WUJNS
Wuhan University Journal of Natural Sciences

Vol.11 No.5 2006 1311-1319

Article ID:1007-1202(2006)05-1311-09

A Formal Model for BPEL4WS Description of Web Service Composition

□ GU Xiwu, LU Zhengding†
College of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, Hubei, China

Abstract: Communicating Sequential Processes (CSP) is a kind of process algebra, which is suitable for modeling and verifying Web service composition. This paper describes how to model Web service composition with CSP. A set of rules for translating composition constructor of Business Process Execution Language for Web Services (BPEL4WS) to CSP notations is defined. According to the rules that have been defined, the corresponding translation algorithm is designed and illustrated with examples. The methods for model checking, model verification and model simulation are also introduced.

Key words: communicating sequential processes; Web service; Web service composition; business process execution language for Web services
CLC number: TP 311

Received date: 2006-03-15
Foundation item: Supported by the National Natural Science Foundation of China(60403027), the Natural Science Foundation of Hubei Province(2005ABA258), and the Open Foundation of State Key Laboratory of Software Engineering (SKLSE05-07)
Biography: GU Xiwu(1967-), male, Ph.D. candidate, research direction: Web service, semantic Web and middleware. E-mail: guxw_wang@sina.com.
† To whom correspondence should be addressed. E-mail: zdlu4409@public.wh.hb.cn

0 Introduction

Web service composition denotes the situation that a client's or a client agency's request can not be satisfied by a single Web service, but by combining some of available component Web services. The component Web services might interact with each other concurrently by communication and exchanges of messages to carry out the task requested by the client. So Web service composition will involve in concurrency, synchronization and communication of component Web service. Obviously, a formal model of Web service composition is needed to guarantee the correctness of concurrency, synchronization and behavior of Web service composition.

In the researching field of Web service composition, there are two trends coming together attempt to deal with the problems mentioned above. On one side, the Web service composition is described in workflow specifications like Business Process Execution Language for Web Services (BPEL4WS)[1]. The composition of the flow is still manually obtained. On the other side, the semantic Web community focuses on reasoning about Web resources with terms defined in ontology. Although two approaches adopt different technical mechanism, they all need suitable formal model for describing, checking and verifying composite Web service so that correctness of Web service composition can be guaranteed.

Research papers founded in this field have proposed various formal models such as Petri-net, Pi-calculus, linear logic, etc. Web Services Flow Language (i.e. WSFL)[2], which is a net-oriented language originated from workflow language, can be regarded as Colored Petri-Net (i.e. CPN). Hamadi and Benatallah[3] proposed a Petri-Net based algebra for composing Web services. The formal semantics of the composition

operators is expressed in terms of Petri-Net by providing a direct mapping from each operator to a Petri-Net construction. Thus, any service expressed using the algebra constructs can be translated into a Petri-Net representation. Aalst and Hofstede[4] developed a Petri-Net-based workflow analyzer to enable the automatic verification of workflow process specified with Petri-Net. Rao Jinghai[5] proposed a solution to automatic semantic Web service composition based on linear logic. In the solution, the Web services and the user's requirement are both specified by DAML-S ServiceProfile. A translator is responsible for translating them into linear logic formulate. Thus the description of existing Web service is encoded as linear logic axioms, and the requirement to the composite service are specified in the form of a linear logic sequence to be proven. Lumpe[6] developed a formal language for software composition based on a variant of Pi-calculus and defined a Java-based composition system.

Hoare's communicating sequential processes (CSP)[7,8] is another kind of formal model for describing and modeling concurrent systems whose component processes interact with each other by communication. In this paper, we adopt CSP to describe and to model Web service composition. We study the BPEL4WS specification and extract the composition constructors of BPEL4WS, then establish a set of mapping rules for translating composition constructors of BPEL4WS to CSP notations. We have designed a translation algorithm for translating BPEL4WS document to CSP model, which can be used to check and verifying BPEL4WS implementation with model verification tools of CSP.

