

Translation-based Verification of Web Services Composition via ZING

Xiangyu Luo^{1,2}, Jingjing Lu¹, Kaile Su^{3,4}, Rongsheng Dong¹

¹School of Computer and Control, Guilin University of Electronic and Technology, Guilin, China

²School of Software, Tsinghua University, Beijing, China

³Key laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China

⁴Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia

E-mails: shiangyuluo@gmail.com, ranzeno1985@163.com, sukl@pku.edu.cn, ccrsdong@263.net

Abstract—BPEL4WS (BPEL for short) is a standard business process execution language for Web services composition. To formally verify the correctness and reliability of Web services compositions, we propose an Input/Output Labelled Transition System (I/OLTS) as the intermediate formal model, which is well adapted to model BPEL constructs and handle faults, events, terminations, message correlation and activities. To be able to automatically verify Web services compositions via a model checker, we first develop a translation procedure to translate BPEL language into I/OLTS, and then develop another translation procedure to translate the I/OLTS model into the input language of ZING, a software model checker developed by Microsoft. The translation-based verification process for Web services composition is illustrated by a case study.

Keywords - Web services; BPEL4WS; I/OLTS; software model checking; ZING

I. INTRODUCTION

Web services are now considered as one of the key paradigms for application integrations. It allows businesses to be able to rapidly adapt to changes in the business environment. BPEL4WS is a standard business process execution language for Web services composition. When testing the correctness and reliability of BPEL programs, the developers have to consider every possible interleaving of events among various processes [3], while such testing process is hard to be done without formal verification methods. A formal verification technique called “model checking” has been proven to be surprisingly effective in the design and testing of concurrent systems. The basic idea of model checking is the following: Firstly, it manages to represent a “model” from a system, where a model abstractly represents only a small amount of information (states) about the system and depict how these information changes in the system, and then it is feasible to systematically explore the execution paths of the model to determine whether the model satisfies the properties we want to check. Such properties are usually expressed in some temporal logics. Basically, the model checking techniques for Web services behavior has remained limited to checking safety and liveness properties [4].

Software model checking is an active area within the model checking community. The goal of software model checking is to expand the application areas of automated

verification. It can be used to verify the correctness of some algorithms during the execution of program. ZING [1,2] is an explicit state software model checker developed by Microsoft, in the spirit of SPIN [5], JPF and BOGOR [13]. In comparison with SPIN, ZING supports several features like objects and function calls, which make it more suitable for automatic extraction of models from programming languages. In comparison with JPF and SLAM [14], ZING implemented newer algorithms for state-space reduction [15], such as reduction and summarization [2].

Web services are of course some software components that communicate with each other via Web. Therefore, in order to automatically verify the correctness and reliability of Web services composition by software model checking techniques, in this paper we propose to apply software model checking techniques to the analysis on Web service flow descriptions. We focus on the BPEL (Business Process Execution Language) as a representative language to describe the Web service flows, and choose the ZING model checker as the verification engine.

The paper is organized as follows: Section II introduces some essential preliminary knowledge of this paper. Section III proposes a procedure to extract and translate the behavioral specification of BPEL application program into I/OLTS model. After that, the I/OLTS model is translated into a ZING model such that we are able to automatically analyze Web services composition by using the ZING model checker. We present a case study by using our verification method in Section IV and conclude this paper in Section V.

II. PRELIMINARIES

A. BPEL Behavior

BPEL is a standard business process execution language for Web services composition [8]. It can be used to describe both the external behavior of a service and its internal implementation. On the other hand, BPEL is a language for expressing behavioral compositions of Web service providers. Each Web service provider can be seen as a Port instance of a particular Port Type.

For the reasons above, we adopt BPEL as the behavior descriptions of Web services and their composition. In this paper we focus on the translations for some main and basic activities of BPEL.

B. Labelled Transition System

A Labelled Transition System (LTS) usually describes the discrete behaviour of some system or protocol (Web services is one kind of such systems): in any state of the system, a number of actions can be performed, each of which leads to a new state. The initial state corresponds to the state in which the system resides before any action has been performed.

