Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 207 (2008) 33–48
www.elsevier.com/locate/entcs

A Rigorous Model of Service Component Architecture

Zuohua Ding [1]

Center of Math Computing and Software Engineering
Zhejiang Sci-Tech University
Hangzhou, 310018, P.R.China

Zhenbang Chen [2]

International Institute For Software Technology
United Nations University
P.O. Box 3058, Macau

National Laboratory for Parallel and Distributed Processing
Changsha, 410073, P.R.China

Jing Liu [3,4]

Institute of Theoretic Computing
East China Normal University
Shanghai, 200062, P.R.China

Abstract

The Service Component Architecture (SCA) provides a platform-independent component model for service-oriented development. A service component with different communication mechanisms and implementation languages can be modeled in SCA. However, it lacks a formal foundation for SCA-based system specification and verification. This paper presents a formal service component signature model with respect to the specification of the SCA assembly model. Inspired by the idea of independence in SCA, a language-independent dynamic behaviour model is proposed for specifying the interface behaviour of the service component by port activities. Based on the dynamic behaviour model, the compatibility relation between components is discussed. A set of transition rules are given to map Business Process Execution Language for Web Services (BPEL) to dynamic behaviour expressions and then to Petri nets, thus the service component based system can be verified with existing tools. A case study is demonstrated to illustrate how to use our approach to constructing a web application in a rigorous way.

Keywords: Service Component Architecture, Formal Semantics, Service Composition, Petri nets, Verification.

[1] Email:zuohuading@hotmail.com
[2] Email:zbchen@iist.unu.edu
[3] Email:jliu@sei.ecnu.edu.cn
[4] Corresponding author

doi:10.1016/j.entcs.2008.03.084

1 Introduction

Nowadays service orientation becomes an important theme in software development. Based on the emerging computer network technologies, Service Oriented Computing (SOC) enables an environment in which software can be collaboratively and remotely developed rather than just locally developed. The key idea in SOC is the Service-Oriented Architecture (SOA), which provides a new paradigm for the existence and development of software system. Currently, many industrial standards are proposed in the spirits of SOA, such as Web services [25] and Open Services Gateway Initiative (OSGI) [22]. Though many specifications are presented in those standards, the main attention is usually put on the service communication and composition, such as SOAP [26], BPEL [1], *etc.* How to implement a service and what the service programming model is are still problems. Recently, in the background of SOA, the Service Component Architecture (SCA) [3] is proposed for facilitating the implementation of service based systems. The SCA definition from the official site [3] is as follows:

> "*Service Component Architecture is a set of specifications which describe a model for building applications and systems using a Service-Oriented Architecture.*"

Besides that, SCA provides a language-independent way to define and compose service components in the system, and it also supports different language-specific ways to implement the components including Java, C++, *etc.* SCA complements some service composition languages (such as BPEL) for enabling the more convenient and efficient service-based development.

From the SCA specification document [3], we can see that the SCA assembly model lacks a formal definition, and only static signature information of the service component can be defined. As a service programming model [2], it is not enough for SCA to just provide informal definitions by which only the static signature information can be defined. A rigorous definition for the programming model is required, and a model for specifying the dynamic behaviour of the service-based system is also needed. In addition, how to ensure the correctness of the service-based system is also an important problem for service-based development.

The contributions of this paper contain four parts. Firstly, a formal service component signature model is presented with respect to the specification of the SCA assembly model. Secondly, based on the signature model, a language-independent dynamic behaviour model is proposed for specifying the interface behaviour of the service component by port activities. Thirdly, a method for translating BPEL to the dynamic behaviour expression is given for showing that the dynamic behaviour model is rich enough in expressivity. Lastly, based on the dynamic behaviour model, we present a verification method for the service component based system by translating the dynamic behaviour model to a Petri nets and using the existing tools for verification.

Through our approach, the component structure of the SCA assembly model is defined rigorously, and some structure constraints are especially reflected. The syntax and semantics of the dynamic behaviour model can capture two different types

of communications, and support the component composition directly. The developer of the service component based system can use our approach to understanding SCA correctly, and specifying the service component formally for verification to ensure the correctness and decrease the errors in development.

The paper is organized as follows. In Section 2, a component model for the assembly model in SCA is presented and the operational semantics of the dynamic behaviour model is given. Section 3 models BPEL interface behaviour with SCA and verifies BPEL process by mapping dynamic behaviour expressions to Petri nets. Section 4 is a case study to illustrate the formalism and the verification. In Section 5, the related work is reviewed and compared. Section 6 concludes the paper and discusses some future directions.

2 Component Model for SCA

SCA aims to encompass a wide range of technologies for service components and for the accessing methods which are used to connect them. One basic artifact of SCA is the service component, which is the construction unit for service-oriented system.

2.1 Service Component Model

A service component may provide or require some services, which can be described by the operation activities as well-defined business function. Component interaction is through message exchange, and the business flow can be represented by the message exchange flow. A service component is a configured instance of a component implementation. A component provides and consumes services via ports as shown in Fig. 1.

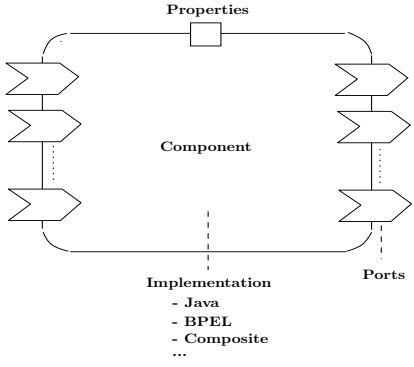

Fig. 1. SCA component diagram

Ports represent the addressable interfaces of the component and the requirement that the component has on a service provided by other component. Each component offers the business functionalities through ports, and a set of methods are contained in each port. In SCA, if the component provides a service through a port, the port is called *service port*; if the component requires a service through a port, the port

is called *reference port*. Beside that, two communication types can be supported by the port in SCA: *synchronous* communication, which means that the sending side needs to wait until the result is received; *asynchronous* communication, which means that the sending side can proceed without waiting for the result. The formal definition of a port is given as follows.

