ISSN 1746-7659, England, UK
Journal of Information and Computing Science
Vol. 4, No. 2, 2009, *pp*. 093-098

A Process Algebra Approach of BPEL4WS

Hui-yun Long [+] and Jian-shi Li

School of Computer Science and Technology, Guizhou University, Guiyang, China, 550025

(*Received July 12, 2008, accepted December 10, 2008*)

Abstract. It is now well-admitted that formal methods are helpful for many issues raised in the Web services area. It is a feasible method of the design and the verification of Web services using process algebras. BPEL4WE can correctly combine Web services actions. It is a important part of Web Services. In this paper, we present a mapping from BPEL4WS code to value-passing CCS, which offer an available way of obtaining a formal model of BPEL4WS.

Keywords: Web services, BPEL4WS, the value-passing CCS, map.

1. Introduction

Recently, with the quick development of the e-business, Web application widely develop. Web services (WSs) is a new distributed computing and become the standards of a new web application mode. It is an active way of Web data and information. In the services of Web, XML describes the Web content and its standardized: WSDL interfaces abstractly describe the process of exchanging information; SOAP provides a protocol for exchanging structured information; UDDI is used to publish and discover WSs, BPEL4WS(BPEL for short) is a notation for describing executable business process behaviors. WSs raise many theoretical and practical issues which are part of on-going research. Some well-known problems related to WSs are to specify them in an adequate, formally defined and expressive enough language, to compose them, to discover them through the Web, to ensure their correctness.

Formal methods provide an adequate framework (many specification language and reasoning tools) to address most of these issues (description, composition, correctness). Different proposals have emerged recently of describing WSs abstractly, most of which are grounded on transition system models, for example: Labelled Transition Systems, Mealy automata, Petri nets, Process Algebra etc. [5,6,7,8,9,10,11]

Process Algebra appeared in the end of 1970s and the beginning of 1980s, including CCS, CSP, ACP and π-calculus. Process algebra is a algebraic way to solve concurrent communication, and it can describe and analyze concurrent, asynchronous, non-determined and distributed action. Process algebra is a formal language creating the model of dynamic entity. It has formal defined rigorously semantics, and it can contact between service action and operation semantic, what's more it can automatically validate their attributes. Thus, it is often used in designing and analyzing work-flowing in concurrent real-time system.

CCS[1,2] is a concurrent computing model built by R.Milner. It can catch algebraic character about concurrency and communication, and it is a algebraic model which can describe the functions of the concurrent system. In CCS model, the system is composed by the process. In system, the process can concurrently evolve, and correspond system action by communication among the processes. Early CCS can not better describe data transfer. To overcome this deficiencies, R.Milner defined the value-passing calculus. The value-passing CCS can better abstract and describe the action of the current system, and reflect data transfer in the current system. Further more, the value-passing calculus provides bisimulation analysis, with which we can establish whether two processes have equivalent behaviors. Bisimulation analysis[3,4] is useful to prove that a service can be substituted another one; another use of bisimulation is to check the redundancy and deadlock of service in a community.

[+] Hui-yun Long. Tel.: +86-133 1227 5357; fax: +86-851-362 7649.
E-mail address: long_huiyun@yahoo.com.cn .

Because process algebras support bisimulation analysis, we can apply to WSs, a well-know designed method that intuitively we start with an abstract description of a process and we refine it iteratively, obtaining a less abstract one at each step. At each stage, using bisimulation we can verify the correspondence between the current version and the previous (more abstract) one. We argue that the bisimulation can be part of the problem of automatic composition of services.

In this paper we focus on the value-passing calculus. We provide a mapping from BPEL to the value-passing calculus by illustrating BPEL codes, thus we give translation from BPEL to the value-passing calculus.