Let Act be a finite set of actions ranged over a, b, c, \dots . Action τ denotes the distinguished invisible (or, unobservable) action, i.e., $\tau \notin Act$. Labelled transition system is formally defined as follows.

Definition 1 (Labelled Transition System, LTS) A labelled transition system M_{LTS} is a tuple (Q, q_0, L, \rightarrow) where

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- $L \subseteq Act$ is a set of observable actions called the alphabet of M_{LTS} ;
- $\rightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ is a transition relation.

To formally verify a composite Web service, the design of a formal model is necessary, because it facilitates the application of model checking techniques. In the next section we will extend LTS to a new formal model called I/OLTS, by which we are able to model Web services composition conveniently.

C. The Software Model Checker ZING

The goal of the ZING project is to build a flexible and scalable model checking infrastructure for concurrent software [1]. It checks properties of concurrent heap-manipulating programs using model checking. It believes that such an infrastructure is effective for finding bugs in software at various levels: high-level protocol descriptions, work-flow specifications, Web services, device drivers, and protocol in the core of the operating system [2]. Therefore, the work in the rest of this paper can be viewed as an effort to extend the ZING model checker to the verification of Web services.

III. TRANSLATION FROM BPEL TO ZING

A. Modeling with I/OLTS

BPEL has language constructs to express data flows and control flows. It is related to the input and output actions. Some of the activities send message via channel, and the incoming messages are stored in some variables. By an analysis of BPEL language constructs, in this paper we propose an extended LTS model, called I/OLTS, to model the behavior of BPEL. I/OLTS is formally defined as follows.

Definition 2 (Input/Output Labelled Transition System, I/OLTS) An Input/Output Labelled Transition System M is a tuple $(Q, q_0, L, V, \rightarrow)$ where

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- L is the Alphabet including symbols in the following forms:
 - $P!X$: output action designator;
 - $P?X$: input action designator;
 - τ : internal action designator;
- V is a finite set of variables;
- $\rightarrow \subseteq Q \times A \times Q$ is a transition relation where A is the set of transition actions $L \times G \times \theta$ with
 - G : guard conditions;
 - θ : variable update functions.

A designator of $P?X$ is used for receiving a message with the value of X coming from channel P . A designator of $P!X$ is used for sending a message X to channel P .

From Definition 2, a tuple $\langle q, (a, g, v := c), q' \rangle$ corresponds to a transition from state q to state q' labeled with a , a guard condition g that specifies when the transition is enabled, and a variable update function $v = c$ that specifies the value of variable v will become c in state q' .

B. BPEL to I/OLTS

In this subsection, by an analysis on the execution semantics of some main and basic activities in BPEL, we give the rules for translating these activities into I/OLTS.

1) Atomic activities

- Receive:

```
<receive partnerLink=L portType=T
operation=Op variable=V>
```

The <receive> activity allows a business process to wait for the arrival of a matching message through the given channel. We define the I/OLTS transition relation for receive activity as Figure 1.

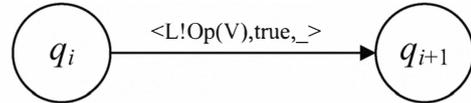


Figure 1. The translation from receive activity to I/OLTS

In Figure 1, the channel is specified by partner-Link L , and the message is constructed from the operation and variable as $Op(V)$. The transition is enabled when a message $Op(V)$ is received through channel L .

- Reply:

```
<reply partnerLink=L portType=T
Operation=Op variable=V>
```

The <reply> activity allows a business process to send a message for responding to the message that was received by inbound message activity. We define the I/OLTS transition relation for reply activity as Figure 2, in which the transition is enabled when a message Op(V) is sent through channel L.

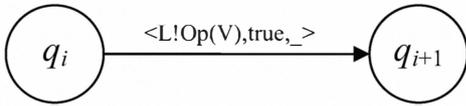


Figure 2. The translation from reply activity to I/OLTS

- Invoke:
<invoke partnerLink=L portType=T
Operation=Op inputVariable=IV
outputVariable=OV>

The <invoke> activity allows a business process to invoke a one-way or request-response operation on a portType offered by a partner. We define the I/OLTS transition relation for invoke activity as Figure 3.

