Test Sequence Generation for Distributed Software System

Shuai Wang
1. Department of Automation, Tsinghua University
2. Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China
wangshuai81@gmail.com

Yindong Ji
1. Department of Automation, Tsinghua University
2. Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China
jyd@tsinghua.edu.cn

Shiyuan Yang
Department of Automation, Tsinghua University
ysy-dau@tsinghua.edu.cn

***Abstract*— This paper considers the test case generation for distributed software (a test case contains one or more test sequences). Applying the single finite state machine (FSM) test approach to distributed software, we will suffer from some problems: 1) the state combinatorial explosion problem; 2) some unexecutable test cases may be generated; 3) some fault may be masked and cannot be isolated accurately. This paper proposed a new test case generation method based on the FSM net model. Instead of testing the global transitions of product machine, the generated test cases are used to verify the local transitions. We discuss the detailed methods of verifying the outputs and the tail states of the local transitions. Moreover, we prove that if all the local transitions are right, the transition structure of the distributed software is right. The tests are generated on the local transition structure of components, so we will not meet the state combinatorial explosion problem. All the outputs of the local transitions are checked, so the fault isolation results may be more accurate.**

***Index Terms*—Distributed software, finite state machine net, output identifying sequence, extended unique input/output sequence, test case generation**

I. INTRODUCTION

Distributed software system is usually composed of several components. These components are distributed in different computers and connected through network. As a result, testing becomes an important work for the validity and reliability of distributed software.

The presence of a formal model or specification, which defines the required behaviors of the software, introduces the possibility of automating or semi-automating much of the testing process, especially the generation of test case. This can lead to more effective and efficient testing.

There are many approaches to formally modeling or specifying a software system. Some formal methods have been used in software testing [1-7]. The formal methods based on finite state machine (FSM) model are widely studied and applied [2-14].

In this paper, we will study how to extend the FSM model on the test generation for distributed software. The distributed software may be more naturally and simply modeled by a set of FSMs, rather than a single FSM, which operate concurrently and may interact by changing messages. Then the behaviors of the software can be described by a product machine which is the equivalent single FSM constructed from the set of FSMs through product operation [15].

Tests can be generated from the product machine using standard FSM test techniques. It is assumed that the model of certain distributed software consists of FSMs $A_1,\cdots A_n$. Then the number of the product machine states is $\prod_i |A_i|$, where $|A_i|$ means the number of states of FSM A_i. Thus we may suffer from the state combinatorial explosion problem when computing the product machine.

Take the software in Fig. 1 for example. We suppose that the software is composed of two components, and each is modeled as a FSM, A_1, A_2. The transitions and

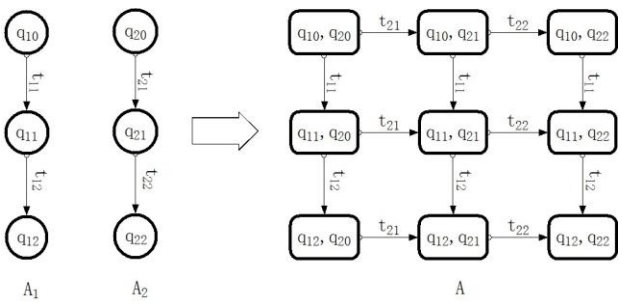

Figure 1. Combining the FSMs of the software components

doi:10.4304/jsw.6.2.175-183

states for each FSM are shown in Fig. 1.A_1 and Fig. 1.A_2. The product machine get through product operation is shown in Fig. 1.A, too. Each component FSM contains three states, and the product machine contains nine states. From this example, we can see that if the software has many components or each component contains many states, the state number of product machine may be too large to handle.

When applied to complex distributed system, the traditional test approach for single FSM mainly has the following problems:

1) State combinatorial explosion problem [16].
2) Unexecutable test sequence: The test sequence combining the local transitions may be unexecutable. If the specification of software constrains that the component A_1 can trigger transition t_{12} only after it receives the output of transition t_{21} , then the transition path $t_{11}, t_{12}, t_{21}, t_{22}$ cannot be carried out. Generally, the unexecutable test sequences are caused by the unreachable state of the product machine.
3) Fault isolation between synchronous transitions: In Fig.1, it is assumed that the output of t_{11} is sent to t_{21} and triggers it executing. Then the two transitions are defined as synchronous transitions. Only the output of t_{21} can be observed by the tester and the two transitions compose a global transition, thus the output of t_{11} is not checked. The fault isolation therefore may be a problem. Sometime the fault may be masked.

To solve these problems, Hierons proposed a method in [16]. His main idea is verifying the local transitions instead of the global transitions. But when we generate the verifying sequences, some parts of the product machine still need to be computed. Meanwhile, how to check the output of the transition when the output is sent to other component was not discussed in his study.

In this paper, we proposed a novel test case generation method for distributed software. By extending the FSM model, we set up the FSM net model as the formal test model for distributed software. We proved that if all the local transitions are right, the transition structure of the software is right. In order to verify the output of the local transition that will be sent to other component, a construction method of the output identifying sequence is proposed. We extend the unique input/output sequence to multiple components to verify the tail state of local transition.

Because we do not need to compute the product machine, we will not meet the state combinatorial explosion problem. All the test sequences are generated from the transition structure of software component, so they are all executable. Since all the outputs of the local transitions are checked, the fault isolation may be more accurate.

The rest of this paper is organized as follows. The basic idea of testing with FSM is introduced in Section 2 and the FSM net model are also presented in this section. How to check the local transition is introduced in Section 3. The verification of distributed software is discussed in Section 4. An example is given in Section 5. Finally, conclusion is presented in Section 6.

II. BASIC PRINCIPLES

For the sake of convenience, in this section we will recall the basic idea of testing with finite state machine (FSM), and then introduce the formal model which we will use to model the distributed software. At the end of this chapter, the concept of product machine will be introduced.

