2012 IEEE Asia-Pacific Services Computing Conference

A Model Checking Based Approach to Automatic Test Suite Generation for Testing Web Services and BPEL

Huiqun Zhao
Department of Computer Science
North China University of Technology
Beijing, China
Zhaohq6625@sina.com

Jing Sun
Department of Computer Science
North China University of Technology
Beijing, China
Sunjing8248@163.com

Xiaodong Liu
School of Computing
Edinburgh Napier University
Edinburgh, United Kingdom
x.liu@napier.ac.uk

***Abstract* — With the rapid increase of Web Service applications, the reliability of web service and service composition has drawn particular attention from researchers and industries. Many methods for testing and verifying the reliability have been discussed, however, the existing methods are weak in test automation and therefore difficult in tackling the dynamic features of modern SOA based application. The traditional method of model checking and the technique of test suite generation have a large potential for the reliability verification of web services and service composition. In this paper, an approach to integrating the test suite generation technique with model checking is presented. The approach takes advantage of model checking to verify BPEL script at the logical level, and to generate test suite automatically based on the model description, and finally to select test cases with respect to the counterexamples of model checking output. The approach contributes a set of algorithms and its implementation to support a translation from BPEL to LOTOS, and LTS(Labeled Transition Systems, LTS in short) to TTCN(Test and Testing Control Notation, TTCN in short) behavior Tree. Finally, a case study is presented to demonstrate and verify the proposed approach.**

Keywords – BPEL Model Checking, Web Service Testing, Automatic Test, Test Case Generation, LOTOS, and TTCN-3.

I. INTRODUCTION

Web services as a software system are designed to support interoperable machine-to-machine interaction over a network [1]. Usually the user specified composite Web Services are located at different places to implement a business process. In order to facilitate the users, IBM published a Business Process Execution Language [2] (BPEL in short) for compositing Web Services. BPEL as a de-facto standard for web service orchestration has drawn particularly attention from researchers and industries. However, BPEL as a semi-formal flow language has complex features such as distributed architecture, asynchronous behavior and lack of user interface. All these features also lead to concerns regarding their trustworthiness because verification and testing activities are dramatically affected. Many researchers from the software testing domain have made great efforts to solve these problems.

A recent survey by Mustafa Bozkurt, Mark Harman and Youssef Hassoun [3] summarizes the techniques of testing web services and classifies the research undertaken into 7 categories: partition testing of web services, unit testing, model-based testing and formal verification of web services, contract-based testing, regression testing, interoperability testing and integration testing. With an intensive investigation, we conclude that the following issues need to be studied urgently:

- The frequency of testing required;
- Testing without disrupting the operation of the service;
- Determining when testing is required and which operations need to be tested.

As a verification technique, Model Checking may play an important role for guaranteeing web service reliability. Model checking allows a model checker to visits all reachable states of the model and verifies whether the expected system properties, specified in temporal logic formulae, are satisfied over each possible path. If a property is not satisfied, the model checker attempts to generate a counterexample in the form of a trace as a sequence of states [4].

Franck van Breugel and Maria Koshkina [5] summarized the recent work on modeling and Web Service verification techniques, and classified it into the following five types: 1) approaches based on Petri Net; 2) approaches based on SPIN; 3) approaches based on process algebra; 4) approaches based on abstract state machine; and 5) approaches based on automata. However, in many practical cases there is a so-called state space explosion that causes the number and/or the length of the traces to be larger than which can be dealt with. In such a case, one has to choose which traces are and which are not to be checked. This selection of interesting traces requires much insight into the problem at hand, and so cannot be automated. Still, support in this process will be useful. Other drawbacks include that the model checking only works at the logical level of BPEL. It can not be a substitute of test technique.

An approach to combining the testing techniques with model checking will be a promising solution. Such an approach as we proposed takes advantages of model checking to implement the reliability verification at logical level of BPEL, and plenty of counterexamples created by adjusting the property model of BPEL are collected to automate the test case generation, and to test the web services involved in BPEL after model checking and before the BPEL is published. This approach can clear the barriers off either in testing web services or the model checking of the BPEL made.

978-0-7695-4897-5/12 $26.00 © 2012 IEEE
DOI 10.1109/APSCC.2012.29

IEEE computer society

The structure of this paper is as follows. In Section 2 background knowledge is introduced including LTS, TTCN Behavior Tree and μ-calculus. The equivalence between LTS and Behavior Tree is discussed in Section 3. By proving the equivalence an algorithm for translating LTS into Behavior Tree is proposed. In Section 4, we expose our novel approach in applications. An algorithm for generating TTCN test suite from counterexample of model checking is presented. To evaluate its effectiveness, a case study of verifying and testing BPEL reliability is introduced. Finally, we summarize related work and conclude our investigation.

II. INTRODUCTION OF LTS, TTCN AND M-CALCULUS

To make our discussion easy to understand, the background knowledge is introduced in this section.

A. Labeled Transition Systems (LTS)

A labeled transition system [6][7] consists of a collection of states and a collection of transitions between them. The transitions are labeled by actions from a given set A that happen when the transition is taken, and the states may be labeled by predicates from a given set P that hold in that state.

Definition 2.1 Let A and P be sets of actions and predicates respectively. A labeled transition system over A and P is a tuple (S, I,→, |=).

Where:

1) S is a collection of states

2) I∈S is a distinguishable member of S called the "Initial state"

3) → is a collection of binary relations $\xrightarrow{a} \subseteq S \times S$ called transition, we denote with $s \xrightarrow{a} t$ s,t∈S and a∈A is an operation.

4) |= is also a collection of binary relations |=⊆ S × P. s |= p says that predicate p∈P holds in state s∈S.