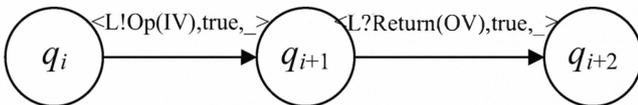


Figure 3. The translation from invoke activity to I/OLTS

- Assign:
<assign>
<copy>
<from variable=X/>
<to variable=Y/>
</copy>
</assign>

The <assign> activity is used to update the values of variables with new data. We define the I/OLTS transition relation for assign activity as Figure 4.

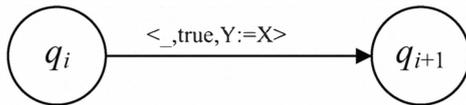


Figure 4. The translation from assign activity to I/OLTS

2) Control activities

- If
<if>
<if condition=C1>...</if>
...
<else if condition=Ck>...</else if>
<else>...</else>
</if>

The <if> activity is used to select exactly one activity for execution from a set of choices. We define the I/OLTS transition relation for if activity as Figure 5. In Figure 5, If condition C1 is satisfied in state q_i , the system will transform from state q_i to state q_{i+1} . If condition Cj

($2 \leq j \leq k$) is satisfied in state q_i , the system will transform from state q_i to state q_{i+j} . Otherwise, the system will transform from state q_i to state q_{i+k+1} . In Figure 5, the symbols “!” and “&” represent the logic NOT and AND operators respectively.

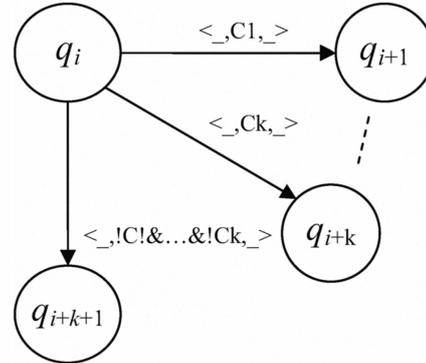


Figure 5. The translation from if activity to I/OLTS

- While
<while condition=C>
...
</while>

The <while> activity is used to define that the sub-activities in the body of the <while> activity are to be executed repeatedly as long as the specified <condition> is true. We define the I/OLTS transition relation for while activity as Figure 6, where states q_{i+1} and q_{i+k} are defined for the starting and the ending sub-activities of the body of the <while> activity respectively. Figure 6 means that if C is true, then the system will transform from state q_i to state q_{i+1} , the starting state of the <while> body. Once the system transformed to state q_{i+k} , the ending state of the <while> body, the system will unconditionally transform from it to state q_i , then a test for condition C will be done again. Such test will be repeated until condition C become false in state q_i , in this case the system will transform to state q_{i+k+1} , the state for the activity following the <while> activity.

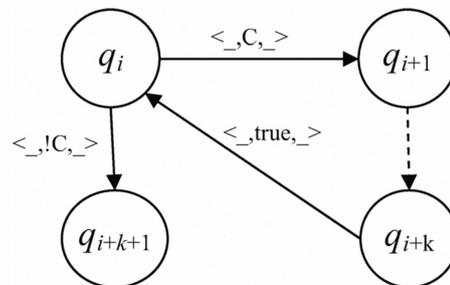


Figure 6. The translation from while activity to I/OLTS

- Flow
- ```

<flow>
 <--sub-activity-1-->
 ...
 <--sub-activity-n-->
</flow>

```

The `<flow>` activity is used to specify one or more activities to be performed concurrently. We define the I/OLTS transition relation for flow activity as Figure 7 and Figure 8.