A. Testing with FSM

Usually the software under test can be modeled by a FSM or a set of FSMs that produce the outputs on its state transitions after receiving the inputs. When the software is modeled as a FSM, the testing of software can be taken as checking the output value of several sequences of input values. Usually, an input is given at an input port, and the outputs associated with the input can be observed at the output ports. The outputs will then be compared with the expected outputs corresponding with the inputs.

Definition 1: A finite state machine is a six-tuple[17]

$$FSM = (Q, q_0, \Sigma, \Lambda, \delta, \lambda) \quad (1)$$

1) Q is the finite set of states;
2) Σ is the finite set of inputs;
3) Λ contains all outputs;
4) $\delta : Q \times \Sigma \to Q$ is the set of state transition functions;
5) q_0 is the initial state;
6) $\lambda : Q \times \Sigma \to \Lambda$ is the set of output functions.

A FSM can be represented by a directed graph $G = (V, E)$, where the set $V = \{v_1, \cdots v_n\}$ of vertices represents the set of specified states Q of the machine and directed edges represents transitions from one state to another in it. An edge in G is fully specified by a triple $(v_i, v_j; L)$, where $L \equiv i_k / o_l$, $L^{(i)} \equiv i_k$ and $L^{(o)} \equiv o_l$. In this paper, it is assumed that G is strongly connected.

The verification of software is implemented through checking the output and the tail sate of every transition. The procedure for testing a specified transition from state q_i to state q_j with input/output i_k / o_l takes place in three steps:

1) The implementation is leaded into state q_i ;
2) Input i_k is applied and the output is checked to see whether it is o_l as expected, or not.
3) The new state of implementation is checked to verify that if the tail state of the specified transition is q_j as expected, or not.

It is assumed that there exist a reset action which is applied to make the software return to its initial state. This ensures that each test is applied in the same state of the implementation. The reset action might be a certain

sequences of the inputs or a single action such as a reset, or the system being closed off and then powered on again.

We also suppose that this implementation has a status message. For each state $q_i \in Q$, this message denotes the state uniquely, such as the unique input/output (UIO) sequence [6]. The tail state is verified by checking this message.

The test case for transition $(q_i, q_j; i_k / o_l)$ is constructed based on the U-method introduced in [14] as follows:

1) Constructing the reset action r to implementation I so that I can be reset to its initial state.
2) Generating the shortest transition sequence that can lead the implementation from state q_0 to state q_i, namely preamble sequence.
3) Applying the input i_k which can enable the transition to be tested.
4) Generating the verifying sequence of tail state q_j, namely postamble sequence.

B. FSM Net Model

The distributed software usually consists of several components and has multi-thread, distributed and parallel properties. Thus we should model such software as a set of FSMs, and each of components behaves as a FSM that may interact with other components. We call this set of FSMs a FSM net.

Definition 2: An finite state machine net is formally defined by a two-tuple $FSMnet = (A, C)$.

1) $A = \{A_1, A_2, \cdots A_n\}$ is the finite set of component finite state machines;
2) $C = \{c_{i,j} : i, j \le n \wedge i \ne j\}$ is the set of finite channel finite state machine, and $c_{i,j}$ is the channel FSM between software component A_i and component A_j.

A component FSM $A_i \in A$ is a classical FSM that is defined in the definition 1. Its states are called local states and its transitions are named local transitions, which are in contrast to global transition and global state that are defined on the product machine of component FSMs.

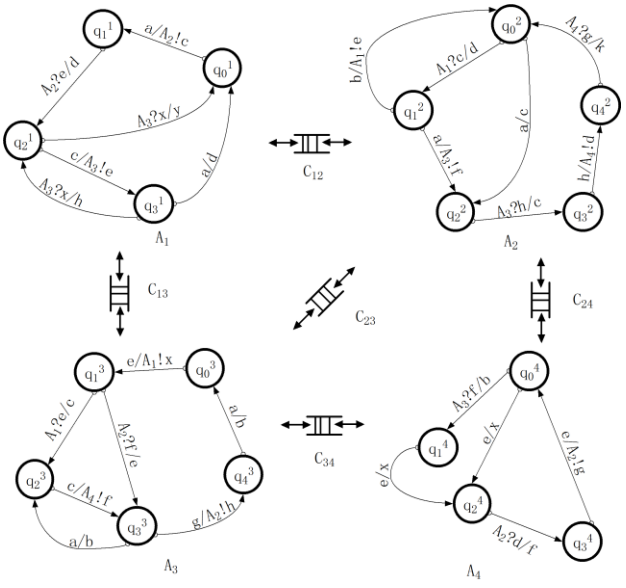

Figure 2. The FSM net model for a distributed software system.

We define channel FSMs to describe the message transporting behaviors between FSMs. The behaviors of $c_{i,j}$ are decided by the channel properties.

Definition 3: A channel FSM is a four-tuple $c_{i,j} = (Q_c, \delta_c, q_{c,0}, M_{ij})$, where:

1) Q_c is the set of finite channel states, its number is decided by the properties and the buffer limit of communication channel;
2) δ_c is the state transition function set of communication channel;
3) $q_{c,0}$ is the initial state of communication channel;
4) $M_{i,j}$ is the set of finite messages transporting through channel $c_{i,j}$. $m \in M_{ij}$ is one of the messages sent from A_i and A_j.

Since this paper focuses on the testing of the transition to verify the correctness of software, the properties of channel will not be discussed here.

According to its properties, the local transition of component FSM can be divided into three types:

1) Non-communication transition: the input of this type transition will be applied at the input port of this component and output can be observed at the output port. It is formally defined as $(q_i, q_j; i_k / o_l)$.
2) Sending message transition: the input of this type transition will be applied at the input port of this component, but the output will be sent to other components. This output is called internal output and cannot be observed by the tester. It is formally defined as $(q_i, q_j; i_k / A_j!o_{ij})$.
3) Receiving message transition: the input of this type transition is received from output of other component, but output can be observed at the output port of this component. When testing, the input of this type transition can be applied by both the tester and other component. It is defined as $(q_i, q_j; A_i?o_{ij} / o_k)$.