LTSs with a singleton (i.e. with → a single binary relation on S) are known as Kripke structures, the models of modal logic. General LTSs (with A arbitrary) are the Kripke models for polymodal logic. The name "labeled transition system" is employed in concurrency theory. Therefore, the elements of S represent the systems one is interested in, and $s \xrightarrow{a} t$ means that system s can evolve into system t while performing the action a. This approach identifies states and systems: the states of a system s are the systems reachable from s by following the transitions. In this realm |= is often encoded by μ-calculus [8].

Definition 2.2 (Label of LTS)[7]**:** Let LTS=(S, I, →, |=) be a LTS, call L(LTS)⊆A× S×P={<a,s,p>| $t \xrightarrow{a} s$ } a Label Class, L(LTS) or even L for short. Specially, When L(LTS) are coded by **STATE FORMULAS** of μ-calculus, call L(LTS) a Label Instance of LTS, note as LI(LTS) or LI.

For example suppose that a S={0,1,2}, I={0}, →={<0,1>, <1,1>*, <1,2>} P={True, False}, A is input action set such as **input *digital***, for convenience here note A={0,1,2…,9}. A Label Instance LI(LTS)={<1,0, True>, <0,1,True>*, <0, 2, False>} is digital numbers 1,0 and 0 is received.

B. μ-calculus[8]

The μ-calculus is a type of propositional modal logic. It is used to describe properties of labeled transition systems and for verifying these properties. Many temporal logics can be encoded in the μ-calculus including CTL* and its widely used fragments—linear temporal logic and computational tree logic. In this section the main properties of the μ-calculus will be introduced for the strategy of selecting test case associated with Model Checking counterexample.

In the μ-calculus uses **STATE FORMULAS** to describe a predicated condition in which a state must be achieved. A *state formula* is a logical formula built from Boolean, modal, and fixed point operators, according to the grammar below:

F **::= "true"**| **"false"**| **"not"** *F*| *F1* **"or"** *F2*| *F1* **"and"** *F2*| *F1* **"implies"** *F2*| *F1* **"equ"** *F2*| **"<"***R* **">"** *F*| **"["** *R* **"]"** *F*| **"@" "("** *R* **")"**| *X*| **"mu"** *X* **"."** *F*| **"nu"** *X* **"."** *F*

The F is a *regular formula* is a logical formula built from action formulas and the traditional regular expression operators, according to the grammar below:

R **::=** *A*| **"nil"**| *R1* **"."** *R2*| *R1* **"|"** *R2*| *R* **"*"**| *R* **"+"**.

The A is an *action formula* is a logical formula built from basic action predicates and boolean connectives, according to the grammar below: *A* **::=** *string*| *regexp*| **"true"**| **"false"**| **"not"** *A*| *A1* **"or"** *A2*| *A1* **"and"** *A2*| *A1***"implies"** *A2*| *A1* **"equ"** *A2*

For example, informally, a safety property expresses that "something bad never happens." Typical safety properties are those forbidding "bad" execution sequences in the LTS. These properties can be naturally expressed using box modalities containing regular formulas. For instance, mutual exclusion can be characterized by the following formula:

[true* . "OPEN !1" . (not "CLOSE !1")* . "OPEN !2"] false

which states that every time process 1 enters its critical section (action "OPEN !1"), it is impossible that process 2 also enters its critical section (action "OPEN !2") before process 1 has left its critical section (action "CLOSE !1"). Other typical safety properties are the *invariants*, expressing that every state of the LTS satisfies some "good" property.

C. TTCN Behavior Tree

Behavior Trees are a formal, graphical modeling language used primarily in systems and software engineering [9]. As a primary technique for software testing, the TTCN[10] employs Behavior Trees as behavioral description of Software system Under Test (SUT in short).

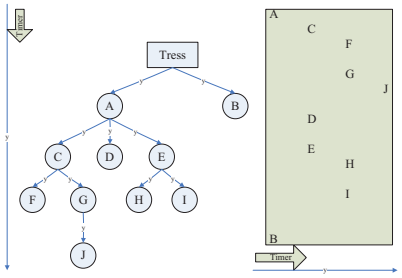

Fig1. TTCN Behavior Trees

The left of above fig. shows a tree structure, it captures a process of interactions between TTCN test suite and SUT through Point of Control and Observation (PCO in short). All sibling nodes belong to same subtree and present alternative behavior. For the above example it has five alternative sets: {A,B},{C,D,E},{F,G},{H,I},{J}. Note that the nodes which do not belong to the same father node are not in the same alternative set. All the alternative behaviors usually are software input or output actions such as **send** or **receive** date to or from software system. A complete behavior process of a SUT is a depth first search. For example, if A is to be achieved then the C will be treated else the D, if C is be achieved then F and so on. In the TTCN behavior tree a logical loop is allowed but the father node must be the next step of the loop.

The right of the picture shows accordingly the statements of TTCN test suit called behavior line. A node maps to a behavior line. The tree structure is represented by using increasing levels of *indentation* to indicate progression into the tree with respect to time.

Definition 2.3(TTCN Behavior Tree, BT in short): A "TTCN Behavior Tree" is a triple T = (N, Φ, r).

where:

1) N is a finite set of behavior nodes n_1, n_2, … n_m; that is, nodes are not necessarily unique, because the same behavior can happen in more than one context.

2) r is a distinguishable member of N called the "root"

3) Φ: N→N is a function, for any arbitrary node n_0 in N there is a sequence of k nodes n_1, n_2 , … n_k ($0 \le k \le m$), n_1, n_2 , … n_k called sons of node n_0 and called them are brothers or members of the same alternative set.

4) Every node n_i is a triple (ID, Beha, Qual) that consists of a node's ID, an operation on node ni and qualifies for constraining the operations.