1 Basic Conception of CSP

CSP is a model language for describing concurrent and distributed computation. The CSP model is based on the idea that several sequential processes are running in parallel to each other. A CSP program is a static set of explicit processes. Pairs of processes communicate synchronously by naming each other in input and output statements.

1.1 Event and Process

In CSP specification, an event is defined as an action without duration. Events can also be named as actions or commands. A process is defined as the set of events that are relevant. This set is called alphabet. The behavior of a process is defined by a set of process names and operators. The following conventions are used in CSP notation:

① Words in lower-case letters denote distinct events, e.g. a, b, c, d.

② Words in upper-case letters denote specific defined processes, e.g. P, Q, R.

③ The alphabet of Process P is denoted as α_P.

Let x be an event and let P be an process, $x \in \alpha_P$, then

$$x \rightarrow P$$

describes a process that first engages in the event x and then behaves exactly like P. Note that $\rightarrow$ operator always takes a process on the right and a single event on the left. Thus we can carefully distinguish the concept of an event from that of a process that engages in events. Event x is called prefix of process P. A process description that begins with a prefix is said to be guarded.

If x and y are distinct events, then

$$(x \rightarrow P \mid y \rightarrow Q)$$

describes a process which initially engages in either events x or y. Since x and y are different events, the choice between P and Q is determined by the first event that actually occurs.

The CSP specification also defines two special processes named SKIP and STOP. The process that never actually engages in any of the events is called STOP. SKIP is defined as a process that does nothing but terminate successfully.

1.2 Recursion

In CSP specification, a process can be defined recursively to describe repetitive behaviors. For example,

$$\text{clock} = (\text{tick} \rightarrow \text{clock})$$

describes a clock which always tick. If $F(X)$ is a guarded expression containing the process name X, then the process X can be defined recursively as:

$$X = F(X).$$

1.3 Communication

In CSP specification, communications between concurrent processes is regarded as special class of an event. A communication is described by a pair of $c.v$ where c is the name of the communication channel and v is the value of messages that pass.

A process which first outputs v on the channel c and then behaves like P is defined as

$$c!v \rightarrow P$$

The only event in which this process is initially prepared to engage is communication event $c!v$.

A process which is initially to input any value x on the channel c, and then behave like P, is defined as

$$c?x \rightarrow P$$

1.4 Concurrency

If P and Q are processes, then the notation

$$P \,||\, Q$$

denotes the process which behaves like the system composed of processes P and Q interacting in synchronization.

1.5 Sequential Processes

If P and Q are sequential processes with the same alphabet, their sequential composition

$$P;Q$$

is a process that first behaves like P; but when P terminates successfully, $(P;Q)$ continues by behaving as Q.

A process P which repeats similar actions is denoted as $*P = P;P;P\cdots$

If x is a program variable and e is an expression and P a process, then

$$(x := e;\ P)$$

is a process that behaves like P, except that the initial value of x is assigned by the value of expression e. Initial values of all other variables are unchanged. Assignment by itself can be given a meaning by

$$(x := e) = (x := e; \mathrm{SKIP})$$

If P and Q are processes, then

$$P \lhd b \rhd Q \ (P \text{ if } b \text{ else } Q)$$

is a process which behaves like P if the initial value of b is true, or like Q if the initial value of b is false.

If iterative process P is performed until the given Boolean condition b no longer holds true, then iterative process P will be written as $b \cdot Q$.

2 Modeling Web Service Composition with CSP

In order to describe a Web service composition using CSP, a relationship between conceptions of Web services composition and CSP is established first. A component Web service is represented by a CSP process. Interaction between component Web services such as invoke, request, response can be represented by a CSP communication event. Let's view the example shown in Fig. 1.