Definition 2.1 [Port] A port p is a tuple (M, t, c), where M is a finite set of methods, t is the port type that can be provided or required, and c is the communication type that can be synchronous or asynchronous.

Each method is of the form $op(\overline{T\ x}\ ;\ \overline{T\ y})$, where op is the method name, $\overline{T\ x}$ and $\overline{T\ y}$ are the input and output parameter lists, in which T is the parameter type and x, y are the parameter names. We use $p.M$ to denote the method set of port p, $p.t$ to denote the port type, and $p.c$ to denote the communication type. Two ports p_1 and p_2 are equal iff they have same methods and types, that is, $p_1.M = p_2.M \wedge p_1.t = p_2.t \wedge p_1.c = p_2.c$. For the sake of simplicity, we use $!p$ to denote the provided port (service port), $?p$ to denote the required port (reference port), $\bullet p$ to denote the synchronous port, and $\blacklozenge p$ to denote the asynchronous port.

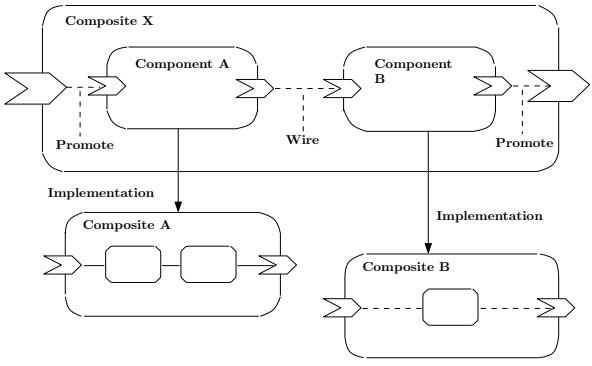

Fig. 2. SCA composite diagram

The SCA assembly model consists of a series of artifacts which define the configuration of an SCA domain in terms of composites. A composite contains assemblies of service components, the connections and related artifacts which describe how they are linked together. The composite diagram is shown in Fig. 2. An SCA *wire* within a composite connects source component required port to target component provided port. If a link is from service port to service port or from reference port to reference port, then it is called *promote*. In addition, a component can be implemented by a composite, which means that the component can be hierarchical. From the SCA specification, a unified static signature definition for SCA components and composites is given as follows.

Definition 2.2 [Component] A component Com is a tuple (P_p, P_r, G, W), in which P_p is a finite set of provided ports, P_r is a finite set of required ports, G is a finite sub component set, $W \subseteq TP \times \bigcup_{C \in G}(C.P_p \cup C.P_r)$ is the port relation that is non-reflexive, where $TP = P_p \cup P_r \cup \bigcup_{C \in G} C.P_r$, $C.P_p$ and $C.P_r$ denote the provided

and required port sets of the sub component C respectively.

A component $Com = (P_p, P_r, G, W)$ is an *atomic component* if $G = \emptyset$, otherwise Com is a *composite*. A port relation (p_1, p_2) is a *promote* if $p_1.t = p_2.t$, and (p_1, p_2) is a *wire* if $p_1.t = \mathsf{required} \wedge p_2.t = \mathsf{provided}$. In SCA specification, there are some constraints on the static signature definition of composite, such as the wire compatibility. Consistency has been defined to represent that the component is statically well-composed [17].

Besides the static consistency checking, the formal reasoning or verification during the development process cannot be supported using the static signature information. We use port activities to describe the component dynamic behavior, the basic activity of which is assumed to be the message exchange between two ports. The syntax of the component dynamic behaviour expression is defined as follows.

BE	::=	$p \bullet_m$	(Synchronous sending message)
		$p_1 \bullet_m p_2$	
		$p \circ_m$	(Synchronous receiving message)
		$p_1 \circ_m p_2$	
		$p \blacklozenge_m$	(Asynchronous sending message)
		$p_1 \blacklozenge_m p_2$	
		$p \lozenge_m$	(Asynchronous receiving message)
		$p_1 \lozenge_m p_2$	
		stop	(Stop)
		$BE \ ; \ BE$	(Sequence)
		$BE \triangleleft b \triangleright BE$	(Condition)
		$b * BE$	(Loop)
		$BE \sqcap BE$	(Non-determinism)
		$BE \parallel BE$	(Parallel)
		$BE[p_1/p_2]$	(Renaming)
		$BE \setminus \alpha$	(Restriction)
		$BE[p_1 \rightarrow p_2]$	(Wiring)

Here b represents the boolean expression, whose definition is omitted, and m stands for the message. $p \bullet_m$ represents that the port p sends the message m synchronously, but p is not wired with any reference port. $p_1 \bullet_m p_2$ represents that the port p_1 synchronously sends the message m to the port p_2. α represents a set of ports. The meanings of the other basic activities are similar.

$BE[p_1/p_2]$ is the syntactic renaming, which just replaces each p_2 in BE with p_1, and can be used for specifying the promote element in SCA composite. $BE[p_1 \rightarrow p_2]$ is used to specify the wiring operation, which is the syntactic transformation defined as follows ($\odot \in \{\bullet_m, \circ_m, \blacklozenge_m, \lozenge_m\}$).

$$BE[p_1 \to p_2] = \begin{cases} p_2 \bullet_m p_1 & BE = p \ \bullet_m \wedge p = p_2 \\ p_1 \circ_m p_2 & BE = p \ \circ_m \wedge p = p_1 \\ p_2 \blacklozenge_m p_1 & BE = p \ \blacklozenge_m \wedge p = p_2 \\ p_1 \lozenge_m p_2 & BE = p \ \lozenge_m \wedge p = p_1 \\ BE_1[p_1 \to p_2] \ ; \ BE_2[p_1 \to p_2] & BE = BE_1 \ ; \ BE_2 \\ BE_1[p_1 \to p_2] \lhd b \rhd BE_2[p_1 \to p_2] & BE = BE_1 \lhd b \rhd BE_2 \\ b * BE_1[p_1 \to p_2] & BE = b * BE_1 \\ BE_1[p_1 \to p_2] \sqcap BE_2[p_1 \to p_2] & BE = BE_1 \sqcap BE_2 \\ BE_1[p_1 \to p_2] \parallel BE_2[p_1 \to p_2] & BE = BE_1 \parallel BE_2 \\ BE_1 \setminus \alpha & BE = BE_1 \setminus \alpha \wedge (p_1 \in \alpha \vee p_2 \in \alpha) \\ BE_1[p_1 \to p_2] \setminus \alpha & BE = BE_1 \setminus \alpha \wedge (p_1 \notin \alpha \wedge p_2 \notin \alpha) \\ BE & BE = p_1 \odot p_2 \end{cases}$$

From the syntax of dynamic behaviour expression, we can see that the ports for message exchange do not need to have the same methods, and the information of the communicating ports will be gotten after wiring.