Definition 2.4(Behavior of BT): Let T = (N, Φ, r) be a BT, call B(BT) ⊆ Beha × Qual a Behavior Class. Where the Beha={$Beha_1$,$Beha_2$,…,$Beha_{m-1}$,$Beha_m$}, Qual={$Qual_1$,$Qual_2$,…,$Qual_{m-1}$, $Qual_m$}. Specially, if Beha × Qual are coded by core Langue of TTCN then note that B(BT) is a Behavior Instance of BT, note as to BI(BT).

For example, suppose that a Beha={L!N-DATArequest, {L?N-DATAindication, L?OTHERwise}}, Qual={**pass, fail, none**}. A Behavior Instance of BT is {<L!N-DATArequest,**pass**>, L?N-DATAindication, **pass> }** or { <L!N-DATArequest,pass>; <L?OTHERwise,**none>}**

III. CONVERTING LTS TO BT

In this section we give a formal discussion for general approach to converting a LTS to BT, and contribute concrete algorithm.

A. Equivalence between LTS and BT

Theorem 1. Let LTS=(S, I,→, |=) be a Labeled Transition System, the binary relations |= are coded by μ-calculus, then there must exits a one-to-one mapping such that Map(LI) is a Behavior Instance of Behavior Tree BT= (N, Φ, r).

Proof: Construct a One-to-One map in accordance with given LTS.

Create a "r" node with respective to Initial Node "i" of LTS such that r∈BT; For ∀*l*∈ LI(LTS) deduce all the *regular formula* R.

- If *l*=a (a∈*A*) then map L(LTS)⊆ A×S×P = {<a,s,p>| $t \xrightarrow{a} s$ } to {<a, $p>_s$} where map predicates **{True, False}** to behavior verdict **{pass, fail}** individually, **True** corresponding to **pass** and **False** to **fail.**
- If *l*= R_1 **"."** R_2, then map L(LTS)⊆A×S×P= {<l,s,p>| $t \xrightarrow{R_1.R_2} s$ } to {<R_1, $p_1>_{s1}$, {<R_2, $p_2>_{s2}$ }} that is s_2 is subtree of s_1
- If *l*= R_1 **"|"** R_2, then map L(LTS)⊆A×S×P ={<l,s,p>| $t \xrightarrow{R_1|R_2} s$ } to {{<R_1, $p_1>_{s1}$}, {<R_2, $p_2>_{s2}$ }} which the s_1 and the s_1 belong to same alternative set.
- If *l*= R_1 **"*"**R_2, then map L(LTS)⊆A×S×P ={<l,s,p>| $t \xrightarrow{R_1*R_2} s$ } to {<R_1, $p_1>_{s1}$*} which the s_1 performs once or non.
- If *l*= R_1 **"+"** R_2, then map L(LTS)⊆A×S×P= {<l,s,p>| $t \xrightarrow{R_1+R_2} s$ } to {{<R_1, $p_1>_{s1}$}*} which the s_1 performs at least once.

All above steps generate a BT obviously therefore the Map(LI) is a Behavior Instance of BT.

Theorem 2. Let BT= (N, Φ, r) be a Behavior Tree and BI is it's a Behavior Instance, then must exit a one-to-one map such that Map(BI) is a LI(LTS). Where the binary relations |= are coded by μ-calculus.

Proof: Thinking of the one-to-one map constructed in proof of theorem 1 it is obviously the fact of theorem 2.

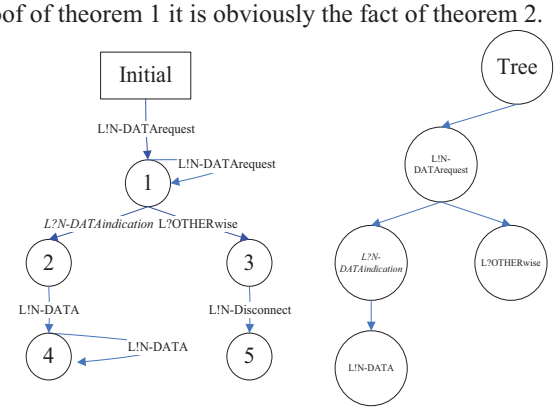

Fig. 2 LTS and Behavior for communication dialog

For example, the left part of the fig. 2 shows a LI(LTS) that specifies a communication process.

LI(LTS)={<Start,Initial,True>,<L!DATArequest,1, True>*,<L?N-DATAindication,2,True>,<L?OTHERwise, 3, False>,<L!DATA,4,True>*,<L!Disconnect,5,False>}, rewrite LI(LTS) with respect to **STATE FORMULAS** of μ-calculus as {< L!DATArequest*.L?N-DATAindication. L!DATA>True,<L!DATArequest*.L?OTHERwise> False}.

A corresponding Behavior Tree is showed in the right part of the fig. 2. the Map(LI)=BI={r, $\{L!DATArequest,psaa\}_1$},{$\{L?N-DATAindication,pass\}_2$,{ $L?OTHERwise,Fail\}_3$},{$L!DATA,pass\}_4$}.

B. A algorithm of converting LI to BI

Algorithm 1 converting LI to BI

Input: LI
Output: BI
Process:
Generate a root node, let r∈BI
Read a element of LI to record R=<l,s,p>
For each R Do
If *l*=a then rewrite <a,s,p> to <a, p>$_s$ replace **True** with **pass** and **False** with **fail**.
If $l=R_1$ *"."* R_2, then rewrite <R_1 .R_2,s,p> to {<R_1, p_1>$_{s1}$, {<R_2, p_2>$_{s2}$ }}
If $l= R_1$ *"|"* R_2, then rewrite <R_1 |R_2,s,p> to {{<R_1, p_1>$_{s1}$}, {<R_2, p_2>$_{s2}$ }}
If $l= R_1$ *"*"* R_2, then rewrite <R_1*R_2,s,p> to {<R_1, p_1>$_{s1}$*}
If $l= R_1$ *"+"* R_2, then rewrite <R_1+R_2,s,p> to {<R_1, p_1>$_{s1}$+}
End for
End process

IV. APPLICATION STUDY

In this section we discuss an approach to integrating the testing with the model checking in verifying the reliability of composite web services. The approach takes advantage of model checking to verify BPEL at the logical level, and to generate test suite automatically based on the model description, and finally to select test cases with respect to the counterexamples of the model checking output.