Figure 1 describe a composite Web service that can handle a purchase order from client agency. The composite service is composed of three component services: Order Service, Ship Service, and Invoice Service. On receiving a purchase order from client, the Order service will initiates Ship service and Invoice service concurrently by sending them the message PO along communication channels shipping and invoicing respectively. Ship and Invoice services will reply message ShipInfo and Invoice to Order service along the same channel after they have carried out the shipping schedule and price calculation respectively. At last Order service will send ShipInfo and Invoice back to client. Fig. 1 shows that: The Order Service communicates with Client Agency, Ship Service and Invoice Service along channel purchasing, shipping and invoicing respectively and the messages exchanged with each other are PO, ShipInfo and Invoice.

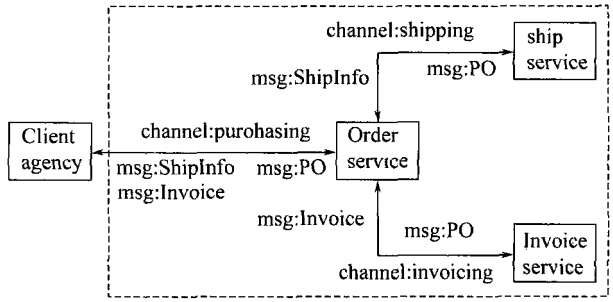

Fig. 1 A composite Web service example

So we can model Order Service as a CSP process OrderServiceProcess which is defined as:

OrderServiceProcess = purchasing ?PO $\rightarrow$(shipping ! PO $\rightarrow$SKIP|| invoicing !PO $\rightarrow$SKIP); (shipiping ?ShipInfo $\rightarrow$SKIP|| invoicing ?Invoice $\rightarrow$SKIP); (purchasing !Invoice $\rightarrow$SKIP|| purchasing !ShipInfo $\rightarrow$SKIP).

Similarly, the Ship service and Invoice service are modeled as:

ShipServiceProcess = shipping ?PO $\rightarrow$shipping ! ShipInfo $\rightarrow$SKIP;

InvoiceServiceProcess = invoicing ?PO $\rightarrow$invoicing ! Invoice $\rightarrow$SKIP;

Finally, the composite process can be modeled as:

CompositeProcess = OrderServiceProcess|| ShipServiceProcess|| InvoiceServiceProces.

3 Mapping BPEL4WS to CSP

BPEL4WS is an emerging standard for specifying and executing workflow specifications for Web service composition invocation. BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. BPEL4WS specification defines following constructors to describe the behavior of a

composite business process, as shown in Table 1.

Given a business process is described by BPEL4WS specification, however, it will also raise the question of whether this composition is verified, is there any potential deadlock and whether the correctness of concurrency and synchronization can be guaranteed. Technically BPEL4WS lacks in verification of composite Web service. In order to systematically evaluate the capabilities and limitations of BPEL4WS, P. Wohed *et al*[9] proposed a framework based on existing workflow and communication patterns that could be used for an in-depth analysis of BPEL4WS. H. Foster *et al*[10] translated BPEL4WS program into FSP process and verified FSP process by LTSA tool. If BPEL4ES implementation can be modeled by CSP, we can verify composite Web service by means of CSP's model verification tools.

To ease the modeling of the BPEL4WS against CSP, the BPEL4WS activities listed in Table 1 are divided into four groups: structure, concurrent, primitive and compensation. Structure represents structured activities executing on the traditional structured programming design principles of sequence, selection and iteration. Concurrent specifies the parallel activities. Primitive represents those activities that are atomic. Compensation activities provide the mechanism for dealing with execution error and execution rollback.