2.2 Operational Semantics

The operational semantic of the dynamic behaviour expression will be presented in this subsection. We use the classical Labelled Transition System (LTS) to define an operational semantics, and the small-step operational semantics is adopted here. The transition label a can be the message activity or the internal action (τ). We use $O(a)$ to denote the communication type of the message activity a: $O(p_1 \odot p_2) = \odot$, $O(p \odot) = \odot$, where $\odot \in \{\bullet_m, \circ_m, \lozenge_m, \blacklozenge_m\}$.

For interpreting the semantics of asynchronous communication, the configuration of the transition system is $\langle BE, \mathcal{E} \rangle$, where BE is the behaviour expression and $\mathcal{E}$ is the global asynchronous activity queue, which is unbounded. We use $\mathcal{E}^\frown a$ to represent the resulting queue after adding activity a to the end of $\mathcal{E}$, $\mathcal{E} \setminus a$ to represent the resulting queue after removing the first activity a in $\mathcal{E}$, and $a \in \mathcal{E}$ to represent that the activity a is contained in the queue $\mathcal{E}$. In the following definitions, BE is used to abbreviate $\langle BE, \mathcal{E} \rangle$ if no conflict exists.

The basic synchronous semi-message activity does not have the complete communicating partner information, which will be achieved in the future. We adopt an optimistic semantics at here, and assume that the component environment will always provide a legal support for the semi-message activities in the component behaviour expression, so the basic synchronous semi-message activity will always complete. After getting the complete port information, the basic synchronous message activity can complete after synchronization.

$$\frac{}{p \bullet_m \xrightarrow{p \bullet_m} \mathsf{stop}} \qquad \frac{}{p_1 \bullet_m p_2 \xrightarrow{p_1 \bullet_m p_2} \mathsf{stop}}$$

$$\frac{}{p \circ_m \xrightarrow{p \circ_m} \mathsf{stop}} \qquad \frac{}{p_1 \circ_m p_2 \xrightarrow{p_1 \circ_m p_2} \mathsf{stop}}$$

The transition of basic asynchronous semi-message activity is similar to that of the synchronous one. The sending of asynchronous message will add the activity to the global queue. If the corresponding sending activity exists in the global queue, the asynchronous receiving activity can complete and the first appeared corresponding

activity is removed from the global queue.

$$\frac{}{p \blacklozenge_m \xrightarrow{p \blacklozenge_m} \mathsf{stop}} \qquad \frac{}{\langle p_1 \blacklozenge_m p_2, \mathcal{E}\rangle \xrightarrow{p_1 \blacklozenge_m p_2} \langle \mathsf{stop}, \mathcal{E} \frown p_1 \blacklozenge_m p_2\rangle}$$

$$\frac{}{p \lozenge_m \xrightarrow{p \lozenge_m} \mathsf{stop}} \qquad \frac{p_2 \blacklozenge_m p_1 \in \mathcal{E}}{\langle p_1 \lozenge_m p_2, \mathcal{E}\rangle \xrightarrow{p_1 \lozenge_m p_2} \langle \mathsf{stop}, \mathcal{E} \setminus p_2 \blacklozenge_m p_1\rangle}$$

There are two rules for the sequential behaviour composition $BE_1; BE_2$. The second rule means that the completion of the first behaviour will begin the second one.

$$\frac{BE_1 \xrightarrow{a} BE_1'}{BE_1; BE_2 \xrightarrow{a} BE_1'; BE_2} \qquad \frac{BE_1 \xrightarrow{a} \mathsf{stop}}{BE_1; BE_2 \xrightarrow{a} BE_2}$$

The semantics of condition expression is also defined by two rules. The evaluation of boolean expression b can be true or false, which determines the one for executing.

$$\frac{b = \mathsf{true} \wedge BE_1 \xrightarrow{a} BE_1'}{BE_1 \lhd b \rhd BE_2 \xrightarrow{a} BE_1'} \qquad \frac{b = \mathsf{false} \wedge BE_2 \xrightarrow{a} BE_2'}{BE_1 \lhd b \rhd BE_2 \xrightarrow{a} BE_2'}$$

The loop expression will continue if the boolean expression is evaluated to be true, otherwise it will complete.

$$\frac{b = \mathsf{true} \wedge BE_1 \xrightarrow{a} BE_1'}{b * BE_1 \xrightarrow{a} b * BE_1'} \qquad \frac{b = \mathsf{false}}{b * BE \xrightarrow{\tau} \mathsf{stop}}$$

The non-determinism expression can select any one branch for execution, and the semantics rules can be given as follows.

$$\frac{BE_1 \xrightarrow{a} BE_1'}{BE_1 \sqcap BE_2 \xrightarrow{a} BE_1'} \qquad \frac{BE_2 \xrightarrow{a} BE_2'}{BE_1 \sqcap BE_2 \xrightarrow{a} BE_2'}$$

The parallel expression will have different cases. If the two wired component can take the synchronous communication, they will both transit to the next configuration, and the global transition action is the internal action τ, which represents that the communication cannot be observed from the outside of the composite containing those two components.

$$\frac{BE_1 \xrightarrow{p_1 \bullet_m p_2} BE_1' \wedge BE_2 \xrightarrow{p_2 \circ_m p_1} BE_2'}{BE_1 \parallel BE_2 \xrightarrow{\tau} BE_1' \parallel BE_2'}$$

The transition of internal action or semi-message activity in one component will not need the participation of the other component, so it will not block the behaviour.