A. Introduction of CADP

CADP is a toolbox developed by the VASY team at INRIA Rhone-Alpes. Its objective is to specify and verify asynchronous finite-state systems. The EVALUATOR 3.0 of CADP toolbox [11] performs on-the-fly model checking of μ-calculus formulas on LTS. It uses a so-called **exhibitor** to perform an on-the-fly search in the Labelled Transition System (LTS), looking for execution sequences (also called "diagnostic sequences") that start from the initial state and match the specified pattern. **Exhibitor** displays on the standard output the diagnostic sequence(s) found, if any, using the simple SEQUENCE format. The case in which no diagnostic sequence has been found is also covered by the simple SEQUENCE format.

In the CADP toolbox, the SEQUENCE format is the standard format for specifying diagnostic sequences. The following BNF-like grammar defines the syntax of the full SEQUENCE format. The axiom of the grammar is a sequence_list.

- sequence_list ::= "| sequence|sequence '[]' '\n' sequence_list
- sequence::= label_group '\n'label_group '\n' sequence| '<deadlock>' '\n'
- label_group::= label| label '*'| label '+'|'<while>' label| '<until>' label| '<while>' label '<until>' label
- label::= simple_label| label '&' simple_label|label '|' simple_label|label '^' simple_label
- simple_label::= '<any>'|string| regular_expression|'~' simple_label|'(' label ')'

CADP also defines a simple format for diagnostic senquence. The axiom of the grammar is sequence_list.

- sequence_list::="|sequence|sequence'[]"\n' sequence_list
- sequence::=string '\n'|string '\n' sequence|'<deadlock>' '\n'

B. Improvement of the converting algorithm

Algorithm 2 Improvement for converting LI to BI
Input: diagnostic sequence
Output: TTCN Test Suite
Process:
For each *Sequence-list*$_i$ of diagnostic sequence do
Read a *sequence* to records
For each *label-group*$_i$ Do
If *label-group*$_i$ =*label* then add label to {i, label} where I present a root of subtree
If *label-group*$_i$ = *label '*'* then add any more label to {i, label}
If *label-group*$_i$ = *label '+'* then add at least a label to {i, label}
If *label-group*$_i$ = '<*while*>*' label* then add a *while label* to {i, label}
If *label-group*$_i$ = '<*until*>*' label* then add any but unlike *label* to {i, label}
If *label-group*$_i$ = *'<while>' label '<until>'* then add *label* from *while* to *until* to {i, label}
For each *label* of *label-group*$_i$ do
If *label =simple_label*$_1$ *'&' simple_label*$_2$ then add both to {I,*label*,{{*simple_label*$_1$}, {*simple_label*$_2$}}}. Where *label* is root note of subtree *simple_label*$_1$ and *simple_label*$_2$ *is a* son note of *simple_label*$_1$
If *label =simple_label*$_1$ *|" simple_label*$_2$ then add both to {I,*label*,{*simple_label*$_1$}, {*simple_label*$_2$}}. Where both *simple_label*$_1$ and *simple_label*$_2$ are brothers.
If *label =simple_label*$_1$ *'^' simple_label*$_2$ then add both to {I,*label*,{{*simple_label*$_1$}, {*simple_label*$_2$}}}. Same as *'&'*.
End for
End for
End for
End process

C. Case Study

Fig.3 is an example of Web service composite. There are five web service components which provide credit policy and loaning service by $Bank_0$ to $Bank_3$. In order to provide the customer flexible and fast service, the HouseLoanBroker has developed a web service composite with BPEL, which receives queries from the customer and then calls the credit policy service and the loan service to compute lending rate with respect to the different credit policies.

The following sample code is part of BPEL for receiving request from Customers. The sentences 1 to 6 are the definition of message which will be sent or received within the HouseLoanBroker and the Customer. The sentences 10 to 12 are groups of operations that achieve service binding through port Type defined in sentence 9.

….

1.＜**message** name="getLoanQuoteRequest"＞

＜**part** name="Customer ID" type= "typens: getLoanQuote Request" /＞

2.＜/**message**＞

3.＜**message** name="getLoanQuoteResponse"＞

＜**part** name="PayRate" type="typens:getLoanQuoteResponse " /＞

4.＜/**message**＞

5.＜/**message**＞

＜**portType** name="HouseLoanBroker"＞

6.＜**operation** name="getLoanQuote"＞

＜**input** message="tns:getLoanQuoteRequest" /＞

＜**output** message="tns:getLoanQuoteResponse" /＞

＜ **fault** name="UnknownNAME" message= "tns:unknownNAMEFault" /＞

7.＜/**operation**＞

8.＜/**portType**＞

＜**plnk:partnerLinkType** name="HouseLoanBrokerPL"＞

＜ **plnk:role** name="HouseLoanBrokerService" portType= "tns:HouseLoanBroker" /＞

＜/**plnk:partnerLinkType**＞

9.＜ **binding** name="HouseLoanBroker" type= "tns:HouseLoanBroker"＞

＜**operation** name="request"＞＜/**operation**＞

10. ＜/**binding**＞

11. ＜**service** name="HouseLoanBrokerService"＞

＜ **port** name="houseloanbroker" binding= "tns:HouseLoanBroker" /＞

12. ＜/**service**＞

……

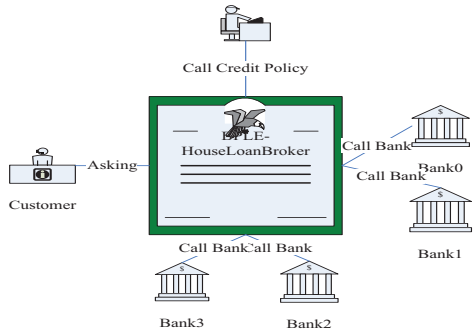

Fig. 3 HouseLoanBroker Service

Similar to the above sample code, the BPEL HouseLoanBroker also has composition with HouseLoanAgency, bank0, bank1, bank2 and bank3 respectively. We only demonstrate sample code of BPEL for calling HouseLoanAgency and bank0 for short.