Table 1 BPEL4WS Constructors

Process		Top level abstract process
Partners		Web services the process interact with
Variables		Data used by the process(message data)
CorrelationSets		Dependencies between messages
Primitive activities	Invoke	Invoke an operation on a partner
	Receive	Receive invocation from a partner
	Reply	Send a reply message in partner invocation
	Assign	Data assignment between containers
	Throw	Detect processing error and throws exception
	Wait	Execution stops for a given time period
	Empty	Do nothing
	Terminate	Terminate a Service Instance
Compensation activities	Fault Handlers	Deals with faulty conditions
	Compensation Hanler	Undo an action
Structured activities	Sequence	Execute activities sequentially
	While	Iterate execution of activities until conditions violated
	Switch	Test conditional branch and execute corresponding activities
	Pick	Wait the occurrence of one of a set of events and then performs the activity associated with the event that occurred
Concurrent activities	Flow	Execute activities in parallel

In order to translate BPEL4WS notation to CSP notation, a relationship between the conceptions of BPEL4WS and ones in the CSP must be established too. A business process described by BPEL4WS corresponds to a CSP's process. Various activities that describe the behavior of a business process can also be mapped to processes of CSP. A partnerLink that characterizes the interaction between a pair of Web Service can be translated to an inter-process communication channel of CSP. Variables stand for message of communication. Table 2 lists the conceptions mapping between BPEL4WS and CSP.

Table 2 Conceptions mapping between BPEL4WS and CSP

Conceptions of BPEL4WS	Conceptions of CSP
process	process
activity	process
partnerLink	communication channel of process
variables	communication message

In Section 3.1 to 3.3, we will define the translation rules of BPEL4WS structured, concurrent and primitive activities to the CSP notation.

3.1 Structured Activities

3.1.1 Sequence

The sequenctial structured activity of BPEL4WS can

be mapped to a CSP process that is defined as: If P and Q are sequential with the same alphabet, their sequential composition $P;Q$ is a process which first behaves like P; but when P terminates successfully, $(P;Q)$ continues by behaving as Q. If P never terminates successfully, neither does $(P;Q)$. The rule of sequence activity translation is shown in Table 3.

Table 3 Sequence translation

BPEL4WS notation	CSP notation
〈sequence〉	ACT_1 = action$_1$ →SKIP
activity$_1$	ACT_2 = action$_2$ →SKIP
activity$_2$	ACT_3 = action$_3$ →SKIP
activity$_3$	SEQUENCE =
〈/ sequence〉	(ACT_1) ; (ACT_2) ; (ACT_3)

3.1.2 Switch

The switch-structured activity of BPEL4WS can be mapped to a CSP process that is defined as: If P and Q are processes, then

$$P \lhd b \rhd Q \quad (P \text{ if } b \text{ else } Q)$$

is a process that behaves like P if the initial value of b is true, or like Q if the initial value of b is false. The rule of switch activity translation is shown in Table 4.

Table 4 Switch translation

BPEL4WS Notation	CSP notation
〈switch〉	CASE = ACT_1
〈case condition	ACT_1 = action$_1$ →SKIP
= bool-expr〉	OTHERWISE = ACT_2
activity$_1$	ACT_2 = action$_2$ →SKIP
〈/ case〉	
〈otherwise〉	SWITCH =
activity$_2$	(CASE)|< bool-expr|>
〈/ otherwise〉	(OTHERWISE)
〈/ switch〉	

3.1.3 While

The while-structured activity of BPEL4WS can be mapped to a CSP process that is defined as: If iterative process P is performed until the given Boolean condition b no longer holds true, then iterative process P will be written as

$$b \cdot Q$$

The rule of while-activity translation is shown in Table 5.

3.1.4 Pick

The structured pick activity of BPEL4WS awaits the occurrence of one of a set of events and then performs the activity associated with the event that occurred. Each pick activity must include at least one onMessage event that is represented by 〈onMessage〉 constructor.

Table 5 While translation

BPEL4WS notation	CSP notation
〈while	ACT = action →SKIP
condition = bool-expr〉	
activity	WHILE =
〈/ while〉	bool-expr ·(ACT)

〈onMessage
partnerLink = "ncname" portType = "qname"
operation = "ncname" variable = "ncname"〉
activity
〈/ onMessage〉

The semantics of the onMessage event is identical to receive activity. A receiving activity can be translated to the input event defined in CSP. The partnerLink attribute of onMessage constructor can be mapped to an input channel of CSP. Similarly, the variable attribute of onMessage constructor can be translated to the message of input event.