Where $A_j!o_{ij}$ means sending message o_{ij} to component A_j, and $A_i?o_{ij}$ means receiving message o_{ij} from A_i.

Example 1: A distributed software system is modeled as FSM net $FSMnet = (A, C)$ shown in Fig. 2, where

$$A = \{A_1, A_2, A_3, A_4\}; C = \{c_{1,2}, c_{1,3}, c_{2,3}, c_{2,4}, c_{3,4}\};$$
$$M_{12} = \{c, e\}; M_{13} = \{x, e\}; M_{23} = \{f, h\};$$
$$M_{24} = \{d, g\}; M_{34} = \{f\}.$$

C. The Product Machine

This paper deals with the problem of generating tests for the transition structure and thus shall only consider testing transitions from stable states. If input values will only be received in stable states, the full behavior of software is equivalent to a product machine [15].

We use $P(A)$ to denote the product machine generated from A, and we use X and Y to denote the input and output sets of $P(A)$. $S = Q_1 \times Q_2 \times \cdots \times Q_n$ is used to

denote the set of stable global states. $S(k)$ denotes the state of the kth component when global state is S. Clearly some elements in S may be unreachable. We therefore use S_r to denote the reachable state in S. The initial state of $P(A)$ is $s_0 = (q_0^1, q_0^2, \cdots q_0^3)$. The state transition functions and the output functions are also denoted as δ and λ. Thus $P(A)$ is defined by

$$P(A) = (S_r, s_0, X, Y, \delta, \lambda). \quad (2)$$

Given an input $\sigma \in \Sigma_k$ and the current global state $s_i \in S_r$, next state and functions δ and λ for the product machine are defined by the following ways.

If $\lambda(s_i(k), \sigma) \notin \bigcup_i \Sigma_i$, the next state is

$$s_{i+1} = \delta(s_i, \sigma) \quad (3)$$

where $\forall m \neq k,\ s_i(m) = s_{i+1}(m)$ and $s_{i+1}(k) = \delta(s_i(k), \sigma)$.

The output is

$$\lambda(s_i, \sigma) = \lambda(s_i(k), \sigma) \quad (4)$$

And this is called the first class global transition.

If $\lambda(s_i(k), \sigma) \in \bigcup_i \Sigma_i$, and the output will be sent to component A_h, such that the next state is

$$s_{i+1} = \delta(s_i^{'}, \lambda(s_i(k), \sigma)) \quad (5)$$

where $\forall m \neq k,\ s_i(m) = s_i^{'}(m)$ and $s_i^{'}(k) = \delta(s_i(k), \sigma)$.

The output is

$$\lambda(s_i, \sigma) = \lambda(s_i(h), \lambda(s_i(k), \sigma)) \quad (6)$$

And this is called the second class global transition.

The first case defines the behavior when the input triggers a non-communication transition and thus this transition forms a global transition. The second case defines the behavior when a sending message transition t is triggered (this is simply the behavior produced if t is executed and the output from t is fed back into A as an input). This process will be executed until a non-communication transition is triggered.

In the second case, the global transition is composed of the local transitions through the synchronous operation between sending message transition and receiving message transition.

Definition 6 synchronous operation: it defined as a transition will send a message to another transition. And we use □ to denote this operation.

Before execute synchronous operation, we apply an admissible preamble sub-sequence to the component that will receive message. Then this component can be leaded to the state that certain massage can be accepted.

Example 2: If we compute the product machine of the distributed software in example 1, its state number is 400. Although some of these states are unreachable, the reachable state number will be very large. The attempt of drawing the state transition graph is impossible. This state combinatorial explosion problem causes the traditional test approaches for single FSM fail for complex distributed software system. In this example, $(q_0^1, q_0^2, q_0^3, q_0^4)$ is the global initial state. $I_1(a)/O_1(c)$ is a first class global transition and

$I_1(a)/O_2(d) \,|\, (I_1(a)/O_1(A_2!c)) \square\ (I_2(A_1?c)/O_2(d))$ is a second class global transition by the synchronous operation between transition $I_1(a)/O_1(A_2!c)$ and transition $I_2(A_1?c)/O_2(d)$.

III. VERIFICANTION OF LOCAL TRANSITION

This section shall consider the problem of verification of local transitions.

For the sake of convenience, we first introduce the fault model of transition. A general survey on a variety of fault models in testing was given in [18]. We use $A_i = (Q_i, q_0^i, \Sigma_i, \Lambda_i, \delta, \lambda)$ to represent the required behaviors of component i and $I_i = (Q_i', q_0^i, \Sigma_i, \Lambda_i', \delta_i', \lambda_i')$ to represent the implementation behaviors of this component. We define the following two faulty types as the major faults that may be encountered when testing a transition:

1) Output fault: We say that a transition in I_i has a output fault if for $\forall q \in Q_i, q' \in Q'_i, a \in \Sigma_i$, following equation holds,
$$q = q' \wedge \delta_i(q,a) = \delta_i'(q',a) \wedge \lambda_i(q,a) \neq \lambda_i'(q',a). \quad (7)$$
2) Transfer fault: We say that I_i has a transfer fault if for $\forall q \in Q_i, q' \in Q'_i, a \in \Sigma_i$, following equation holds,
$$q = q' \wedge \delta_i(q,a) \neq \delta_i'(q',a) \wedge \lambda_i(q,a) = \lambda_i'(q',a). \quad (8)$$

Certainly, the two type faults can happen simultaneously.

A local transition is said to be a correct implementation when its output and tail state are all right. The verification of local transition is carried out based on the following hypothesis:

Hypothesis 1: The inputs will be applied at the stable state. This means that the inputs are applied when no internal events can occur.

Hypothesis 2: We assume that message transporting time is zero, then $s_i!m_{ij}$, $s_i?m_{ij}$, $s_j!m_{ij}$ and $s_j?m_{ij}$ are synchronous. This means that the behaviors of channel between two components are modeled as empty channel.