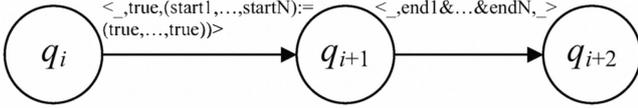


Figure 7. A main I/OLTS fragment for flow activity

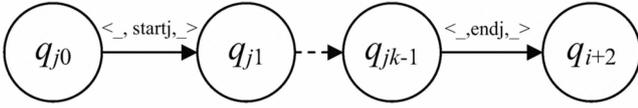


Figure 8. A fragment of sub I/OLTS  $M_j$

For each sub I/OLTS  $M_j (1 \leq j \leq n)$  we define two Boolean variables  $start_j$  and  $end_j$  to express the starting and ending conditions. When  $start_j$  is true, the system will execute to the flow. Once a sub I/OLTS  $M_j$  terminate, the variable  $end_j$  will be set to true. For the main fragment, when all  $start_j$  become true, the state will transfer to the next state  $q_{i+1}$ , which means the start of the flow. When all  $end_j$  become true, the state will transfer to the termination state  $q_{i+1}$  of the flow.

### C. I/OLTS to ZING

This section describes the method of using the ZING model checker for the representation and analysis of I/OLTS model. The basic idea is to translate an I/OLTS into the specification language of ZING [1].

The transformation I/OLTS into ZING as follows:

- An I/OLTS  $M$  is translated into a process in ZING.
- The variable  $V$  in I/OLTS becomes a variable in ZING. Except for string type, primitive types map to the Zing language one to one. String type is not available in the Zing language so the literals are represented as arrays of integers.
- The variable update function  $\theta$  assigns a new value to a variable.
- The guard condition  $G$  in I/OLTS are translated into a specifically condition in ZING.
- The communication  $P$  in the input or output action designator is denoted as a ZING channel.

--The ZING channel declaration takes into account the type of message exchanged.

Channel-declaration denoted as follow:

chan identifier type:

- Translating the transition relation  $\rightarrow$  is the most important part of the translation. The basic transformation as follows:

#### a) Communication Activities

- Receive activity is modeled as a single transition with a message label at the transition source state represented as:  
Select {receive (L,Op (V)) ->;};
- Reply and asynchronous invocation activities can be modeled as a single transition with a message label at the transition source state represented as:  
Send (L, Op (V));
- Synchronous invoke activity is modeled as two transitions, while the first is a sending transition annotated with the message name, the second one is a receiving transition annotated with true. The resulting ZING statements are represented as:

```

{
 Send (L,Op (IV));
 Select {receive (L, Return (OV))->;}
};

```

#### b) Structural Activities

- While activity is denoted as follow:

While-statement:

While(C) embedded-statement;

When C yields true, the ZING program is ready for executing the embedded statement. After the ZING program executes the end point of the embedded statement, the control is transferred to the beginning of the while statement. Therefore, the while statement conditionally executes the embedded statement zero or more times.

- If activity is denoted as follow:

```

{
 If (C1) embedded-statement;
 ...
 If (Ck) embedded-statement;
 Else embedded-statement;
}

```

The conditions  $C1 \dots Ck$  are  $k$  Boolean expressions. The if statement selects an embedded statement for execution when the corresponding condition is the first satisfied condition from  $C1$  to  $Ck$ .

- Flow activity is denoted as follow:

Choose (start-expression);

{Foreach (true in start-expression) embedded-statement;}

If (end-expression) embedded-statement;

enum start-expression= {start 1,..., start n};

End-expression= {end 1  $\wedge$  end2  $\wedge$  ...  $\wedge$  end n};

Zing supports a nondeterministic construct called choose. The choose operator is used to make a non-deterministic choice from a set of alternative values. When applied to an enumeration type, a range type, or set variable, a selection is

made dynamically from the contents of the variable at runtime. The value of choose operator is that of the selected value, and the ZING model-checker will generate successor states corresponding to each of the possible values. The foreach statement enumerates the elements of a set or array, executes an embedded statement for each element of the collection.

#### IV. EXPERIMENT

This section presents an example of a WS-BPEL process for handing a purchase order [9]. When receiving the purchase order from a customer, the process initiates three paths concurrently: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed concurrently, there are control and data dependencies between the three paths. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three concurrent paths are completed, invoice processing can proceed and the invoice is sent to the customer.