The structured pick activity of BPEL4WS can be mapped to a CSP process that is defined as: if x and y are distinct events, then

$$(x \rightarrow P \mid y \rightarrow Q)$$

describes a process which initially engages in either events x or y. Since x and y are different events, the choice between P and Q is determined by the first event that actually occurs. The rule of pick activity translation is shown in Table 6.

Table 6 Pick translation

BPEL4WS notation	CSP notation
〈pick〉	ONMESSAGE$_1$ =
〈onMessage	plname$_1$?vname$_1$ →ACT_1
partnerLink = "plname$_1$"	ACT_1 = action$_1$ →SKIP
variable = "vname$_1$"	ONMESSAGE$_2$ =
activity$_1$	plname$_2$?vname$_2$ →ACT_2
〈/ onMessage〉	ACT_2 = action$_2$ →SKIP
〈onMessage	
partnerLink = "plname$_2$"	PICK =
variable = "vname$_2$"	(ONMESSAGE$_1$|
activity$_2$	ONMESSAGE2)
〈/ onMessage〉	
〈/ pick〉	

3.2 Concurrent Activity

The concurrent flow activity of BPEL4WS can be mapped to a CSP process that is defined as: If P and Q

are processes with the same alphabet, then the notation

$$P \| Q$$

denotes a process which is composed of processes P and Q running concurrently. The rule of flow activity translation is shown in Table 7.

Table 7 Flow translation

BPEL4WS Notation	CSP Notation
〈flow〉 activity$_1$ activity$_2$ 〈/flow〉	$ACT_1 = action_1 \rightarrow SKIP$ $ACT_2 = action_2 \rightarrow SKIP$ $FLOW = (ACT_1) \,||\, (ACT_2)$

3.3 Primitive Activity

3.3.1 Invoke, receive and reply

The primitive invoke activity allows the business process to invoke a request-response operation on a portType offered by a partner. It can be regarded as the input/output event between business process issuing the invocation and corresponding partner's process. The partnerLink attribute can be translated to a communication channel, whist inputVariable and outputVariable can be mapped to the message of input/output event. Similarly, the primitive receiving activity is mapped to an input event of CSP and the primitive replying activity corresponds to an output event in the same way. The rules of invoke, receive and reply activity translation are shown in Table 8.

Table 8 Invoke, receive and reply translation

BPEL4WS notation	CSP notation
〈invoke name = "INVOKE" partnerLink = "plname" inputVariable = "x" outputVariable = "y" … /〉	INVOKE = plname $!x \rightarrow$ plname $?y \rightarrow$ SKIP
〈receive name = "RECEIVE" partnerLink = "plname" variable = "x" …/〉	RECEIVE = plname $?x \rightarrow$ SKIP
〈reply name = "REPLY" partnerLink = "plname" variable = "y" …/〉	REPLY = plname $!y \rightarrow$ SKIP

3.3.2 Assign

The primitive assign activity of BPEL4WS can be mapped to a CSP process that is defined as: If x is a program variable, and e is a expression, then

$$x := e; SKIP$$

denotes the value of x is defined by value of expression e. The rule of assign activity translation is shown in Table 9.

Table 9 Assign translation

BPEL4WS notation	CSP notation
〈assign〉 〈copy〉 〈from variable = "y_1"〉 〈to variable = "x_1"〉 〈/copy〉 〈copy〉 〈from variable = "y_2"〉 〈to variable = "x_2"〉 〈/copy〉 〈/assign〉	$COPY_1 =$ $(TO_1 := FROM_1; SKIP)$ $TO_1 = x_1$ $FROM_1 = y_1$ $COPY_2 =$ $(TO_2 := FROM_2; SKIP)$ $TO_2 = x_2$ $FROM_2 = y_2$ $ASSIGN =$ $(COPY_1); (COPY_2)$

3.3.3 Terminate and empty

The semantic of primitive terminate activity is to immediately terminate the behavior of a business process within which the terminate activity is performed. It can be translated to STOP that is defined in CSP as: The process that never actually engages in any of the events is called STOP. Similarly, the primitive empty activity corresponds to SKIP which is defined in CSP as a process that does nothing but terminate successfully.