$$\frac{BE_1 \xrightarrow{a} BE_1'}{BE_1 \parallel BE_2 \xrightarrow{a} BE_1' \parallel BE_2} \quad (a \in \{\tau, p \odot\})$$

The asynchronous message activity will also not block, but it cannot be observed from the outside of the composite.

$$\frac{BE_1 \xrightarrow{a} BE_1'}{BE_1 \parallel BE_2 \xrightarrow{\tau} BE_1' \parallel BE_2} \quad (a = p_1 \blacklozenge_m p_2 \vee a = p_1 \Diamond_m p_2)$$

The switching of the expressions of the parallel expression will not change the behaviour.

$$\frac{BE_1 \parallel BE_2 \xrightarrow{a} BE_3}{BE_2 \parallel BE_1 \xrightarrow{a} BE_3}$$

The message exchange can only occur between non-restricted ports.

$$\frac{BE_1 \xrightarrow{a} BE_2}{BE_1 \setminus \alpha \xrightarrow{a} BE_2 \setminus \alpha} \quad (a = p_1 \odot_m p_2 \vee a = p_1 \odot_m) \wedge (p_1 \notin \alpha \wedge p_2 \notin \alpha)$$

The stop represents the termination of the dynamic behaviour, and it cannot engage in any action.

The dynamic behaviour of the component can be interpreted by the sequences of actions. A finite action trace tr is $a_1, a_2, ..., a_n$. We use $BE \xrightarrow{tr} BE'$ to represent that BE can transit to BE' by the preceding operational semantics rules, and the transition sequence is the action trace tr, that is, $BE \xrightarrow{a_1} BE_1, ..., BE_{n-1} \xrightarrow{a_n} BE'$.

2.3 Component Composition

In an SCA composite, the wires can be generated automatically. This feature can be reflected by the automatic composition, which can automatically relate the required ports and provided ports of two components, and generate the resulting composite by promoting the remaining non-wired ports.

At the signature level, two components $Com_i = (P_p^i, P_r^i, G_i, W_i)$ $(i \in \{1, 2\})$ are *composable* if they do not provide the same port, that is, $P_p^1 \cap P_p^2 = \emptyset$. We define the automatically related required port set as follows:

$$\mathcal{M}_p(Com_1, Com_2) = (P_r^1 \Cap P_p^2) \cup (P_r^2 \Cap P_p^1),$$

where $P_r \Cap P_p = \{p \mid p \in P_r \wedge (p.M, \mathsf{provided}, p.c) \in P_p\}$. The automatically generated wire set can be defined as follows:

$$\mathcal{M}_w(Com_1, Com_2) = \{(p_1, p_2) \mid p_1 \in \mathcal{M}_p(Com_1, Com_2) \wedge p_2 = (p_1.M, \mathsf{provided}, p_1.c)\}.$$

The automatic composition is defined as follows, where the definition of the wire set operation is:

$$BE[W] = \begin{cases} BE & if\ W = \emptyset, \\ BE[p_1 \to p_2][W \setminus \{(p_1, p_2)\}] & (p_1, p_2) \in W. \end{cases}$$

Definition 2.3 [Automatic Composition] Given two composable components $Com_i = (P_p^i, P_r^i, G_i, W_i)$ and their dynamic behaviour expressions BE_i $(i \in \{1, 2\})$, their automatic composition $Com = (P_p, P_r, G, W)$ (denoted by $Com_1 \oplus Com_2$) is defined as follows:

(1) $P_p = (P_p^1 \cup P_p^2) \setminus \{(p.M, \mathsf{provided}, p.c) \mid p \in \mathcal{M}_p\}$; (2) $P_r = (P_r^1 \cup P_r^2) \setminus \mathcal{M}_p$;

$(3)G = G_1 \cup G_2$;$(4)W = \mathcal{M}_w \cup \{(p,p) \mid p \in P_p \cup P_r\}$;$(5)BE = (BE_1 \parallel BE_2)[\mathcal{M}_w]$, where $\mathcal{M}_p$ and $\mathcal{M}_w$ are the abbreviations for $\mathcal{M}_p(Com_1, Com_2)$ and $\mathcal{M}_w(Com_1, Com_2)$ respectively.

Theorem 2.4 $Com_1 \oplus Com_2 = Com_2 \oplus Com_1$.

Theorem 2.5 *If* $\mathcal{M}_p(Com_i, Com_j) \cap \mathcal{M}_p(Com_i, Com_k) = \emptyset$, *where* $i, j, k \in \{1, 2, 3\}$, $i \neq j$, $i \neq k$, *and* $j \neq k$, *then* $(Com_1 \oplus Com_2) \oplus Com_3 = Com_1 \oplus (Com_2 \oplus Com_3)$.

Definition 2.6 [Promoting] Given a component $Com = (P_p, P_r, G, W)$ and a promote (p_1, p_2), and the dynamic behaviour expression of Com is BE, where $p_2 \in P_p \cup P_r$, the dynamic behaviour expression of the resulted component after promoting is $BE[p_1/p_2]$.

Based on the promoting definition and the behaviour semantics, the composition of specific port relation can be defined as follows, where the definition of promote set operation is:

$$BE\langle W\rangle = \begin{cases} BE & W = \emptyset, \\ BE[p_1/p_2]\langle W \setminus \{(p_1,p_2)\}\rangle & (p_1,p_2) \in W. \end{cases}$$

Definition 2.7 [Composition] Given two composable components $Com_i = (P_p^i, P_r^i, G_i, W_i)$, their dynamic behaviour expressions BE_i $(i \in \{1,2\})$, and a port relation set W, the specific composition of Com_1 and Com_2 under W is $Com = (P_p, P_r, G, W_c)$ (denoted by $Com_1 \oplus_W Com_2$), which is defined as follows:
(1) $P_p = \{p_1 \mid (!p_1, !p_2) \in W\}$;(2) $P_r = \{p_1 \mid (?p_1, ?p_2) \in W\}$;(3) $G = \{G_1, G_2\}$;
(4) $W_c = W$; (5) $BE = (BE_1 \parallel BE_2)[\mathcal{W}_w]\langle \mathcal{W}_p\rangle$, where $\mathcal{W}_w = \{(p_1,p_2) \mid (p_1,p_2) \in W \wedge p_1.t \neq p_2.t\}$, and $\mathcal{W}_p = \{(p_1,p_2) \mid (p_1,p_2) \in W \wedge p_1.t = p_2.t\}$.