The ZING model checker will verify that the data is conveyed to the receivers without corruption, duplication, and of message loss. The ZING compiler should be invoked from the command line. "zc" is the name of the zing compiler. To compile the example of purchase order, we can use the command "zc purchaseorder.zing". TABLE I summarizes the experiment of testing for deadlocks. Processes are the concurrent units in ZING. Processes may be created statically in the initial state of a model, or be created dynamically as the execution of a model proceeds. Once a process is created, it concurrently executes with all other executable processes in the ZING model.

TABLE I. EXPERIMENTAL SUMMARY TO TEST FOR DEADLOCKS

| Name          | Purchase order                   |
|---------------|----------------------------------|
| BPEL Features | receive,invoke,flow,reply,assign |
| States        | 249                              |
| processes     | 5                                |

#### V. CONCLUSION

This paper proposes a translation procedure of BPEL specification into ZING model by using I/OLTS as the intermediate formal model. We actually use this translation procedure to test the BPEL specification of the purchase order, an example of Web services composition. It is a successful case study for the analysis of the BPEL from Web Service Version 2.0 documents [9]. Finally, as for our future work, we attempt to apply the translation-based verification

method to more examples and checking more properties related to the correctness and reliability of Web services composition.

#### ACKNOWLEDGMENT

We would like to thank the anonymous referees for their valuable comments. This work is supported by the Chinese National 973 Plan (2010CB328103), the National Natural Science Foundation of China (60725207 and 60763004), the China Postdoctoral Science Foundation (20090450389), the Young Science Foundation of Guangxi Province of China (GuiKeQing0728090), and the ARC Future Fellowship (FT0991785).

#### REFERENCES

- [1] Zing Language Specification-<http://research.microsoft.com/zing>.
- [2] T.Ball andrews, S.Qadeer, S. K.Rajamani, J. Rehof, and Y.Xie. Zing: A model checking for concurrent software. Technical report, Microsoft Research ,2004
- [3] S.Nakajima.On Verifying Web Service Flows,In proc.SAINT 2002 Workshop,pp.223-224,2002
- [4] M.lallali, F.Zaidi,A.Cavalli. Transforming BPEL into Intermediate Format Language for Web Services Composition Testing,IEEE,page 191-197,2008
- [5] G.Holzmann.The model checker SPIN.IEEE Transactions on Software Engineering,23(5):279-295,May 1997.
- [6] S.Nakajima.Model Checking of Safety and Security Aspects inWeb Service Flow, In Proc.WLFM'05,July 2005.
- [7] X.Fu,T.Bultan,and J.Su.Model checking interactions of composite web services.UCSB Computer Science Department Technical Report (2004-05)
- [8] M.Koshkina and F.van Breugel,Verification of Business Processes for Web Service,Technical Report CS-2003-11,York Univ,Oct.2003.
- [9] OASIS Web Service Business Process Execution Language(WSBPEL)TC.Web Service Business Pricess Execution Language Version 2.0,2007
- [10] J.G. Fanjul ,J.Tuya,and C.de la Rina.Generating Test Cases Specifications for Compositions of Web Service using SPIN.In Proceedings of WS-MaTe'06 Workshop ,pages 83-94,2006
- [11] A. Wombacher, P.Frankhauser, and A.terHofserde,"Transforming BPEL into annotated Deterministic Finite State Automata for Service Discovery" , In proc ICWS '04 , july, 2004
- [12] G.Salaun, L.Bordeaux, and McIlraith, "Describing and Reasoning on Web Services using Process Algebra," In Proc.ICWS'04, july, 2004
- [13] Robby, M.Dwyer, and J.Hatchliff. Bogor: An extensible and highly-modular model checking framework. In FSE 03: Foundations of software Engineering, page 26 -276.ACM, 2003
- [14] T.Ball and S.K.Rajamani. The SLAM project: Debugging system software via static analysis. In POPL 02: Principles of Programming Languages, Page 1-3. ACM, January 2002.
- [15] C.Fournet , C.A.R.Hoare, S.K..Rajamani, and J.Rehof. Stuck-free conformance. In CAV 04: Computer-Aided Verification, LNCS. Springer-Verlag,2000