……

1.＜**message** name="getHouseNumberRequest"＞

＜ part name="Customer ID" type= "typens:getHouseNumberRequest"/＞

2.＜/**message**＞

3.＜**message** name="getHouseNumberResponse"＞

＜ part name="HouseNumber" type= "typens:getHouseNumberResponse" /＞

4.＜/**message**＞

5.＜**portType** name="HouseLoanAgency"＞

＜**operation** name="getHouseNumber"＞

＜**input** message="tns:getHouseNumberRequest" /＞

＜**output** message="tns:getHouseNumberResponse" /＞

＜ **fault** name="UnknownNAME" message="tns:unknownNAMEFault"/＞

＜/**operation**＞

6.＜/**portType**＞

7.＜**plnk**:partnerLinkType name="HouseLoanAgencyPL"＞

＜ plnk:role name="HouseLoanAgencyService" portType="tns:HouseLoanAgency" /＞

8.＜/**plnk**:partnerLinkType＞

……

1.＜**message** name="getLoanQuoteRequest"＞

＜ part name="HouseNumber" type="typens:getLoanQuoteRequest" /＞

2.＜/**message**＞

3.＜**message** name="getLoanQuoteResponse"＞

＜part name="PayRate" type="typens:getLoanQuoteResponse" /＞

4.＜/**message**＞

5.＜**portType** name="Bank"＞

＜**operation** name="getLoanQuote"＞

＜**input** message="tns:getLoanQuoteRequest" /＞

＜**output** message="tns:getLoanQuoteResponse" /＞

＜/**operation**＞

6.＜/**portType**＞

7.＜**plnk**:partnerLinkType name="BankPL"＞

＜**plnk**:role name="BankService" portType="tns:Bank" /＞

8.＜/**plnk**:partnerLinkType＞

……

For verifying the logical correctness of BPEL we developed a tool to model BPEL with LOTOS [21]. It translates BPEL into LOTOS for model checking with EVALUATOR 3.0 in which a LTS is as output. A part of translated LOTOS model and its LTS are shown as following.

```
Client [HouseLoanBroker]
    |[HouseLoanBroker]|
HouseLoanBroker [HouseLoanBroker, HouseLoanAgency, Bank0, Bank1, Bank2, Bank3]
    |[HouseLoanAgency, Bank0, Bank1, Bank2, Bank3]|
(HouseLoanAgency [HouseLoanAgency] ||| Bank0 [Bank0] ||| Bank1 [Bank1] ||| Bank2 [Bank2] ||| Bank3 [Bank3]
where
      process Client [HouseLoanBroker] : noexit :=
        HouseLoanBroker !0; Client [HouseLoanBroker]
      [] HouseLoanBroker !1; Client [HouseLoanBroker]
      [] HouseLoanBroker !2; Client [HouseLoanBroker]
      [] HouseLoanBroker !3; Client [HouseLoanBroker]
      [] HouseLoanBroker !4; Client [HouseLoanBroker]
      [] HouseLoanBroker !5; Client [HouseLoanBroker]
      [] HouseLoanBroker !6; Client [HouseLoanBroker]
      [] HouseLoanBroker !7; Client [HouseLoanBroker]
      [] HouseLoanBroker !8; Client [HouseLoanBroker]
Endproc
```

Table 1 and Fig.4 and presents a normal state space of LTS where all Web services provide their service through their interface. However, all Web services are independent

on the BPEL HouseLoanBroker. There is no reason for web services to tell its caller when its interface change for some reasons. In order to validate the reliability of Web services the BPEL designer has to make a model checking at its logical level firstly. Meanwhile some counterexamples will be output because the change of interface. It is time-consuming to perform a rechecking after fixing all defaults. An economic strategy is that the designer of BPEL carries out a test at the point of defaults. It is more effective to generate testing suite based on the LTS of model checking and to select test cases from the counterexamples.

Table 1 LTS of HouseLoanBroker

States	transitions	labels	deadlock states
65	103	39	10 12…(26 in total)

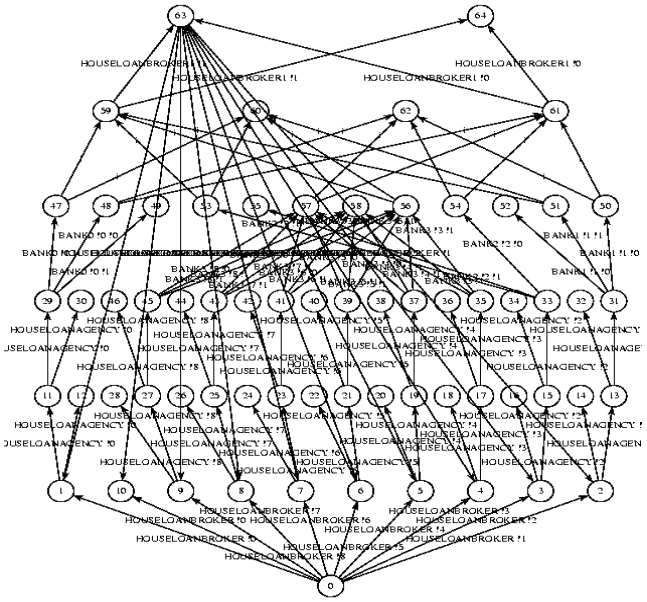

Fig. 4 LTS of HouseLoanBroker

In order to test the reliability of web service involved in BPEL we developed a TTCN test tool based on algorithm 1 and 2. It can help the designer of BPEL to translate a LTS into TTCN behavior tress automatically, and facilitate test case selection. We focus on three kind of property of BPEL such as the RELIABILITY, LIVENESS and FAIRNESS [11].