Theorem 2.8 $Com_1 \oplus_W Com_2 = Com_2 \oplus_W Com_1$.

Lemma 2.9 $Com_1 \oplus Com_2 = Com_2 \oplus_W Com_1$, *where* $W = \mathcal{M}_w(Com_1, Com_2) \cup \{(p,p) \mid p \in ((P_p^1 \cup P_p^2) \cup (P_r^1 \cup P_r^2)) \setminus (\mathcal{M}_p \cup \{(p.M, \mathit{provided}, p.c) \mid p \in \mathcal{M}_p\})\}$, *and* $\mathcal{M}_p$ *is the abbreviation for* $\mathcal{M}_p(Com_1, Com_2)$.

Through the above definitions, we can calculate the dynamic behaviour expression of the composite based on the dynamic behaviour expressions of the contained components. Given the composite $Com = (P_p, P_r, G, W)$, and the dynamic behaviour expressions of the sub components, the dynamic behaviour expression BE of Com can be calculated as follows: $BE = \Pi\{BE_C \mid C \in G\}[\mathcal{W}_w]\langle \mathcal{W}_p\rangle$, where $\Pi\mathcal{B} = B_1 \parallel ... \parallel B_n$ if $\mathcal{B} = \{B_1, ..., B_n\}$, $\mathcal{W}_w = \{(p_1,p_2) \mid (p_1,p_2) \in W \wedge p_1.t \neq p_2.t\}$, and $\mathcal{W}_p = \{(p_1,p_2) \mid (p_1,p_2) \in W \wedge p_1.t = p_2.t\}$.

With the assumption that the environment will always provide what the component needs, the dynamic behaviour compatibility between two components only requires that the dynamic behaviour of the composed composite can complete, which can be defined as follows.

Definition 2.10 [Compatibility]Given two components Com_1 and Com_2, their dynamic behaviour expressions BE_1 and BE_2, and a port relation W, Com_1 and Com_2

are compatible under W (denoted by $Comp_W(Com_1, Com_2)$) if the following conditions hold: (1) Com_1 and Com_2 are composable; (2) there exists an action trace tr such that $BE \xrightarrow{tr} \mathsf{stop}$, where BE is the dynamic behaviour of $Com_1 \oplus_W Com_2$.

3 SCA Modeling and Verification of BPEL

BPEL [1] is an XML language that supports service composition and can be used to describe executable business process behavior. WSDL is now a standard for describing its interfaces. WSDL proposes a model containing two parts: in the first part there are methods; the second part defines the messages each method can send and receive. However, WSDL addresses only static interface specifications and it does not describe the observable behavior of the web service. We show that, as a language-independent dynamic behaviour model, SCA can be used to model the interface behavior of BPEL.

In BPEL, the participating services are called *partners*, and message exchange or intermediate result transformation is called an *activity*. A process thus consists of a set of activities. A process interacts with external partner services through a WSDL interface. An interaction is characterized by the partner link, the port type, and the operation involved in the two communicating partners (each partner defines these three elements for each interaction). We say that an interaction corresponds to a port of a component. For example, if an interaction is to receive a request from a client,

$$\begin{aligned}&\langle receive\ partnerLink = "client"\\ &\quad portType = "com : InsuranceSelectionPT"\\ &\quad operation = "SelectInsurance"\\ &\quad variable = "InsuranceRequest"\\ &\quad createInstance = "yes"/\rangle\end{aligned}$$

then there is a port p=(M, t, c), where

$M = \{SelectInsurance(T\,InsuranceRequest;)\}$,

t=provided,

c=asynchronous.

In this example, the *partnerLink*, *portType* and *createInstance* are all contained in the message and will be used in the management components in the future. In this paper, we only consider functionality components.

BPEL supports primitive and structural activities. Primitive activities represent basic constructs and are used for common tasks, structural activities manage the control flows. Some commonly used activities are: $\langle invoke\rangle$, $\langle receive\rangle$,$\langle reply\rangle$, $\langle sequence\rangle$, $\langle flow\rangle$, $\langle switch\rangle$, $\langle while\rangle$, $\langle pick\rangle$, *etc.*

By defining some transition rules for these constructs, we may map a BPEL process model to an SCA model, and then following the sematic model in Section 2, we can study the dynamic behaviour of BPEL. Due to the space limit, the rules can be refered to [17].

BPEL provides no way to verify correctness. A few works have been done to overcome the this problem such as mapping BPEL to LOTOS [8], translating BPEL into FSP [14], analyzing BPEL composite web services communicating through asynchronous messages [15], *etc.* Our solution is mapping dynamic behaviour expressions of SCA to Petri nets, so that we can use the existing net tools for the verification. In this way, the BPEL process model may get the following benefits: 1) it has strong scalability and can manipulate both control and data, 2) it can be verified, and 3) it stands a good position for service substitution [17]. Another reason choosing Petri net as a verification method is due to the work by Tsai and Xu [24]: Petri-nets, as the intermediate formal basis, can be transformed into the input languages of existing analysis tools such as SPIN, SMV, SMC, and IOTA.

By setting some rules, we can transform component dynamic behaviour expressions to Petri nets [16]. The details can be found in [17].

4 Case Study

In this section we present an online shop as our example process. It is a simple but realistic business process.

The process is structured into two concurrent activities: customers initial choice and order processing. These activities are synchronized by two order links. Hence, the order processing is started only if the customer sends an order right at the beginning or he sends a request, the product is available, and he decides to order after getting the confirmation. Assuming the customer has ordered a product, he gets the invoice or he is asked questions concerning his order exactly once. The payment is handled by charging the credit card. Finally, the process sends the delivery data.