Hypothesis 3: It is assumed that when testing a transition, all the other transitions are correct.

Hypothesis 4: We suppose that the ports of different components can be distinguished by tester, so the same inputs applied to different components and same outputs observed from different components can be distinguished by tester and were seen as different inputs and outputs.

A. Verification of Local Transition

The verification of local transition is implemented through checking its output and its tail sate. Then the procedure for testing a specified local transition of A_i from state q_m^i to state q_n^i with input/output i_k / o_l takes place in three steps:

1) The implementation of software component A_i is leaded to state q_m^i;
2) Input i_k is applied and the output is checked to see whether it is o_l as expected, or not.

3) The new state of implementation is checked to verify that if the tail state of the specified transition is q_n^i as expected, or not.

This procedure is the same as that for transition of single FSM verification, but its operations in each step are different. We also assume that there exist a reset action which is applied to make the software return to its initial state.

The local transition sequence that can lead A_i from the state q_0^i to the state q_m^i is named the local preamble sequence. If the test cost is assigned to every transition, the least test cost preamble sequence can be obtained from the sub-graph of FSM A_i. Especially, the required inputs for the receiving message transitions are applied by tester not the corresponding component. Taking the testing of transition $I_2(a)/O_2(A_3!f)$ in A_2 for instance, the local preamble sequence $I_2(c)/O_2(d)$ is constructed for its verification. After the input i_k is applied, the output and tail state will be checked. The construction of the output checking sequence and the tail state verifying sequence will be discussed in following two sections respectively.

B. Checking the Output

The checking for the output of a local transition can be divided into two cases based on the type of transition.

The outputs of the receiving message transitions and the non-communication transitions can be observed by the tester and it is seen as a global output of system. Then the check of them is easy to be implemented through comparing them with required outputs.

The outputs of the sending message transition will be applied to other components and cannot be observed by the tester. We have assumed that when checking a transition all the other transitions are correct. Then we can observe that if the output of sending message transition is right, it will trigger another local transition belonging to other FSM A_j. This output can be denoted uniquely by the flowing input/output sequence. For example, the output of local transition $I_2(b)/O_2(A_1!e)$ can be denoted by the sequence $u(t)=I_1(A_2?e)/O_1(d),I_1(x)/O_1(y)$. When we input b to A_2, d is observed at A_1 (A_1 has been leaded to q_1^1) and when we input x to A_1, y is observed at A_1. These observations mean the output of transition $I_2(b)/O_2(A_1!e)$ must e, because when other output is generated by this transition, sequence $O_1(d),I_1(x)/O_1(y)$ cannot be observed by tester. $u(t)$ is an output identifying sequence (OIS) for local transition, if the following holds.

$$\{\lambda^*\{q_m^j,\cdots\},u(i))\,|\,u(i)_1=O_m\}\bigcap\{\lambda^*\{q_m^j,\cdots\},u'(i))\,|\,u'(i)_1\neq O_m\}=\varnothing \quad (9)$$

where $u(i)$ is the input sequence of $u(t)$; $u(i)_1$ is first input of this sequence which is the output of transition belonging to A_i.

So the verification of output of a sending message transition can be carried out through checking the OIS. The OIS is constructed as possible as by the transitions belonging to A_j. If the transition sequence belonging to A_j cannot denote the output uniquely, we can combine the transitions belonging to other components, such as the sequence $I_2(A_3?h)/O_2(c),I_2(h)/O_2(A_4!d)\square I_4(A_2?d)/O_4(f)$ for output $O_3(A_2!h)$ verifying of transition $I_3(g)/O_3(A_2!h)$.

C. Checking the Tail State

This section shall consider the problem of checking the tail state of a local transition.

The UIO sequence method was proposed in [6] to check the tail state of the transition in the single FSM. It is possible to extend it for the verification of the tail state of local transition in this paper. An local input sequence $u(i)$ in A_i is capable of verifying a local state q_m^i, if the following holds:

$$\{\lambda^*(q_m^i,u(i))\}\bigcap\{\lambda^*(q_n^i,u(i))\,|\,q_m^i\neq q_n^i\}=\varnothing .$$

Unfortunately, not every state in component A_i has a status message that is composed of the local transitions belonging to A_i. We extend the UIO sequence to multiple components, and then the UIO sequence can be composed of the local transitions belonging to different components. We noted this sequence extended unique input/out sequence (EUIO). It is the "status message" for local state.

An extended input sequence $u(i)$ is capable of verifying a local state q_m^i, if the following holds:

$$\{\lambda^*(\{q_m^i,\cdots\},u(i))\}\bigcap\{\lambda^*(\{q_n^i,\cdots\},u(i))\,|\,q_m^i\neq q_n^i\}=\varnothing .(10)$$

Let us see the software in example 1. The local state q_4^3 in A_3 does not have UIO sequence composed of the local transitions belonging A_3, but EUIO sequence

$$I_3(a)/O_3(b),(I_3(e)/O_3(A_1!x))\square(I_1(A_3?x)/O_1(h)) \quad (11)$$

can denote the state q_4^3 uniquely. This means q_4^3 is the unique state after $I_3(a),I_3(e)$ is applied $O_3(b),O_1(h)$ is observed by tester.

D. Test Sequences for Lcal Transition

Based on previous discussions, the test case for local transition $(q_m^i,q_n^i;i_k/o_l)$ contains two test sequences, and is constructed in following ways:

The test sequence for the output checking (required only when the outputs are sent to other component and cannot be observed by tester):

1) Constructing the reset action r to implementation I so that the software under test can be reset to its initial state;
2) Generating the least test cost preamble sequence for local transition to be tested;
3) Generating the preamble sequence for the synchronous transition of the transition to be tested which has the least test cost;
4) Applying the input i_k which can enable the local transition to be tested;

5) Generating the OIS for output o_l. (if it contains some synchronous operations in it, the admissible preamble sequences for all operations are needed)

The test sequence for the tail state verifying:

1) Constructing the reset action r to implementation I so that I can be reset to its initial state.
2) Generating the shortest transition sequence that can lead the machine from state q_0 to state q_i, namely preamble sequence.
3) Applying the input i_k which can enable the transition to be tested.
4) Generating the verifying sequence of tail state q_n^i.