SAFETY PROPERTIES

Informally, a safety property expresses that "something bad never happens." Typical safety properties are those forbidding "bad" execution sequences in the LTS. These properties can be naturally expressed as:

[true*] < true > true

LIVENESS PROPERTIES

Informally, a liveness property expresses that "something good eventually happens." Typical liveness properties are *potentiality* assertions (i.e., expressing the reachability on a sequence) For instance:

<true* . 'BANK0 !.* !.*'> true

<true* . 'BANK1 !.* !.*'> true

<true* . 'BANK2 !.* !.*'> true

<true* . 'BANK3 !.* !.*'> true

FAIRNESS PROPERTIES

These are similar to Liveness properties, except that they express reachability of actions by considering only *fair* execution sequences. A sequence is fair if it does not infinitely often enable the reachability of a certain state without infinitely often reaching it. For instance:

['HOUSELOANBROKER !0'] <true . true . 'BANK0 !.* !.*'> true

['HOUSELOANBROKER !1'] <true . true . 'BANK1 !.* !.*'> true

['HOUSELOANBROKER !2'] <true . true . 'BANK2 !.* !.*'> true

['HOUSELOANBROKER !3'] <true . true . 'BANK3 !.* !.*'> true

We execute model checking with the above three property and get three counterexamples.

Counterexamples 1 for SAFETY PROPERTIES:

In according with the business rule, if HouseLoanAgency receives ID 1 it responses to HOUSELOANBROKER the house number 3. However, because the service HouseLoanAgency does not work well, it returns a number 0.Tthe output sequence of counterexample 1 is follow:

*"HOUSELOANBROKER !1"."**HOUSEAGENCYSERVICE !1 !0**"."BANK0 !0 !2" "HOUSELOANBROKER!2"*

Counterexample 2 for LIVENESS PROPERTIES:

*The **bank1** have an evolution on its interface, it adds HouseNumber into new one, so a counterexample sequence will be output as follow:*

*HOUSELOANBROKER!2"."HOUSEAGENCYSERVICE !2 !1" "**BANK1 !1 !3**" "**Deadlock**"*

Counterexample 3 for FAIRNESS PROPERTIES:

We inject a default into LTS model with μ-calculus formula (HOUSELOANBROKER !2)* and get a counterexample3 as follow:

"HOUSELOANBROKER!1"."HOUSELOANAGENCYSERVICE!"."BANK0!*!*""HOUSELOANBROKER !*"*

Meanwhile, the character '*' represents an integer, when it is an ID of customer the '*' represents a duration from 0 to 3. Actually the counterexample is normal process because we let an abnormal μ-calculus formula as property of LTS.

Based on the above counterexamples a group of test cases are selected in according with its output sequence.

For SAFETY PROPERTIES:

$\{HouseLoanBroker!CustomerID,pass\}_{HouseLoanBroker}$,$\{HouseLoanAgency!CustomerID,pass\}_{HouseAgencyrService}$,$\{Bank0!HouseNumber,pass\}_{Bank0}$

For LIVENESS PROPERTIES:

$\{HouseLoanBroker!CustomerID,pass\}_{HouseLoanBroker}$,$\{HouseLoanAgency!CustomerID,pass\}_{HouseAgencyrService}$,$\{Bank*!HouseNumber,pass\}_{Bank*}$// '*' represent a integer duration from 0 to 3

For FAIRNESS PROPERTIES:

$\{HouseLoanBroker!CustomerID,pass\}_{HouseLoanBroker}$,$\{HouseLoanAgency!CustomerID,pass\}_{HouseAgencyrService}$,$\{Bank*!$

HouseNumber,pass}$_{Bank}$*// '*' represent a integer duration from 1 to 3.

Fig. 5 is a form of performing TTCN-3 test for BPLE: HouseLoanBroker. In the middle of fig. is a select area where have tree groups of test case.

Table 1 is a performance result. The checking consumes much more time than the testing does.

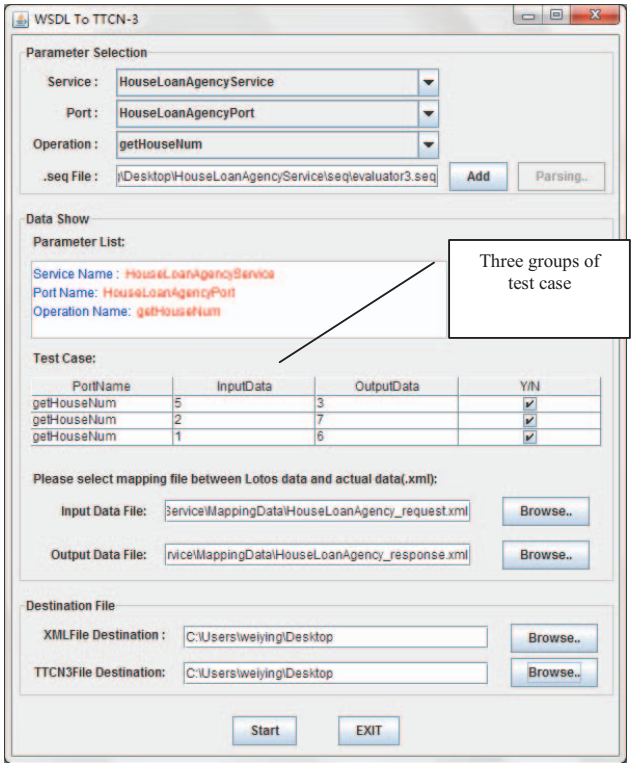

Fig. 5 Test for BPLE: HouseLoanBroker

Table 1 A time analysis of model checking and testing

Steps	Actions	Timer
Checking	LOTOS Model	2s
	Simulating	48s
	Checking	59s
Testing	TTCN-3 Test suit	5s
	Testing	10s