The interface of the service is defined by WSDL and the interaction is defined by BPEL4WS. We may get the SCA model by transforming WSDL/BPEL defines to dynamic behavior expressions. The SCA model of the online shop is shown in Figure 3.

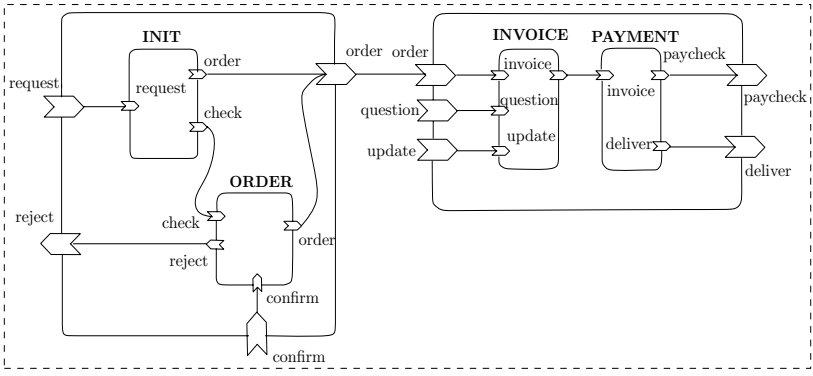

Fig. 3. SCA model of online shop.

For example, ports

$p^{11}_{request} = (\{request(T\ Req; T\ Resp)\}$, provided, synchronous),
$p^{11}_{check} = (\{check(T\ checkReq; T\ checkResp)\}$, required, synchronous),
$p^{11}_{order} = (\{order(T\ orderReq; T\ orderResp)\}$, required, synchronous),

form atomic component C^{11} = INIT with dynamic behavior expression

$$BE^{11} = p^{11}_{request}; (p^{11}_{check} \triangleleft b \triangleright p^{11}_{order}).$$

Ports p^{12}_{check}, p^{12}_{reject} and p^{12}_{order} form atomic component C^{12} = ORDER with dynamic behavior expression

$$BE^{12} = p^{12}_{check}; (p^{12}_{reject} \triangleleft b \triangleright p^{12}_{confirm}); p^{12}_{order}.$$

Ports p^{21}_{order}, $p^{21}_{question}$, p^{21}_{update} and $p^{21}_{invoice}$ form atomic component C^{21} = INVOICE with dynamic behavior expression

$$BE^{21} = p^{21}_{order}; p^{21}_{question}; p^{21}_{update}; p^{21}_{invoice}.$$

Ports $p^{22}_{invoice}$, $p^{22}_{paycheck}$ and $p^{22}_{deliver}$ form atomic-component C^{22} = PAYMENT with dynamic behavior expression

$$B^{22} = p^{22}_{invoice}; p^{22}_{paycheck}; p^{22}_{deliver}.$$

Let C^1 be the automatic composition of components C^{11} and C^{12}, then $C^1 = C^{11} \oplus C^{12} = (P^1_p, P^1_r, G_1, W_1)$, where

$$\begin{aligned}
P^1_p &= P^{11}_p \cup P^{12} \setminus \{(p.M, \text{provided}, p.c) | p \in \mathcal{M}_p)\} \\
&= \{p^{11}_{request}, p^{12}_{reject}\}, \\
P^1_r &= P^{11}_r \cup P^{12}_r \setminus \mathcal{M}_p(C^{11}, C^{12}) \\
&= \{p^{12}_{confirm}, p^{11}_{order}, p^{12}_{order}, p^{11}_{check}\} \setminus \{p^{11}_{check}\} \\
&= \{p^{12}_{confirm}, p^{11}_{order}, p^{12}_{order}\}, \\
G_1 &= \{\}, \\
W_1 &= \mathcal{M}_w \cup \{(p,p) | p \in P^1_p \cup P^1_r\} \\
&= \{(p^{11}_{check}, p^{12}_{check})\} \cup \{(p,p) | p \in P^1_p \cup P^1_r\} \\
&= \{(p^{11}_{check}, p^{12}_{check})\} \cup \{(p^{11}_{request}, p^{11}_{request}), \\
&\qquad (p^{11}_{order}, p^{11}_{order}), (p^{12}_{reject}, p^{12}_{reject}), \\
&\qquad (p^{12}_{confirm}, p^{12}_{confirm}), (p^{12}_{order}, p^{12}_{order})\}, \\
BE^1 &= (BE^{11} \| BE^{12})[\{(p^{12}_{check} \to p^{11}_{check})\}].
\end{aligned}$$

Similarly, we can get the automatic composition of components C^{21} and C^{22} as $C^2 = C^{21} \oplus C^{22} = (P^2_p, P^2_r, G_2, W_2)$ with behaviour expression:

$$BE^2 = (BE^{21} \| BE^{22})[\{(p^{22}_{invoice} \to p^{21}_{invoice})\}].$$

Assume that port relation set is $W = W_1 \cup W_2 \cup \{(p^1_{order}, p^2_{order})\}$, then a semantic model of dynamic behavior of online shop, denoted as SCA_{online}, would be

$$SCA_{online} = C^1 \oplus_W C^2$$

with behaviour expression:

$$\begin{aligned} BE = &(BE^1 \| BE^2)[\mathcal{W}_w]\langle \mathcal{W}_p \rangle, \\ &\text{where } \mathcal{W}_w = \{(p_1, p_2) | (p_1, p_2) \in W \wedge p_1.t \neq p_2.t\}, \\ &\text{and } \mathcal{W}_p = \{(p_1, p_2) | (p_1, p_2) \in W \wedge p_1.t = p_2.t\}. \end{aligned}$$

Our interest is located in computing the dynamic behavior BE. Based on Lemma 1,

$$SCA_{online} = C^1 \oplus_W C^2 = C^1 \oplus C^2.$$

Thus, the dynamic behavior

$$\begin{aligned} BE &= (BE^1 \| BE^2)[\mathcal{M}_w] \\ &= BE^1[\mathcal{M}_w] \| BE^2[\mathcal{M}_w], \end{aligned}$$

where

$$\mathcal{M}_w = \{p^{12}_{check} \to p^{11}_{check}, p^{22}_{invoice} \to p^{21}_{invoice}, p^{21}_{order} \to p^{12}_{order}, I = User - Inputs\}.$$