4 Translation Algorithm and Model Verification

Section 3 defines the translation of BPEL4WS structured, concurrent and primitive activities to the corresponding CSP notation. In this section we will introduce the algorithm for translating a BPEL4WS document that describes a composite Web service to a CSP model. The whole idea about the algorithm is: The algorithm parses the BPEL4WS document from the root node process and its child nodes. For every node n that stands for a structured or concurrent activity, the algorithm imports a new intermediate symbol as the process name mapping from the node n. An intermediate process name of node n will be defined by next hierarchy process names that stand for the child nodes of node n until the parsing reaches a primitive node. When the parsing of whole BPEL4WS document is finished, all intermediate symbols will be substituted recursively and the composite Web service

will be only expressed by atomic process mapping from primitive activities.

In order to avoid conflict of the intermediate symbol, we define the naming rules of intermediate symbol as follows:

Rule 1 Intermediate symbol is named by corresponding node name.

Rule 2 Intermediate symbol is indexed.

Rule 3 The index of intermediate symbol of node N is of two dimensions. Hierarchy of node N in the node tree determines the first dimension and the second dimension is determined by whether there are any nodes whose tag name and hierarchy are equal to that of node N.

Rule 4 All intermediate symbols at right side of equation are put in parentheses.

To illustrate the naming rules, an example node tree is shown in Fig. 2.

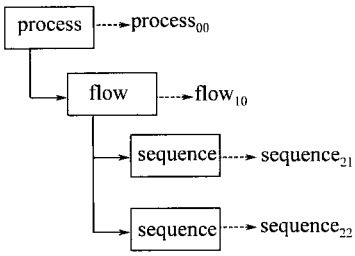

Fig. 2 Example node illustrating naming rules

In the example node tree, the hierarchy of node process is at the top, so the algorithm imports an intermediate symbol $process_{00}$ to denote the corresponding CSP process of node process. The node flow has two identical child nodes and the hierarchy of these two nodes is 2, so we import two new intermediate symbols $sequence_{21}$ and $sequence_{22}$ to denote the corresponding CSP process of them respectively.

With the naming rules of intermediate symbol, the algorithm can be described as follows:

Step 1 Locate the root node process.
Import intermediate symbol $process_{00}$.
CurrentSymbol = $process_{00}$.
CurrentNode = process.

Step 2 Parse the child node N of process node.
Import intermediate symbol S for child node N.
$process_{00} = (S)$.
CurrentSymbol = S.
CurrentNode = N.

Step 3 Parse the child nodes of CurrentNode.
If CurrentNode has child nodes then
Set $N = 0$
For each child node N_i
Import new intermediate symbol S_i.
$N = N + 1$.
End For
Define CurrentSymbol using next hierarchy intermediate symbol S_i ($i = 1, \cdots, N$) according to the mapping rules described in Section 3.
For each child node N_i
if N_i has child nodes then
CurrentSymbol = S_i.
CurrentNode = N_i.
repeat step 3 recursively.
End if
End For
End If

Step 4 For each node N_i that has no child node
Define the symbol of N_i according to map rules described in section 3.
End For

Step 5 Substitute all intermediate symbol until the composite Web service is only expressed by atomic process mapping from primitive activities.