V. Related Work

Web service testing and verification already draw much attention in the recent years. A lot of papers have been published. Here we focus on summarizing some papers that are closely related with our work.

Thierry J' and Pierre Morel [12] argue that model-checking and testing are different activities, at least conceptually. Nevertheless, there are also similarities in models and algorithms. They proposed a new on-the-fly test generation algorithm with respect to the classical graph algorithm for example LTS.

Paul E. Black and William Majurski [13] apply a model checker to help test generation in a new application of mutation analysis. They use the concept of syntactic operators to describe slight variation on a given model. The operators define a form of mutation analysis at the level of the model checker specification. A model checker generates counterexamples which distinguish the variations from the original specification. The counterexamples can easily be turned into complete test cases. Authors appraise the substantial advantages to combine a model checker with test generation.

Angelo Gargantini and Constance Heitmeyer[14] defined a concept of trap properties. The idea is neither to use model checking neither for verification nor to detect specification errors but to construct test sequences. Nevertheless, the authors base the method on two ideas. Firstly, the model checker is used as an oracle to compute the expected outputs. Secondly, the model checker's ability to generate counterexamples is used to construct the test sequences to force the model checker to construct the desired test sequences.

All the work above demonstrates how to generate test sequence from model checking. There is no effort to generate and carry test cases. Both the logical verification and the practical test still need to be connected.

Yongyan Z., Jiong Z. and Paul K.[15] head on the challenge of time-consuming and error prone in test case generation manually where testing BPEL orchestration provides a model checking based test case generation framework for BPEL. This framework employs a Web Service Automata (WSA in short) to describe the operational semantics for BPEL. Using LTL and CTL temporal logic models test coverage criteria that associate to SPIN and NuSMV test case generator. State coverage and transition coverage are created for BPEL control flow testing, and all-du-path coverage is used for BPEL data flow testing. Two levels of test cases can be generated to test whether the implementation of web services conforms to the BPEL behavior and WSDL interface models. The generated test cases are executed on the JUnit test execution engine.

Quite similar with our work, A, Ferrara's [16] defines a two-way mapping between BPEL and LOTOS and then uses LOTOS to reconstruct the business process. He carries on model checking with the toolbox CADP, where employ LOTOS as model language, to verify the business reachability of BPEL. The main difference from our work is that the authors only tell reader how to generate test cases from model checking in an engineering way. There is no proof for its rightness, and no automatic connection to test environment.

Based on the SPIN, a model checking toolbox, [17] established an automatic test framework for web services composition of BPEL. To facilitate BPEL verification a translation method from BPEL into Promela, a language for describing the properties under checking in SPIN, is given. Applying this framework discusses how to describe properties for test case generation for BPEL. This is a really static test method. It only pays attention to checking the service reachability at logical level without any discussion for on line test. Not only this, the authors misunderstand the conception between software verification and software testing.

Mounir Lallali, Fatiha Zaidi and Ana Cavalli[18] intensively observe the drawbacks in web services testing and state that the majority problem is requirement for a intermediate format between BPEL and a formal language, in which the test case can be generated automatically. A transformation procedure of the BPEL specification into an Intermediate Format (IF) model that is based on timed automata is proposed in this paper. This IF format is well adapted to model BPEL (timed) constructs and to handle faults, events, termination, message correlation and activities synchronization. The proposed transformation was implemented in the BPEL2IF tool.

[19] shows some negative influence on testing activities where web services perform asynchronous behavior, distribute availability and the lack of user interface. Bearing in their mind of those challenges, a model based testing method is proposed. It still employs SPIN as a model checking tool, therefore a transformation of the composition specified in BPEL into a Promela has been discussed.

Ana R. at el. [20] present a methodology and a set of tools for the modeling, validation and testing of Web service composition. This methodology includes several modeling techniques, based mainly on some variations of Timed Extended Finite State Machines (TEFSM) formalism, which provides a formal model of the BPEL description of Web services composition. These models are used as a reference for the application of different test generation and passive testing techniques for conformance and robustness checking. This paper mainly focuses on the application of various testing methods on web services, rather than the method of test case generation.

Mustafa Bozkurt, Mark Harman and Youssef Hassoun [3] argue that testing web services is more challenging than testing traditional software due to the complexity of web service technologies and the limitations that are caused by the SOA environment. The complexity of web services due to their standards not only affects the testing but also slows down the transition to web services. Limited control and ability to observe render most of the existing software testing approaches inapplicable. There are other issues in web service testing, such as:

- The frequency of testing required,
- Testing without disrupting the operation of the service,
- Determining when testing is required and which operations need to be tested.

Franck van Breugel1 and Maria Koshkina [5] summarize the recent work on modeling and Web Service verification techniques, and classify it into five different types: 1) approaches based on Petri Net; 2) approaches based on SPIN; 3) approaches based on process algebra; 4) approaches based on abstract state machine, and 5) approach based on automata. All the methods take on some technical strategies which map each BPEL process to a formal model, such as Petri Nets, SPIN, process algebra, abstract state machine and automata. Not only this approach provides a model, but also allows the verification techniques and tools developed to be exploited in the context of BPEL processes.