2. Value-passing CCS

The value-passing CCS（CCS for short in this paper） is a kind of computation model to describe and analyze concurrent system. In the value-passing, every expression denotes an agent, and express a concurrent entity which can freely run. The communication between agents implement by exchanging information in named channel. Now we shall define Σ, the set of agent expressions, and let $E, F, \cdots$ range over Σ.The agent can be of the following forms:

- 0: represents the process is inactive and it does not perform any actions.
- $a.E$: first performs prefix action a, the continues as E. Prefix action has three kinds of form: a) τ stands for an internal and unobservable action; b) $a(x)$ represents that variable receives values from input channel a; c) $\overline{a}(e)$ denotes that the value of variable send out along output channel a.
- $E_1 \mid E_2$: means process E_1 and E_2 which are executing concurrently. E_1 and E_2 can perform independently and also communicate each other.
- $E_1 + E_2$: represents a non-deterministic choice which either E_1 or E_2 proceeds.
- $(a)E$: creates new name a for process E, the name is private and its scope of a is limited in E.
- if b then E : represents that E will execute if Boolean expressions b is 'true'.
- $A(x_1,...,x_n)$: represents constant A with arity n. There is a defining equation: $A(x_1,...,x_n) = E$ where the right-hand side E may contain no agent variables, and no free value variable except $x_1,...,x_n$.

In giving meaning to the value-passing CCS, we shall give the general notion of a labeled transition system.

Definition 1 (Labeled Transition System (LTS)) LTS is a 3-tuple (E, A, $\{\xrightarrow{a} : a \in A\}$), where E is a set of agents, A is a set of actions and $\xrightarrow{a} \subseteq E \times E$ for each $a \in A$. According to the LTS, the operational semantics of the value-passing calculus is denoted as the following:

$$\text{Act}_1 \ \frac{}{\tau.E \xrightarrow{\tau} E} \qquad \text{Act}_2 \ \frac{}{\overline{a}(e).E \xrightarrow{\overline{a}(e)} E}$$

$$\text{Act}_3 \ \frac{}{a(x).E \xrightarrow{a(x)} E} \qquad \text{Sum}_1 \ \frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'}$$

$$\text{Sum}_2 \ \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'} \qquad \text{Com}_1 \ \frac{E_1 \xrightarrow{a} E_1'}{E_1 \mid E_2 \xrightarrow{a} E_1' \mid E_2}$$

$$\text{Com}_2 \ \frac{E_2 \xrightarrow{a} E_2'}{E_1 \mid E_2 \xrightarrow{a} E_1 \mid E_2'} \qquad \text{Com}_3 \ \frac{E_1 \xrightarrow{\overline{a}(e)} E_1', E_2 \xrightarrow{a(x)} E_2'}{E_1 \mid E_2 \xrightarrow{\tau} E_1' \mid E_2'\{e/x\}}$$

$$\text{Res} \ \frac{E \xrightarrow{a} E'}{(b)E \xrightarrow{a} (b)E'}$$

We will consider a notion of equivalence between agents. A preliminary definitions are needed.

Definition 2 (t-descendant) We shall assume an infinite set A of names, and denote by $\overline{A}$ the set of co-names. $L = A \cup \overline{A}$ is the labels, and τ is the silent or perfect action. We define $Act = L \cup \{\tau\}$ to be the set of actions. L^* and Act^* are the transitive reflexive closure of L and Act. If $t \in Act^*$, then $t^{\$} \in L^*$ is the sequence gained by deleting all occurrences of τ from t. If $t = a_1...a_n \in Act^*$, then we write $E \xrightarrow{t} E'$ if $E \xrightarrow{a_1} ... \xrightarrow{a_n} E'$. If $t = a_1...a_n \in Act^*$, then $E \Rightarrow^t E'$ if $E(\xrightarrow{\tau})^* \xrightarrow{a_1} (\xrightarrow{\tau})^* ... (\xrightarrow{\tau})^* \xrightarrow{a_n} (\xrightarrow{\tau})^* E'$. If $t \in Act^*$, then E' is a t-descendant of E iff $E \Rightarrow^{\$} E'$.

So, bearing in mind what we said about matching a τ action by zero or more τ actions, we give a notion of equibalence.