For the I with the extended status message feature, the test case for local transition in A_i is of the form

$$\begin{aligned} ts_1 &: r, ts_{pre}^{test}, t, ts_{post}^{state}; \\ ts_2 &: r, ts_{pre}^{test}, ts_{pre}^{synchronous}, t, ts_{post}^{output}. \end{aligned} \tag{12}$$

where r is the reset action; ts_{pre}^{test} is the preamble sequence for the transition t; $t = (q_m^i, q_n^i; i_k / o_l)$ is the local transition to be tested; ts_{post}^{state} is the postamble sequence to check the tail state of transition t; $ts_{pre}^{synchronous}$ is the preamble sequence for synchronous operation; ts_{post}^{output} is the postamble sequence to check the output of transition t.

IV. Verification of Distributed Software

As discussed in [6], the verification of software can be implemented through verifying all transitions. Therefore, the verification of distributed software can be carried out through checking of outputs and tail states of all global transitions.

In this section, we will prove that the checking of all local transitions has the same ability to validate the correctness of software by checking of all the global transitions.

Lemma 1: If all the local transitions are right, all the global transitions are right.

Proof.

Given a global transition $s_j = \delta(s_i, \sigma)$, and we let $\sigma \in \Sigma_k \in X$ and

$$s_i, s_j \in S \mid s_i = (q_i^1, q_i^2, \cdots, q_i^3), s_j = (q_j^1, q_j^2, \cdots, q_j^3).$$

We consider the two class global transitions.

1) For the first class, a non-sending message transition forms a global transition. Given a local transition $(q_i^m, q_j^m; i_k / o_l)$ of A_m, then the global transition is $(s_i, s_j; i_k / o_l)$, where $\forall k \in (1,n), k \neq m, s_i(k) = s_j(k)$ and $s_i(m) = q_i^m$, $s_j(m) = q_j^m$.

Now, we prove that for this class, if the local transition is right, the global transition is right. We will use apagoge to prove it.

When the local transition has an output fault, $o'_l \neq o_l$, the global output is o'_l. So there is an output fault of global transition. When the local transition has a tail state fault, $(q_i^m, q'^m_j), q'^m_j \neq q_j^m$, the global state transition is $(s_i, s'_j), s'_j \neq s_j$. So there is a tail state fault of the global transition. Thus a global transition is right only if the local transition is right.

Then we show that the faults of global transition can be detected by checking the local transition.

Obviously, the output fault can be checked through checking the output of the local transition which forms this global transition. If the tail state fault is $(s_i, s'_j), s'_j \neq s_j, s_j'(m) \neq s_j(m)$, this fault can be detected through checking the tail state of the local transition which forms this global transition. If the tail state fault is $s'_j \neq s_j, s_j'(n) = s_j(n), n \neq m$, this means a transition of other component has been triggered. This must be caused by an output fault of local transition. The non-communication local transition might send a message to component A_n. This can be detected by checking the output of local transition.

Up to here, we prove that for the first class global transition, the verification of global transition can be implemented through checking corresponding local transition.

2) For the second class, more than one communication transitions form the global transition through synchronous operation. We only need to prove the situation in which the global transition is composed of two transitions. The situation having more two transitions can be proved recursively by the situation of two transitions.

Given a sending message transition $(q_i^m, q_j^m; i_k / A_j ! o_{mn})$ in A_m and a receiving message transition $(q_i^n, q_j^n; A_n ? o_{mn} / o_l)$ in A_n then the global transition is $(s_i, s_j; i_k / o_l)$, where $s_i(m) = q_i^m$, $s_j(m) = q_j^m$, $s_i(n) = q_i^n$, $s_j(n) = q_j^n$ and $\forall k \in (1,n)$, $k \neq m$, $k \neq n$, $s_i(k) = s_j(k)$.

First, we prove that if the two local transitions are all right, the global transition is right. We also use apagoge to prove it.

When any of the local transitions has an output fault, the global output is not o_l. So there is an output fault of global transition. When any of the local transitions has a tail state fault, $(q_i^m, q'^m_j), q'^m_j \neq q_j^m$ or $(q_i^n, q'^n_j), q'^n_j \neq q_j^n$, the global state transition is $(s_i, s'_j), s'_j \neq s_j$. So there is a tail state fault of global transition. Thus a global transition is right only if the two local transitions are all right.

Second, we show the fault of global transition can be detected by checking the local transitions.

Obviously, the output fault can be checked the output of local transition (q_i^m, q_j^m) and (q_i^n, q_j^n). If the tail state fault is (s_i, s'_j), $s'_j \neq s_j \mid s_j'(m) \neq q_j^m$, the fault can be detected through checking the tail state of local transition (q_i^m, q_j^m). If the tail state fault is (s_i, s'_j),

$s'_j \neq s_j \mid s_j'(n) \neq q_j^n$, the fault can be detected through checking the tail state of local transition (q_i^n, q_j^n). If the tail state fault is $s'_j \neq s_j$, $s_j'(k) \neq s_j(k) \mid k \neq m, k \neq n$, this means a transition of other component has been triggered. This must be caused by the output fault of the local transition. The destination of sending message transition in A_m is then wrong or the receiving message transition in A_n sends a message to other component. This can be detected by checking the outputs of the two local transitions, too.

In short, we prove that for the second class global transition, the verification of global transition can be implemented through checking all the corresponding local transitions.

Based on the above discussions, we can conclude that the verification of global transition can be implemented by checking of local transitions. In other words, if all the local transitions belonging to software system are all right, the implementation of the distributed software system is right.

V. EXPERIMENTS

In this section, we will discuss the test case generation for the software in Example 1 to exam our approach.