Web service testing and verification is a new research area with the new concept of web service. It claims a new approach or method to guarantee the quality of web services. However, from our point of view, existing work has not solved the new verification challenges incurred by the new features of SOA applications such as dynamicity, basically due to the lack of automation in testing and test suit generation. New methods and tools are still in need urgently.

VI. CONCLUSIONS AND FUTURE WORK

The proposed approach is promising. The model of BPEL for interesting properties can be constructed with the proposed algorithm. The interesting properties can be verified by existing model checking tools, for example CADP. The traces of counterexamples from model checking can be used as a TTCN-3 test suite which can be automatically generated by the proposed translating algorithm.

The state spaces explosion of model checking is restrained because we only selecting the traces of counterexamples as clued to generate test case. The hard work of the creation of test traces for web service can be automated. We not only demonstrate how to generate test sequence from model checking but also show how to execute test case automatically. Both the logical verification and the practical tests are connected smoothly.

Our approach is not a blue-sky idea; we accomplished our work step by step from theoretical research to practice. We proved the equivalence between LTS and BT for generating test case from model checking counterexamples. There is no existing work let us follow.

In the future, we will investigate and develop an approach to improving the reliability during web service evolution, find the possible methods of how to generate traces based on the clue where cooperation protocols are changed.

ACKNOWLEDGEMENT

The work in this paper has been supported by Natural Science Foundation of China (Grant No: 61070030, 61111130121) and the British Royal Society of Edinburgh (RSE-Napier E4161). It is also partly supported by Beijing Government and Education Committee (Grant No. PHR201107107).

REFERENCES

[1] E. Christensen, F. Curbera, G. Meredith, and S.Weerawarana. *Web services description language (WSDL) 1.1*. http://www.w3.org/TR/wsdl, 2001.

[2] IBM Corporation. *Business Process Execution Language for Web Services BPEL-4WS* (Version 1.1), 2002. http://www.ibm.com/developerworks/library/ws-bpel.

[3] Mustafa Bozkurt, Mark Harman and Youssef Hassoun. Testing Web Services: *A Survey, Technical report TR-10-01*. www.dcs.kcl.ac.uk/technical-reports/papers/TR-10-01.pdf. 2010.1.

[4] Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled, *Model Checking*: MIT Press, 1999, ISBN 0-262-03270-8

[5] Franck van Breugel1 and Maria Koshkina. *Models and Verification of BPEL*. http://www.cse.yorku.ca/~franck /research/drafts/tutorial.pdf

[6]Joost-Pieter Katoen. Labelled *Transition Systems Model-Based Testing of Reactive Systems* Lecture Notes in Computer Science, 2005, Volume 3472/2005, 615-616.

[7]R.J. van Glabbeek. *Bisimulation*, http://www.cse. unsw.edu.au/~rvg/pub/Bisimulation.pdf.

[8] Radu Mateescu. Local Model-Checking of Modal Mu-Calculus on Acyclic Labeled Transition Systems, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002* (Grenoble, France), April 2002

[9] Colvin, R., Grunske L. and Winter, K., Probabilistic Timed Behaviour Trees", in Integrated Formal Methods*, 6th Intl. Conference, IFM 2007*, Springer, LNCS 4591, July 2007, pp157-175.

[10] ETSI ES 201 873-1 V2.2.1 - TTCN-3 Home page. *www.**ttcn**-3.org/doc/es_ 20187301v020201p_chinese.pdf*

[11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe, CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. *TACAS'2011* (Saarbrücken, Germany), March 2011.

[12] Thierry J' eron and Pierre Morel. Test Generation Derived from Model-Checking. *Lecture Notes in Computer Science 1633,* Springer 1999, *CAV 1999*, pp 108-121, Trento, Italy, July 1999.

[13]Paul Ammann, Paul E. Black, William Majurski. Using Model Checking to Generate Tests from Specifications. *ICFEM'98*, December,1998, Brisbane, Queensland, Australia, pp 46-56.

[14] Angelo Gargantini, Constance L. Heitmeyer: Using Model Checking to Generate Tests from Requirements Specifications. ESEC / SIGSOFT FSE 1999: 146-162. *7th European Software Engineering Conference*, Toulouse, France, September 1999, Lecture Notes in Computer Science 1687 Springer 1999.

[15] Yongyan Zheng, Jiong Zhou, Paul Krause: An Automatic Test Case Generation Framework for Web Services. *JSW 2007* Vol.2(3): 64-77.

[16] A. Ferrara, Web services: a process algebra approach, *In Proc. of ICSOC*. ACM Press, 2004, p. 242-343.

[17] Rongsheng Dong, Zhao Wei, Xiangyu Luo, Fang Liu. Testing Conformance of BPEL Business Process Based on Model Checking, *JOURNAL OF SOFTWARE*, 5(9), SEPTEMBER 2010, 1030-1037.

[18] Mounir Lallali, Fatiha Zaidi, Ana Cavalli. Transforming BPEL into Intermediate Format Language for Web Services Composition Testing. *NWESP '08*. Oct. 2008 Seoul. pp 191-197.

[19] José García-Fanjul, Claudio de la Riva, Javier Tuya: Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking. *TAIC PART 2006*: 127-130.

[20] Ana R. Cavalli, Tien-Dung Cao, Wissam Mallouli, Eliane Martins, Andrey Sadovykh, Sébastien Salva, Fatiha Zaïdi: WebMov: A Dedicated Framework for the Modeling and Testing of Web Services Composition. *ICWS 2010*: 377-384.

[21] T. Bolognesi, J. v.d. Lagemaat, C. A. Vissers (editors) *LOTO Sphere: Software development with LOTOS,* Kluwer Academic Publishers, 1994.