Definition 3 (Bisimulation) Bisimulation, written ≈, is a binary relation S on agents. If $(E_1, E_2) \in S$ implies, for all $a \in Act$,

(i) Whenever $E_1 \xrightarrow{a} E_1^{'}$ then, for some $E_2^{'}$, $E_2 \Rightarrow^{\$} E_2^{'}$ and $(E_1^{'}, E_2^{'}) \in S$

(ii) Whenever $E_2 \xrightarrow{a} E_2^{'}$ then, for some $E_1^{'}$, $E_1 \Rightarrow^{\$} E_1^{'}$ and $(E_1^{'}, E_2^{'}) \in S$

3. Mapping from BPEL To CCS

The translation from BPEL to CCS preserves the BPEL structure. In our presentation, we refer to Table 1 and Table 2, where we show sample code of both languages; The correspondence is the mapping from BPEL to CCS calculus. An external view of interacting WSs shows services running concurrently. Such system in CCS is described by using process expressions: it instantiates agents composed in paralleling and synchronizing on all actions and their interaction is shown by the same way. At the basis of the mapping there is a correspondence between CCS actions and BPEL interactions. The direction from BPEL to CCS is straightforward. We simply automatically build a main behavior containing the instantiation of all the agents. To describe BPEL behaviors, in CCS we have the process definition. In CCS a defined agent can be instantiated with names passing. From BPEL to CCS, we use the service description to generate both the agents definition and the agent instantiations.

3.1. Basic Behaviors

The core of BPEL model is the interaction between partners. All BPEL basic activities perform interactions between WSs. An interaction is characterized by the partner link and the communication between partners. In parallel, CCS has the concept of agent to describe synchronizations among agents by names. When process or services are instantiated, CCS synchronizing agents are equivalent to BPEL interactions. When the agent representing a service is defined, a name is simply an emission or a reception. In the emission case, the parter link and operation in BPEL are stored on the receiver, on the sender in the reception case. This name can contain the information of the interaction, as shown in Table 1.

Table.1: The mapping: examples for basic behaviors from BPEL to CCS

BPEL Codes	CCS Processes
< ...act1> < /act1 >	$(a)(\ldots \text{act1} \,.\, \bar{a}(5) \mid a(x) \,.\text{act2} \ldots)$
< assign ... > < copy > <from expression="5"/ > < to var="x"/ > < /copy > < /assign >	
< act2 ...> < /act2 >	
< receive ... variable="m" > < /receive >	$a(m)$
< reply ... variable="m" > < /reply >	$\bar{a}(m)$
< invoke ... invar="mS" Outvar="mR" > < /invoke >	$\bar{a}(mS) \,.\, c(mR)$

Let us go forward in more details. In CCS, ordinal structure of behaviors and the transfer of information

are reflected by prefix operator. (a) illuminate that a is a private channel, which guarantees that the channel is used in specified fields. (a)(*...act1* . $\overline{a}$ (5) | a (x) *.act2...*) denotes that it passed the value to the variable x by channel a after *act1* is executed, and then enable *act2*. '...' specify that it is considered. In order to guarantee *act1* and *act2* to be executed, we tolerate that channel a don't appear in which.

In BPEL, receiving a message is expressed by *receive* activity and a prefixing operator with an input in CCS. The emission is written with the *reply* or the *asynchronous* invoke activity in BPEL whereas in CCS we use a prefixing operator with an output. The BPEL synchronous invoke, performing two interaction, sending a request and receiving a response, corresponds in CCS to an output followed by an input. In CCS we use two different names, because we have two interactions in BPEL.