As an example we compute BE^1 to illustrate our method:

$$BE^1 = ((BE^{11}[\mathcal{M}_w]) \| (BE^{12})[\mathcal{M}_w]),$$

where

$$\begin{aligned} & BE^{11}[\mathcal{M}_w] \\ = \; & (p^{11}_{request} \circ; (p^{11}_{order} \blacklozenge \lhd b \rhd p^{11}_{check} \bullet))[\mathcal{M}_w] \\ \overset{I}{=} \; & p^{11}_{request} \circ [\{I, \dots\}]; (p^{11}_{order} \blacklozenge \lhd b \rhd p^{11}_{check} \bullet)[\{p^{12}_{check} \to p^{11}_{check}, \dots\}] \\ = \; & stop; (p^{11}_{order} \blacklozenge \lhd b \rhd p^{11}_{check} \bullet)[\{p^{12}_{check} \to p^{121}_{check}, \dots\}] \\ = \; & (p^{11}_{order} \blacklozenge [\{\dots\}] \lhd b \rhd p^{11}_{check} \bullet)[\{p^{12}_{check} \to p^{11}_{check}, \dots\}] \\ \overset{check}{=} \; & p^{11}_{check} \bullet [\{p^{12}_{check} \to p^{11}_{check}, \dots\}] \\ = \; & p^{11}_{check} \bullet p^{12}_{check} \end{aligned}$$

and

$$\begin{aligned} & BE^{12}[\mathcal{M}_w] \\ = \; & (p^{12}_{check} \circ; (p^{11}_{reject} \bullet \lhd b \rhd p^{11}_{confirm} \circ); p^{12}_{order} \blacklozenge)[\{p^{12}_{check} \to p^{11}_{check}, I, \dots\}] \\ = \; & p^{12}_{check} \circ [\{(p^{12}_{check} \to p^{11}_{check})\}]; \\ & (p^{11}_{reject} \bullet \lhd b \rhd p^{11}_{confirm} \circ)[\{I, \dots\}]; p^{12}_{order} \blacklozenge)[\{p^{21}_{order} \to p^{12}_{order}\}] \\ = \; & p^{12}_{check} \circ p^{11}_{check}; p^{11}_{confirm} \bullet [\{I, \dots\}]; p^{12}_{order} \blacklozenge [\{p^{21}_{order} \to p^{12}_{order}, \dots\}]. \end{aligned}$$

Thus,

$$\begin{aligned} BE^1 = &(p^{11}_{check} \bullet p^{12}_{check}) \| (p^{12}_{check} \circ p^{11}_{check}; p^{11}_{confirm} \bullet [\{I, \dots\}]; \\ &p^{12}_{order} \blacklozenge [\{(p^{21}_{order} \to p^{12}_{order}, \dots\}]) \end{aligned}$$

$$= stop \parallel (stop; p^{11}_{confirm} \bullet [\{I, \dots\}]; p^{12}_{order} \blacklozenge [\{p^{21}_{order} \rightarrow p^{12}_{order}, \dots\}])$$
$$= p^{11}_{confirm} \bullet [\{I, \dots\}]; p^{12}_{order} \blacklozenge [\{p^{21}_{order} \rightarrow p^{12}_{order}, \dots\}]$$
$$= stop; p^{12}_{order} \blacklozenge [\{p^{21}_{order} \rightarrow p^{12}_{order}, \dots\}]$$
$$= p^{12}_{order} \blacklozenge p^{21}_{order}.$$

After translating the dynamic behavior expression into a Petri nets, we can perform the analysis on the SCA model of the online shop by the following two ways: either using net tools such as Design/Net [6], or converting the Petri-net model into the input languages of analysis tools such as SPIN, SMV, SMC, and IOTA [24]. In Design/CPN, we can perform two kinds of analysis through simulation. The first can be done on the standard simulation report directly, which includes Boundedness and Liveness properties. The second is performed based on user-defined queries about the Occurrence Graph, which is the state space of the model.

5 Related Work

Because SCA is a new proposed specification, there are few existing work in formalization and verification for SCA. In [13], a small core formal language (SRML-P) is presented to specify the interaction protocol between components. SRML-P follows the ideas in SCA and provides a mathematical framework in which some service-modeling primitives are defined and application models can be reasoned about. Based on the primitives, some communication paradigm can be specified, such as synchronous, asynchronous, timeout, *etc.* Besides that, the proposed concepts, such as modules and external interface, are also related to the modeling elements in SCA. Compared with the work in [13], our work directly formalizes the SCA assembly model and the service dynamic behaviour model also supports synchronous and asynchronous communication.

In the implementation level, [4,27] propose formal models to formalize XML-based Web service languages. In [27], H. Yang *et al.* propose a formal model to WS-CDL to guarantee the correct interaction of independent web services. Based on the this model, it is possible to formally reason about the choreography in a manual way. In [4], Antonio Brogi *et al.* formalize WSCI using CCS. Compatibility and replaceability between Web services are discussed. In [23], G. Pu *et al.* develop semantics for BPEL and then to verify BPEL. Compared with them, we are concerned with the interface dynamic behavior from an orchestration view, and apply our platform-independent description mechanism to BPEL. In [28], CSP is directly used to formalize WS-CDL and BPEL, and verification can be taken after translation. We share the idea of using existing tools for verification, and the dynamic behavior model is proposed according to the SCA specification, which is convenient to model the system using SCA.

In component based development (CBD), how to construct composite components from existing ones is not new [11,12]. In the object-oriented programming community, there has been extensive research on attacking this issue, such as SuperGlue [21], Jiazzi [20], the calculus of assemblages [19], *etc.* However, our model

is constructed based on the service. It is not only directly for internet use but also provides dynamic property check in composition. When chaining components together, the verification and calculation should be based on the semantics of the behavior. In [7], static structure, dynamic behavior and refinement of component systems is proposed based on UML 2.0 superstructure. Interface Protocol State Machine (IPSM) in UML 2.0 is used to specify the interface behaviour and contract automata is presented as the transition model of IPSM. The component structure model is also hierarchical, and the stateless operation feature is included in the contract automata. Compared with the work in [7], we unify the component definition using port, and take into account different communication types in the behaviour model.