A. Test Cases

Using the proposed test case generation method, we first indentify the EUIO sequences for all the local states and the results are shown in Table I. More than one EUIO can denote the state uniquely and we only list the shortest one in the table. The OISs for all communication outputs are listed in Table II. Then the generated test cases for all local transitions are shown in Table III.

TABLE I.
EXTENDED UNIQUE INPUT/OUTPUT

State	EUIO
q_0^1	$(I_1(a)/O_1(A_2!c))\ (I_2(A_1?c)/O_2(d))$
q_1^1	$I_1(e)/O_1(d)$
q_2^1	$I_1(x)/O_1(y)$
q_3^1	$I_1(x)/O_1(h)$
q_0^2	$I_2(c)/O_2(d)$
q_1^2	$(I_2(a)/O_2(A_3!f))\ (I_3(A_2?f)/O_3(e))$
q_2^2	$I_2(h)/O_2(c)$
q_3^2	$I_2(h)/O_2(A_4!d)\ I_4(A_2?d)/O_3(f)$
q_4^2	$I_2(g)/O_2(k)$
q_0^3	$(I_3(e)/O_3(A_1!x))\ (I_1(A_3?x)/O_1(y))$
q_1^3	$I_3(e)/O_3(c)$
q_2^3	$I_3(c)/O_3(A_4!f)\ I_4(A_3?f)/O_3(b)$
q_3^3	$I_3(g)/O_3(A_2!h)\ I_2(A_3?h)/O_2(c)$
q_4^3	$I_3(a)/O_3(b), (I_3(e)/O_3(A_1!x))\ (I_1(A_3?x)/O_1(h))$
q_0^4	$I_4(f)/O_4(b)$
q_1^4	$I_4(e)/O_4(x), I_4(d)/O_4(f)$
q_2^4	$I_4(d)/O_4(f)$
q_3^4	$(I_4(e)/O_4(A_2!g))\ (I_2(A_4?g)/O_2(k))$

TABLE II.
OUTPUT VERIFYING SEQUENCE

Internal output	OIS
$O_1(A_2!c)$	$I_2(A_1?c)/O_2(d)$
$O_1(A_3!e)$	$I_3(A_1?e)/O_3(c)$
$O_2(A_1!e)$	$I_1(A_2?e)/O_1(d), I_1(x)/O_1(y)$
$O_2(A_3!f)$	$I_3(A_2?f)/O_3(e)$
$O_2(A_4!d)$	$I_4(A_2?d)/O_3(f)$
$O_3(A_1!x)$	$I_1(A_3?x)/O_1(y)$
$O_3(A_4!f)$	$I_4(A_3?f)/O_3(b)$
$O_3(A_2!h)$	$I_2(A_3?h)/O_2(c), I_2(h)/O_2(A_4!d)\ I_4(A_2?d)/O_4(f)$
$O_4(A_2!g)$	$I_2(A_4?g)/O_2(k)$

B. Discussions

In the introduction chapter, we have pointed out the potential problems when the traditional test method for single FSM is applied to the complex distributed software system. In this section we will discuss if our method can benefit these problems.

1) State combinatorial explosion problem: it is not necessary to compute the product machine using our method when generating test cases, so we will not meet the state combinatorial explosion problem.
2) Unexecutable test sequence: all the test sequences are generated based on the transition structure of the local component and synchronous operation, so all of them are executable.
3) Test cost: a local transition can form more than one global transitions, such as :
   $((q_0^1, q_0^2, q_0^3, q_0^4), (q_1^1, q_0^2, q_0^3, q_0^4); a/A_2!c)$ and
   $((q_0^1, q_2^2, q_0^3, q_0^4), (q_1^1, q_2^2, q_0^3, q_0^4); a/A_2!c)$
   are both formed by local transition $a/A_2!c$. With our method, only one local transition needs to be verified, but using traditional methods for single product machine, more than one global transition need to be verified. So the test cost of our method may be less than traditional methods.
4) Fault isolation between synchronous transitions: the outputs of sending message transitions can be checked by the OIS, but this cannot be implemented by the product machine method.

In summary, our method is an efficient method to solve the problems that encountered by the single product machine method.

VI. CONCLUSION

When we use traditional test methods for distributed software testing through computing the product machine, we will suffer from the state combinatorial explosion problem and some generated test sequences may be unexecutable. Besides, some outputs of the sending

message transition cannot be checked. In this paper, we proposed a new test case generation method based on FSM net model. The construction of the output identifying sequence is used to verify the output of the local transition. Considering the limitation of the UIO sequence, we extend it to multiple components. Then the EUIO sequence can be composed of the local transitions belonging to different components, and this makes more