3.2. Structured Behaviors

Table.2: The mapping: examples for structured behaviors from BPEL to CCS

BPEL Codes	CCS Processes
< pick ... >	
< onMessage ... variable="m1" >	a (m1).act1 + c(m2).act2
< act1 >	
< /onMessage >	
< onMessage ... variable="m2" >	
< act2 >	
< /onMessage >	
< /pick >	
< sequence ... >	
< ...act1 >	(a)(...act1. $\overline{a}$ (0) | a (x).act2...)
< act2 ...>	
< /sequence >	
< flow ... >	
< ...act1 >	(a)($link$)(...act1. $\overline{a}$ (0) |
< source linkname="link"	(a (y). if (cond) then $\overline{link}$ (1)
Condition="cond"/ >	+ a (y).if (¬ cond) then $\overline{link}$ (0))
< /act1 >	| $link$ (x). if (x=1) ...act2...)
< ...act2... >	
< target linkname="link"/ >	
< /act2 >	
< /flow >	
< switch >	
< case condition=	if (x>=0) then ...act1...
"bpws:getVariableData(x)>=0" >	+
< ... act1... >	if (x<0) then ...act2...
< / act1 >	
< /case >	
< otherwise >	
< ...act2... >	
< / act2 >	
< /otherwise >	
< /switch >	
< while condition =	A(x)
"bpws:getVariableData(x)>=0" >	
<... act1 >	(A(x):= if(x>=0) then ...act1.A(x))
< / act1 >	

< /while>

There is a relevant structure in CCS, Corresponding to the BPEL structured activities. As shown in Table 2, the *pick* BPEL activity is executed when it receives one message defined in one of its *onMessage* tag. In CCS, the equivalent construct is obtained by using the non deterministic choice '+', in which the action of each branch is an input prefix. It is chosen when an output prefix appears.

The *sequence* activity in BPEL matches with the CCS operator '.', which represents the ordinal operation. The action of channel a is which enable *act2* after *act1* was over. The variable x do not appear in *act2* because the value is not passed. In BPEL we have the flow activity, in CCS the simulation is implemented by using the operator '|'. The mapping of the *link* tag is more complicated, because CCS does not have an explicit construct of dependence relation. In BPEL we specify with the *source* tag the activity that has to occur first, and with the *target* tag, the dependent activity. In CCS we have a name for each link. These names are put after the end of the source behavior, and before the beginning of the target one; the two behaviors synchronizes on these actions. In Table 2, in the flow sample, activity *act2* can be executed only both after executing activity *act1* and the condition *cond1* is true. In CCS after executing *act1*, we execute the output prefix $\overline{a}$ which enable the name *link* and carry the value 1 if the condition *cond* is true, 0 otherwise; *act2* can be executed only if the condition is true and after *act1*, because it can be executed only after the operator *link* enabled.

The *switch* tag defines an ordered list of case tag in BPEL. A case corresponds to a possible activity which may be executed. The condition of a case is a Boolean expression on variables. In our process algebra we have an agent expression and non deterministic choice. In BPEL, the *while* tag correspond to the process *A(x)* in CCS. We define that *A(x)* is '*if(x>=0) then ...act1.A(x)*'. *A(x)* is a recursion, and it means a circulation.

4. Example

In this section, we give an example about the agent services of travel agency, to model the Web service composition by CCS and verify that deadlock do not appear in this service.

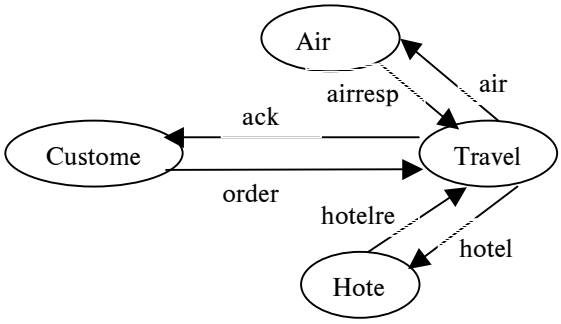

Fig. 1: An agent services of the travel agency.

A typical Web service composition of travel agency's agent is as follows: the customer send a request of the order form by the channel *order*; the travel agency will order aircrews and rooms for the airline and the hotel after receiving the order form; the airline and the hotel will return a reply to the travel agency; the travel agency give a result to the customer in the end. As shown in Fig.1, we describe the business process by CCS. In Fig.1, the ellipses represent the processes; the real lines stand for the fixed communication channels between processes. From the customer's angle, the service is a single composition service. The channel are *order* and *ack* between the services and the customer. *Travel* is a main coordinator, and is responsible for transferring and composing sub-service *Air* and *Hotel.* Travel services is described by BPEL4WS is as follows:

```
<process name="Travel"...>
  <sequence>
     <receive partnerLink="order", variable="orderreq".../>
    <flow>
      <sequence>
       <invoke partnerLink="air", outputVariable="resulta", inputVariable="orderreq".../>
```

```
    <reply partnerLink="order", variable="resulta".../>
  </sequence>
  <sequence>
    <invoke partnerLink="hotel", outputVariable="resulth", inputVariable="orderreq".../>
    <reply partnerLink="order", variable="resulth".../>
  </sequence>
 </flow>
</sequence>
</process>
```