6 Conclusion

We have proposed a rigorous approach to supporting service oriented development based on service component. Service component is modeled according to the standard of service component architecture issued recently. Dynamic behavior expression is developed to describe the composition activities in service composition. A set of transition rules are given to map BPEL to dynamic behavior expression, thus the service composition process can be verified with Petri net. A case study is given to show how to use this approach to constructing web based software in a rigorous way.

Our future work includes introducing agents to route service, in order that a source can easily find the target one. We will also look into adaptable service component architecture to improve the adaptability of the component to the environment. It will be an interesting research topic to investigate how different verification techniques and tools can be applied.

Acknowledgments

This work is partially supported by the National High Tech Research 863 Program of China under Grant No.2006AA01Z165; the National Natural Science Foundation of China under Grant No.60673114, No.60603033, No.60603037, the projects HTTS funded by Macao Science and Technology Fund. The second author is partially supported by the project HighQSoftD funded by Macao Science and Technology Development Fund, the National Basic Research Program of China (973) under Grant No.2005CB321802 and NSFC under Grant No.90612009 and No.60573085.

References

[1] Arkin, F. *et al.* Web Services Business Process Execution Language Version 2.0, http://www.oasis-open.org/apps/org/workgroup/wsbpel/, 2005.

[2] Apache TUSCANY, http://cwiki.apache.org/TUSCANY/sca-overview.html, 2007.

[3] Beisiegel, M. *et al.*, *Service component architecture specification*, http://www.osoa.org/display/Main/Home, 2007.

[4] Brogi, A., C. Canal, E. Pimentel and A. Vallecillo, *Formalizing web wervice choreographies*, Electr. Notes Theor. Comput. Sci. 105 (2004), pp.73-94.

[5] Brogi, A., C. Canal and E. Pimentel, *Component adaptation through flexible subservicing*, Science of Computer Programming, Vol.63, No.1 (2006), pp.39-56.

[6] Design/CPN, http://www.daimi.au.dk/designCPN/.

[7] Dong, W., Z. Chen and J. Wang, *A contract-based approach to specifying and verifying safety critical systems*, Electr. Notes Theor. Comput. Sci. 176(2007), pp.89-103.

[8] Ferrara, A., *Web services: a process algebra approach*, in *Proceedings of 2nd ACM International Conference on Service Oriented Computing*, New York, USA, 2004, pp.242-251.

[9] Fisteus, J.A., L.S. Fernández and C. D. Kloos, *Formal verification of BPEL4WS business collaborations*, in *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web'04)*, 2004.

[10] Farahbod, R., U. Glässer, M. Vajihollahi, *Specification and validation of the Business Process Execution Language for web services*, LNCS, vol. 3052 (2004), Springer-Verlag, pp.78-94.

[11] Chen, Z.B., Z. Liu, V. Stolz, L. Yang and A. Ravn, *A Refinement Driven Component-Based Design* , in *Proc. Of ICECCS* 2007, pp. 277-289, IEEE.

[12] Chen, Z.B., Z.Liu, A. Ravn, V. Stolz and N. Zhan, *Refinement and Verification in Component -Based Model Driven Design*, Technical report 388, UNU-IIST, Nov. 2007.

[13] Fiadeiro, J. L., A. Lopes and L. Bocchi, *A formal approach to service component architecture*, in *Proc. of Third International Workshop Web Services and Formal Methods*, LNCS, vol. 4184(2006), Springer-Verlag, pp.193-213.

[14] Foster, H., S. Uchitel, J. Magee and J. Kramer, *Model-based verification of web service compositions*, in *Proc. of ASE'03*, Canada, 2003, pp.152-163.

[15] Fu, X., T. Bultan and J. Su, *Analysis of interacting BPEL web services*, in *Proc. of WWW'04*, USA, 2004, ACM Press.

[16] Girault, C. and V. Valk, Petri nets for systems engineering, a guide to modeling, verification and applications, Springer-Verlag, Berlin, 2003.

[17] Ding, Z. and J. Liu, *A rigorous model of service component architecture*, Technical report, preprint of Institute of Software Engineering, East China Normal University, Dec. 2007, will be available at: http://www.sei.ecnu.edu.cn/~jliu/.

[18] Hinz, S., K. Schmidt and C. Stahl, *Transforming BPEL to Petri nets*, LNCS, vol. 3649 (2005), pp. 220-235.

[19] Liu, Y. and S. Smith, *Modules with interfaces for dynamic linking and communication*, LNCS, vol. 3086 (2004), pp.414-439.

[20] McDirmid, S., M. Flatt and W. Hsieh, *Jiazzi: new-age components for old-fashioned java*, in *Proc. Of OOPSLA* 2001, pp.211-222, ACM.

[21] McDirmid, S. and W. Hsieh, *Superglue: component programming with object-oriented signals*, LNCS, vol.4067 (2006), pp.206-229.

[22] Open Services Gateway Initiative, OSGi Service Platform Specification, 3rd Release, 2003.

[23] Pu, G., X. Zhao, S. Wang and Z. Qiu, *Towards the semantics and verification of BPEL4WS*, Electr. Notes Theor. Comput. Sci. Vol.151, No.2 (2006), pp.33-52.

[24] Tsai, J.J.P. and K. Xu, *An empirical evaluation of deadlock detection in software architecture specifications*, Annals of Software Engineering, 7(1999), pp.95-126.

[25] W3C World Wide Web Consortium, Web Services Architecture. http://www.w3.org/TR/ws-arch/, 2004.

[26] W3C World Wide Web Consortium, Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/soap/, 2000.

[27] Yang, H., X. Zhao, Z. Qiu, G. Pu and S. Wang, *A formal model for Web Service Choreography Description Language (WS-CDL)*, ICWS 2006, pp.893-894.

[28] Yeung, W. L., *Mapping WS-CDL and BPEL into CSP for Behavioural Specification and Verification of Web Services*, ECOWS 2006, pp.297-305.