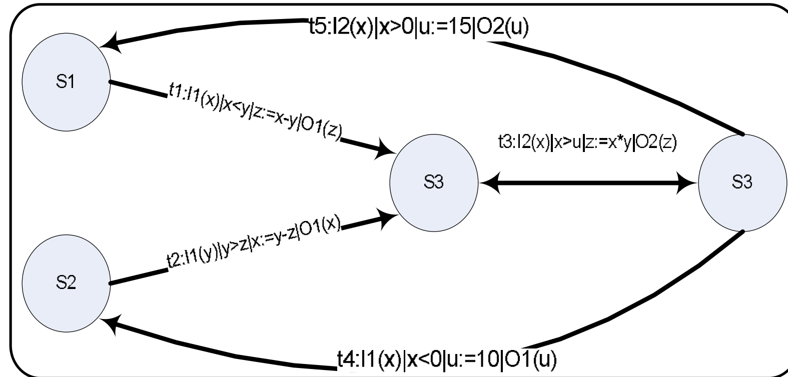

Fig. 3. EFSM Example

Observation with backward walks

The delay after each event (I1, O1, and I2) gives the observer opportunities to perform backward walks. The observer executes the following operations: processInput(I1(15)), performBackWalk, processOutput(O1(13)), performBackWalk, processInput(I2(0)).

As illustrated in Table 1, after executing the first three operations, SPS contains S3. In TPPS, S2 is the child of S3. To get to S2, the only previous transition is t4 which assigns 10 to variable u. From this point forward, the homing procedure is completed since SPS has one state and SUV is empty. Later on when receiving I2(0), transition t3 can not be fired since its predicate (x>u) is evaluated to FALSE. The observer notifies the WSO that a fault just occurred.

Table 1. Content of SPS, SPVV, SKV, SUV, TPPS

	I1(15)	Backward walk	O1(13)
SPS	S1, S2	S1, S2	S3
SPVV	t1 : x:=15, y> 15, z:= x-y or t2 : y:=15, z <15, x:=y-z	t4 : u:=10 or t5 : u:=15	x:=13, y:=15, z:=2, u:=10
SKV	t1 : x, or t2 : y	u, x or u, y	x, y, z, u

SUV	t1 : y, z, u or t2 : x, z, u	y, z or x, z	Ø
TPPS	Figure 4.a	Figure 4.b	

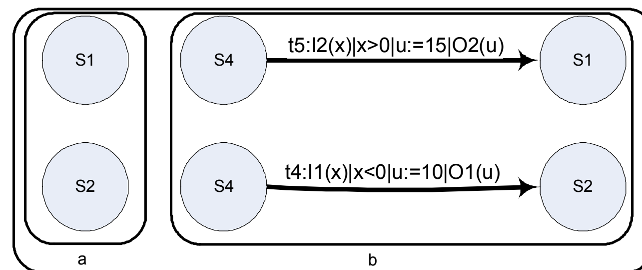

Fig. 4. TPPS

4 Conclusion

Fault detection is a basic operation in management of Web Services. It is conducted through testing which can be passive or active. An active tester applies selected test cases to the WSUT and checks the responses. Unlike active testers, a passive tester observes, passively, the interactions between the WSUO and its client. Based on this observation, correctness of requests and/or responses is verified.

FSM models have been used for passive testing for network management. However, it does not support data flows, an important aspect in Web Services XML-messaging. EFSM has the ability to specify both control and data flow parts of Web Services. When designing EFSM-based observers, the homing procedure has to assign appropriate values for different variables.

In this paper, we presented a novel approach for homing EFSM-based observers. This approach is based on observed events and on backward walks in the EFSM model of the WSUO. Whenever a trace is observed, it's immediately processed by the observer. Otherwise, the observer analyzes the possible paths that could bring the WSUO to its actual state. Analyzing the set of constraints on different paths could reduce the set of possible values variables can have at a specific state.