Here, we omit the BPEL4WS description of the Air, Customer and Hotel. About these description, we can obtain the translation concerning CCS:

- Travel ($\underline{order}$, *air*, *hotel*, *ack*)=(airresp) $\underline{order}$(orderreq). ($\overline{air}$.(orderreq). *airresp*(resulta). $\overline{ack}$(resulta). 0 | $\overline{hotel}$(orderreq). *hotelresp*(resulth). $\overline{ack}$(resulth). 0)
- Customer(*order*, orderreq, *ack*)= $\overline{order}$ (orderreq). (*ack*(resulta). 0 | *ack*(resulth). 0)
- Air(*air*, resulta)=(*airresp*) *air*(orderreq). $\overline{airresp}$ (resulta). 0
- Hotel(*hotel*, resulth)=(*hotelresp*) *hotel*(orderreq). $\overline{hotelresp}$ (resulth). 0

It is easy to prove that the bisimulation is obtained between the whole system *Customer*|*Travel* | *Air*| *Hotel* and 0, that is *Customer*|*Travel* | *Air*| *Hotel*≈0. It specifies that the system can achieve end. Thus, the deadlock do not appear in this system.

5. Conclusion

We gave a mapping from BPEL codes to the value-passing CCS, and we can make the BPEL activities model by using the value-passing CCS. We can verify and research BPEL reliability and correctness and at the same time apply model analysis to BPEL's design because the formal method can supply not only model checking but also bisimulating. We do not consider dynamic process interaction about the given mapping. It is the next job in our research.

6. References

[1] R.Milner. *A Calculus of Communicating systems*. Lecture notes in Computer Science 92, Springer-Verlag, 1980.

[2] R.Milner. *Communication and Concurrency*. New York, Prentice Hall, 1989.

[3] R.D.Nicola and M.Hennessy. Testing equivalences for processes. *Theoretical Computer Science*. 1984, **34**(1-2): 83-133.

[4] S.Abramsky. Observation equivalences as a testing equivalence. *Theoretical Computer Science*. 1987, **53**(2-3):225-241.

[5] S.Narayanan and S.McIlraith. Analysis and Simulation of Web Services. *Computer Networks.2003,***42**(5): 675-693.

[6] R.Hamadi and B.Benatallah. A Petri Net-based Model for Web Service Composition. *In K.-D. Scheme and X.zhou,editor, Proc.of ADC'03,volume 17 of CRPIT.* Australia, Australia Computer Society, 2003.

[7] X.Fu,T.Bultan,and J.Su. Analysis of Interacting BPEL Web Services. *In Proc.of www'04*. USA, ACM Press, 2004.

[8] A.Lazovik, M.Aiello, and M.P.Papazoglou. Planning and Monitoring the Execution of Web Service Requests. In M.E.Orlowska, S.Weerawarana, M.P.Papazoglou, and J.Yang, editors, *Proc. of ICSOC'03*, volume 2910 of LNCS. Italy, pp.330-350, 2003.

[9] D.Berardi, D.Calvanese, G.De Giacomo, M.Lenzerini, and M.Mecella. Automatic Composition of E-services That Export Their Behavior. In M.E.Orlowska, S.Weerawarana, M.P.Papazoglou, and J.Yang, *editors Proc. of ICSOL'03*, volume 2910 of *LNCS.* Italy, pp.45-58, 2003.

[10] R.Hull, M.Benedikt, V.Christophicles, and J.Su. E-Services: a Look Behind the Curtain. *In ACM, editor, Proc. of PODS'03.* USA, pp.1-14, 2003.

[11] Ferrara A. Web Services: A Process Algebra Approach. *Proc of the Int'l Conf on Service Oriented Computing.* 2004, pp.1-18.