TABLE III.
TEST CASES

	Transition	Test case
1	$I_1(a)/O_1(A_2!c)$	$Ts_1^1: I_1(a)/O_1(A_2!c), I_1(e)/O_1(d); Ts_2^1: I_1(a)/O_1(A_2!c)\ I_2(A_1?c)/O_2(d);$
2	$I_1(A_2?e)/O_1(d)$	$Ts_1^2: I_1(a)/O_1(A_2!c), I_1(e)/O_1(d), I_1(x)/O_1(y);$
3	$I_1(A_3?x)/O_1(y)$	$Ts_1^3: I_1(a)/O_1(A_2!c), I_1(e)/O_1(d), I_1(x)/O_1(y);$
4	$I_1(c)/O_1(A_3!e)$	$Ts_1^4: I_1(a)/O_1(A_2!c), I_1(e)/O_1(d), I_1(c)/O_1(A_3!e), I_1(x)/O_1(h);$ $Ts_2^4: I_1(a)/O_1(A_2!c), I_1(e)/O_1(d), I_3(e)/O_1(A_1!x), I_3(e)/O_3(A_1!x), I_1(c)/O_1(A_3!e)\ I_3(A_1?e)/O_3(c)$
5	$I_1(A_3?x)/O_1(h)$	$Ts_1^5: I_1(a)/O_1(A_2!c), I_1(e)/O_1(d), I_1(c)/O_1(A_3!e), I_1(x)/O_1(h);$
6	$I_1(a)/O_1(d)$	$Ts_1^6: I_1(a)/O_1(A_2!c), I_1(e)/O_1(d), I_1(c)/O_1(A_3!e), I_1(a)/O_1(d);$
7	$I_2(A_1?c)/O_2(d)$	$Ts_1^7: I_2(c)/O_2(d), I_3(e)/O_3(A_1!x), (I_2(a)/O_2(A_3!f))\ (I_3(A_2?f)/O_3(e));$
8	$I_2(a)/O_2(c)$	$Ts_1^8: I_2(a)/O_2(c), I_2(h)/O_2(c);$
9	$I_2(b)/O_2(A_1!e)$	$Ts_1^9: I_2(c)/O_2(d), I_2(b)/O_2(A_1!e), I_2(c)/O_2(d);$ $Ts_2^9: I_2(c)/O_2(d), I_1(a)/O_1(A_2!c), I_2(b)/O_2(A_1!e)\ I_1(A_2?e)/O_1(d), I_1(x)/O_1(y);$
10	$I_2(a)/O_2(A_3!f)$	$Ts_1^{10}: I_2(c)/O_2(d), I_2(a)/O_2(A_3!f), I_2(h)/O_2(c);$ $Ts_2^{10}: I_2(c)/O_2(d), I_3(e)/O_3(A_1!x), I_2(a)/O_2(A_3!f)\ I_3(A_2?f)/O_3(e);$
11	$I_2(A_3?h)/O_2(c)$	$Ts_1^{11}: I_2(a)/O_2(c), I_2(h)/O_2(c), I_4(e)/O_3(x), I_2(h)/O_2(A_4!d)\ I_4(A_2?d)/O_3(f);$
12	$I_2(h)/O_2(A_4!d)$	$Ts_1^{12}: I_2(a)/O_2(c), I_2(h)/O_2(c), I_2(h)/O_2(A_4!d), I_2(g)/O_2(k);$ $Ts_2^{12}: I_2(a)/O_2(c), I_2(h)/O_2(c), I_4(e)/O_3(x), I_2(h)/O_2(A_4!d)\ I_4(A_2?d)/O_3(f);$
13	$I_2(A_4?g)/O_2(k)$	$Ts_1^{13}: I_2(a)/O_2(c), I_2(h)/O_2(c), I_2(h)/O_2(A_4!d), I_2(g)/O_2(k), I_2(c)/O_2(d);$
14	$I_3(e)/O_3(A_1!x)$	$Ts_1^{14}: I_3(e)/O_3(A_1!x), I_3(e)/O_3(c); Ts_2^{14}: I_1(a)/O_1(A_2!c), I_1(e)/O_1(d), (I_3(e)/O_3(A_1!x))\ (I_1(A_3?x)/O_1(y));$
15	$I_3(A_1?e)/O_3(c)$	$Ts_1^{15}: I_3(e)/O_3(A_1!x), I_3(e)/O_3(c), I_3(c)/O_3(A_4!f)\ I_4(A_3?f)/O_3(b);$
16	$I_3(A_2?f)/O_3(e)$	$Ts_1^{16}: I_3(e)/O_3(A_1!x), I_3(f)/O_3(e), I_3(a)/O_3(b);$
17	$I_3(c)/O_3(A_4!f)$	$Ts_1^{17}: I_3(e)/O_3(A_1!x), I_3(e)/O_3(c), I_3(c)/O_3(A_4!f), I_3(a)/O_3(b);$ $Ts_2^{17}: I_3(e)/O_3(A_1!x), I_3(e)/O_3(c), I_3(c)/O_3(A_4!f)\ I_4(A_3?f)/O_3(b)$
18	$q_4^3, q_0^3; I_3(a)/O_3(b)$	$Ts_1^{18}: I_3(e)/O_3(A_1!x), I_3(f)/O_3(e), I_3(a)/O_3(b), I_3(c)/O_3(A_4!f)\ I_4(A_3?f)/O_3(b);$
19	$I_3(g)/O_3(A_2!h)$	$Ts_1^{19}: I_3(e)/O_3(A_1!x), I_3(f)/O_3(e), I_3(g)/O_3(A_2!h), I_3(a)/O_3(b), I_1(a)/O_1(A_2!c),$ $I_1(e)/O_1(d), (I_3(e)/O_3(A_1!x))\ (I_1(A_3?x)/O_1(h));$ $Ts_2^{19}: I_3(e)/O_3(A_1!x), I_3(f)/O_3(e), I_2(a)/O_2(c), I_3(g)/O_3(A_2!h)\ I_2(A_3?h)/O_2(c),$ $I_4(e)/O_4(x), I_2(h)/O_2(A_4!d)\ I_4(A_2?d)/O_2(f);$
20	$q_3^3, q_2^3; I_3(a)/O_3(b)$	$Ts_1^{20}: I_3(e)/O_3(A_1!x), I_3(f)/O_3(e), I_3(g)/O_3(A_2!h), I_3(a)/O_3(b), I_1(a)/O_1(A_2!c), I_1(e)/O_1(d),$ $(I_3(e)/O_3(A_1!x))\ (I_1(A_3?x)/O_1(y))$
21	$I_4(A_3?f)/O_4(b)$	$Ts_1^{21}: I_4(f)/O_4(b), I_4(e)/O_4(x), I_4(d)/O_4(f)$
22	$q_0^4, q_2^4; I_4(e)/O_4(x)$	$Ts_1^{22}: I_4(e)/O_4(x), I_4(d)/O_4(f)$
23	$q_1^4, q_2^4; I_4(e)/O_4(x)$	$Ts_1^{23}: I_4(f)/O_4(b), I_4(e)/O_4(x), I_4(d)/O_4(f)$
24	$I_4(A_2?d)/O_4(f)$	$Ts_1^{24}: I_4(f)/O_4(b), I_4(e)/O_4(x), I_4(d)/O_4(f), I_2(a)/O_2(c), I_2(h)/O_2(c), I_2(h)/O_2(A_4!d),$ $(I_4(e)/O_4(A_2!g))\ (I_2(A_4?g)/O_2(k))$
25	$I_4(e)/O_4(A_2!g)$	$Ts_1^{25}: I_4(f)/O_4(b), I_4(e)/O_4(x), I_4(d)/O_4(f), I_4(e)/O_4(A_2!g), I_4(f)/O_4(b);$ $Ts_2^{25}: I_4(f)/O_4(b), I_4(e)/O_4(x), I_4(d)/O_4(f), I_2(a)/O_2(c), I_2(h)/O_2(c), I_2(h)/O_2(A_4!d),$ $(I_4(e)/O_4(A_2!g))\ (I_2(A_4?g)/O_2(k))$