In order to illustrate the algorithm, we present a simple example of a BPEL4WS process for handling a purchase order. The process is composed of nesting structured activities. The workflow of the process is very simple: On receiving a purchase order from customer, the process initiates two subtasks concurrently: selecting a shipping and calculating price. When the two subtasks completes, an invoice is sent to customer. The BPEL4WS process is defined with BPEL4WS as follows:

```
〈process name="purchaseOrderProcess" …〉
 〈sequence〉
  〈receive partnerLink="purchasing"
      variable="PO" …/〉
  〈flow〉
    〈sequence〉
     〈assign〉
      〈copy〉
       〈from variable="PO" …/〉
       〈to variable="shippingRequest" …/〉
      〈/copy〉
     〈/assign〉
     〈invoke partnerLink="shipping"
      inputVariable="shippingRequest"
      outputVariable="shippingInfo" …/〉
```

```
    〈receive partnerLink ="shipping"
      variable ="shippingSchedule" …/〉
  〈/sequence〉
  〈sequence〉
   〈invoke partnerLink ="invoicing"
    InputVariable ="shippingInfo" …/〉
   〈receive partnerLink ="invoicing"
    variable ="Invoice" …/〉
   〈/sequence〉
  〈/flow〉
  〈reply partnerLink ="purchasing"
   variable ="Invoice" …/〉
 〈sequence/〉
〈/process〉
```

We can translate this BPEL4WS document to CSP with following steps:

Step 1 $process_{00} = (sequence_{10})$

Step 2 $sequence_{10} = (receive_{20}); (flow_{20}); (reply_{20})$

Step 3 $receive_{20} = purchasing?PO \rightarrow SKIP;$
$flow_{20} = (sequence_{31}) || (sequence_{32})$
$reply_{20} = purchasing!Invoice \rightarrow SKIP$

Step 4 $sequence_{31} =$
$(assign_{40}); (invoke_{41}); (receive_{41});$
$sequence_{32} = (invoke_{42}); (reveive_{42})$

Step 5 $assign_{40} = shippingRequest := PO$
$invoke_{41} =$
$shipping!shippingRequest \rightarrow shipping?$
$shippingInfo \rightarrow SKIP$
$receive_{41} =$
$shipping?shippingSchedule \rightarrow SKIP$
$invoke_{42} = invoicing!shippingInfo \rightarrow SKIP$
$reveive_{42} = invoicing?Invoice \rightarrow SKIP$

Step 6 $process_{00} = ((purchasing?PO \rightarrow SKIP);$
$(((shippingRequest := PO);$
$(shipping!shippingRequest \rightarrow shipping?$
$shippingInfo \rightarrow SKIP);$
$(shipping?shippingSchedule \rightarrow SKIP)) ||$
$((invoicing!shippingInfo \rightarrow SKIP);$
$(invoicing?Invoice \rightarrow SKIP)));$
$(purchasing!Invoice \rightarrow SKIP))$

Once Web service composition has been formally modeled by CSP, we can utilize its firm mathematical framework to reduction the behavior of composite Web service and verify the correctness of composition. System behavior reduction means that the behaviors of two formally modeled software systems can be reduced according the reduction laws of formal model. If the reduction results of two systems are identical, the behaviors of two systems are equivalent. Model verification can be used to guarantee the correctness of system such as deadlock avoidance, concurrency and synchronization.

As a strong and mature formal model, CSP has various supporting tools for model checking, model correctness verification and model simulation. The emergence of tools for CSP has had a profound impact on the utility of the CSP notation.

The CSP model-checking tool FDR (i.e. Failures-Divergence Refinement) developed by Roscoe[11] may be of particular interest. FDR was the first commercially available tool for CSP and played a major role in driving the evolution of CSP from a notation to a concrete language. It allows the checking of a wide range of correctness conditions, including deadlock and livelock freedom as well as general safety and liveliness properties. When these conditions are not satisfied, the reasons can be investigated.

ProBE[11] is an another tool offering from Formal Systems. In contrast to FDR's automatic checking of properties, the ProBE tool enables the user to "browse" a CSP process by following events that lead from one state of the process to another. It uses a hierarchical list to display the possible actions and states of a process in much the same way as a file system viewer shows directories and files.