We are currently implementing observers based on the algorithms presented above and Web Services that will be used to evaluate the detection capabilities of such observers.

References

[1] W3C, "World Wide Consortium," at http://www.w3.org 2006.

[2] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John, "Passive testing and applications to network management," *International Conference on Network Protocols*. Atlanta, GA, USA: IEEE Computer Society, (1997), pp. 113-22.

[3] R. E. Miller, "Passive testing of networks using a CFSM specification," *International Performance, Computing and Communications Conference*. Tempe/Phoenix, AZ, USA: IEEE, (1998), pp. 111-16.

[4] M. Tabourier, A. Cavalli, and M. Ionescu, "A GSM-MAP protocol experiment using passive testing," *Formal Methods. World Congress on Formal Methods in the Development of Computing Systems*, vol. vol.1, *Lecture Notes in Computer Science (LNCS)*. Toulouse, France: Springer Verlag, (1999), pp. 915-34.

[5] A. Cavalli, C. Gervy, and S. Prokopenko, "New approaches for passive testing using an Extended Finite State Machine specification," *Information and Software Technology*, vol. 45, pp. 837-852, (2003).

[6] B. Alcalde, A. Cavalli, D. Chen, D. Khuu, and D. Lee, "Network protocol system passive testing for fault management: a backward checking approach," *Formal Techniques for Networked and Distributed Systems (FM)*, *Lecture Notes in Computer Science (LNCS)*. Madrid, Spain: Springer Verlag, (2004), pp. 150-66.

[7] B. T. Ladani, B. Alcalde, and A. Cavalli, "Passive testing - a constrained invariant checking approach," *17th International Conference on Testing of communicating systems (TestCom)*, *Lecture Notes in Computer Science (LNCS)*. Montreal, Que., Canada: Springer Verlag, (2005), pp. 9-22.

[8] D. Lee, C. Dongluo, H. Ruibing, R. E. Miller, W. Jianping, and Y. Xia, "A formal approach for passive testing of protocol data portions," *10th International Conference on Network Protocols*. Paris, France: IEEE Computer Society, (2002), pp. 122-31.

[9] D. Lee, C. Dongluo, H. Ruibing, R. E. Miller, W. Jianping, and Y. Xia, "Network protocol system monitoring-a formal approach with passive testing," *IEEE/ACM Transactions on Networking*, vol. 14, pp. 424-37, (2006).

[10] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test development for communication protocols: towards automation," *Computer Networks*, vol. 31, pp. 1835-72, (1999).

[11] R. E. Miller and K. A. Arisha, "On fault location in networks by passive testing," *International Performance, Computing, and Communications Conference*. Phoenix, AZ, USA: IEEE, (2000), pp. 281-7.

[12] R. E. Miller and K. A. Arisha, "Fault identification in networks by passive testing," *34th Annual Simulation Symposium*. Seattle, WA, USA: IEEE Computer Society, (2001), pp. 277-84.

[13] K. A. Arisha, "Fault management in avionics telecommunication using passive testing," *20th Digital Avionics Systems Conference (DASC)*, vol. 1. Daytona Beach, FL, USA: IEEE, (2001), pp. 1-7.

[14] C. Dongluo, W. Jianping, and C. HuiCheng, "Passive testing on TCP," *International Conference on Communication Technology (ICCT)*. Beijing, China: Beijing Univ. Posts & Telecommun, (2003), pp. 182-6.

[15] J. A. Arnedo, A. Cavalli, and M. Nunez, "Fast testing of critical properties through passive testing," *15th IFIP International Conference on Testing of Communicating Systems*, *Lecture Notes in Computer Science (LNCS)*. Sophia Antipolis, France: Springer Verlag, (2003), pp. 295-310.

[16] E. Bayse, A. Cavalli, M. Nunez, and F. Zaidi, "A passive testing approach based on invariants: application to the WAP," *Computer Networks*, vol. 48, pp. 247-66, (2005).