local states have status messages.

The tests are generated on the local transition structure of components, so we will not meet the state combinatorial explosion problem. By applying the admissible preamble sub-sequence of all synchronous operations, all the test sequences are executable. All the outputs of the local transitions can be checked by OIS, so the fault isolation may be more accurate.

The experiment in section 5 shows that this method has better properties than single product machine method. It is a promising way for distributed software testing.

ACKNOWLEDGMENT

This work was supported in part by the National Key Technology R&D Program under Grant 2009BAG12A08, the R&D Foundation of the Ministry of Railways under Grant 2009X003 and the Research Foundation of Beijing National Railway Research and Design Institute of Signal and Communication.

REFERENCES

[1] S. Ghosh, A. P. Mathur, "Issues in testing distributed Component-Based Systems," Proceedings of the First Interational ICSE Workshop Testing Distributed Component-based System, May, 1999.

[2] I.-H. Cho, J. D. McGregor, "Component specification and testing interoperation of components," Proc. of the IASTED Int'l Conf., Software Engineering and Applications, pp.27--31, 1999.

[3] I.-H. Cho, J. D. McGregor, "A formal approach to specifying and testing the interoperation between components," Proceedings of the 38th annual on Southeast regional conference, pp.161-170, 2000.

[4] A. Petrenko and N. Yevtushenko, "Test suite generation from a FSM with a given type of implementation errors,'' Proc. IFIP 12th Int. Symp. Protocol specification. Testing, and Verification XII, North-Holland, pp. 229-243, 1992.

[5] H. S. Hong , I. Lee , O. Sokolsky, "Automatic Test Generation from Statecharts Using Model Checking," Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software, pp.31-36, 2001.

[6] Y. Choi, D. Kim, J. Kim, and et al., "Protocol test sequence generation using UIO and BUIO," 1995 IEEE International Conference on communications, pp. 362-366, 1995.

[7] S. Fujiwara, G. v. Bochmann, F. Khendek and et al., "Test selection based finite state models," IEEE Transactions On Software Engineer, vol.17, Issue 6, pp.591-603, 1991.

[8] S. C. Pinto Ferraz Fabbri, M. E. Delamaro, J. C. Maldonado and et al., "Mutation analysis testing for finite state machine," Proceedings of 5th International Symposium on Software Reliability Engineer, pp.220-229, 1994.

[9] S. T. Chanson, Q. Li, "On static and dynamic test case selections in protocol conformance testing," The 5th Int'l Workshop on Protocol Test Systems, pp.225-267, 1992.

[10] D. P. Sidhu and T. K. Leung, "Formal methods for protocol testing: A detailed study," IEEE Trans. Software Eng., vol. 15, Issue 4, pp. 413-426, Apr. 1989.

[11] T.Chow, "Testing Software Design Modeled by Finite Machines," IEEE Transaction on Software Engineering, vol. 4, pp.178-187, 1978.

[12] D. Drusinsky, "Model checking of statecharts using automatic white box test generation," Circuits and Systems, 2005. 48th Midwest Symposium on, pp.327-332, 2005.

[13] A. V. Aho, A. T. Dahbura, D. Lee and M. U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours," Protocol Specification, Testing, and Verification VIII, pp.75-86, 1988.

[14] G. v. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, pp.109-124, 1994.

[15] G. Luo, G. v. Bochmann, and A. Petrenko, "Test selection based on communicating nondeterministic finite state machines using a generalized Wp-method," IEEE Transactions on Software Engineering, vol. 20, pp.149-162, 1994.

[16] R.M. Hierons, "Checking States and Transitions of A Set of Communication finite state machines," Microprocessors and Microsystems, vol. 24, pp.443-452, 2001.

[17] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to automata theory, languages, and computation. 2nd ed., Pearson Education, 2000.

[18] G. v. Bochmann, A. Das, R. Dossouli, M. Dubuc and et al., "Fault model in testing", Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV, pp.17-30, October 15-17, 1991.

Shuai Wang was born in Changchun, Jilin Province, China, on April 3, 1981. He received his B.S. degree in control science and engineering from Beijing Institute of Technology University, Beijing, China in 2004. Currently, he is a PH.D candidate working in fields of control science and engineering at Tsinghua University. His major research interests include system test, fault diagnosis and reliability analysis.

Yindong Ji was born in Beijing, China in 1962. He received his B.S. and M.S. all from the Department of Automation, at Tsinghua University, in 1985 and 1989, respectively. His main research areas are digital signal process, fault diagnosis, modeling & simulations. He is a member of IEEE.

Prof. JI is with the Department of Automation, and Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China. He has published over 60 papers in journals. His current research interest is in the area of train control system of high speed railway.

Shiyuan Yang was born in Shanghai, China, in 1945. He received his B.S. and M.S degree from Tsinghua University in 1970 and 1981, respectively.

Currently, he is a Professor in automation of department in Tsinghua University. He is an Associate Director of the FTC committee, China. His main research interests are home automation network, test technology, electronic technology application, system fault diagnosing.