CCSP[12] is an execution environment for CSP programs from the University of Missouri-Rolla, USA, provides an execution environment for CSP programs. CCSP consists of two parts, a parser and a run-time system using Berkeley sockets. The parser takes a CSP program as input and produces a C program as output. These C programs are then run as individual processes on a network of UNIX workstations.

JCSP[13] (i.e. CSP for Java) is a complete library for building complex functionality through layered networks of communicating processes. It conforms to the CSP model of communicating systems so that CSP theory, tools and practical experience can be brought to bear in the support of Java multi-threaded applications.

In order to utilize CSP model-checking tool FDR to verify a CSP model describing a composite Web service, an important step is to translate the CSP scripts to machine-readable dialect of CSP (CSP_M). After a CSP_M script which describing a composite Web service is loaded by FDR, we can verify the correctness of composite Web

service model through FDR's operations such as deadlock checking, tracing and process debugging. The verification results can then be used to correct invalid behaviors in Web service composition design.

5 Conclusion

A composite Web service is composed of several component Web services, which interact and communicate with each other concurrently. A formal model of Web service composition is needed to guarantee the correctness of concurrency, synchronization and behavior of Web service composition. In this paper, we present a formal model CSP with which the semantic of Web service composition can be characterized and describe the mapping algorithm between BPEL4WS constructors and CSP notations. Examples for illustrating how composite Web services can be modeled by CSP are given. The methods for model checking, model verification and model simulation are also introduced.

References

[1] Andrews T, Curbera F, Dholakia H, *et al*. Business Process Execution Language for Web Services Version1.1[EB/OL]. [2003-05-05]. *http://www-128.ibm.com/developerworks/library/specification/ws-bpel/*.

[2] Leymann F. Web Services Flow Language (WSFL1.0)[EB/OL]. [2005-10-20]. *http://www-306.ibm.com/software/solutions/Webservices/pdf/WSFL.pdf*.

[3] Hamadi R, Benatallah B. A Petri Net-based Model for Web Service Composition [C]// *Database Technologies* 2003, *Proc of Fourteenth Australasian Database Conference (ADC2003)*. Sydney: Australian Computer Society, 2003: 191-200.

[4] Van der Aalst W M P, ter Hofstede A H M. Verification of Workflow Task Structures: A Petri-Net-Based Approach [J]. *Information Systems*, 2000, **25**(1): 43-69.

[5] RAO Jinghai. *Semantic Web Service. Composition via. Logic-Based Program. Synthesis* [D]. N-7491 Trondheim: Department of Computer and Information Science, Norwegian University of Science and Technology, 2004.

[6] Lumpe M. *A Pi-Calculus Based Approach to Software Composition* [D]. Switzerland: Instiute of Computer Science and Applied Mathematics, University of Bern, 1999.

[7] Hoare C A R. *Communicating Sequential Processes* [M]. New Jersey: Prentice-Hall, 2004.

[8] Hoare C A R. Communicating Sequential Processes [J]. *Communications of the ACM*, 1978, **21**(8): 666-677.

[9] Wohed P, van der Aalst W M P, Dumas M, *et al*. *Pattern Based Analysis of BPEL4WS* [R]. Queensland: Queensland University of Technology, 2004.

[10] Foster H, Uchitel S, Magee J, *et al*. Model-Based Verification of Web Service Compositions[C]// *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. Montreal, Canada: IEEE Computer Society Press, 2003: 152-161.

[11] Roscoe A W. *The Theory and Practice of Concurrency* [M]. New Jersey: Prentice-Hall, 1997.

[12] Lutfiyya H, Mc Millin B, Arrowsmith B, *et al*. *CCSP—A Formal System for Distributed Program Debugging* [R]. Rolla, New Jersey: University of Missouri, 1994.

[13] Welch P, Austin P. CSP for Java(JCSP) [EB/OL]. [2005-11-05]. *http://www.cs.kent.ac.uk/projects/ofa/jcsp/explain.html*.

□