Service Selection Based On Behavior Matching

Mingyue Jiang ,Zuohua Ding
Lab of Scientific Computing and Software Engineering
Zhejiang Sci-Tech University
Hangzhou, Zhejiang 310019, China
zouhuading@hotmail.com, jiang_my@126.com

Jing Liu
Software Engineering Institute
East China Normal University
Shanghai, 200062, China
jliu@sei.ecnu.edu.cn

Abstract—**Service selection is a key step in the service composition process. The selection mainly depends on the search function of UDDI. However, UDDI provides poor search facilities: it allows only a keyword based search of web services. Such kind of the keyword search method can not capture the underlying semantics of web services, and thus can not protect us from obtaining "bad" services since UDDI registry contains checked and unchecked services. In this paper, based on the formalized Service Component Architecture (SCA) model in our previous work and the enhanced service publishing framework by OASIS, we present behavior based service selection technique. By plugging our selection algorithm to IBM WebSphere Process Server, which is based on the SCA, the web services binding can be automatically completed in the running time. BPEL has been used to describe service composition.**

Index Terms **-Service selection, dynamic behavior matching, Service Component Architecture, BPEL.**

I. INTRODUCTION

Web service composition is a distributed model to construct new web services on top of existing primitive or other composite services. This model provides us a framework that services can be designed locally and can be composed in the running time.

Usually, web services are first registered and published using UDDI, then discovered and bound by the application programs, finally a service composition is formed. UDDI plays a key role in this process. However, UDDI provides poor search facilities: it allows only a keyword based search of web services. Such kind of the keyword search method can not capture the underlying semantics of web services, thus can not protect us from obtaining "bad" services since UDDI registry contains checked and unchecked services. To address this issue, at least three approaches are proposed. Ankolekar et al.[1] created DAML-S, a web service description for semantic web, to narrow the selected services; Based on the same description, Paolucci et al.[9] developed algorithms for semantic matching of web services capabilities; Rodíguez et al.[10] proposed an approach to compute semantic similarities between services. These methods are basically to enhance the selection, but are still keyword based.

In this paper, we present a behavior based selection method to select services for service composition. The method is based on the formalized Service Component Architecture (SCA) model in our previous work[3] and the service publishing platform proposed by OASIS WSBPEL TC[1]. 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 BPEL, Java, C++, etc. The behavior used for the selection is the dynamic behavior of service component. OASIS WSBPEL TC can publish BPEL process in a UDDI registry. It is a good platform for web services participating in longer conversations to describe the behavior of the services in terms of dependencies, either logical or temporal, among exchanged messages.

As one can see BPEL is involved in both aspects, thus this paper will use BPEL to describe service composition. BPEL, standing for Business Process Execution Language, is an XML language that supports service composition and can be used to describe executable business process behaviors. BPEL offers services connectivity and is easy to be implemented. By plugging our selection algorithm to IBM WebSphere Process Server (Process Server) V6.0, which is based on the Service Component Architecture (SCA), the web services binding can be automatically completed in the running time. Hence, this method can offer us at least two advantages:

- the checked services can be selected based on their behaviors;
- the services can be automatically bound in the running time.

The paper is organized as follows. Section II is a general description of behavior based service selection process. Section III briefly describes formalized component model SCA. Section IV integrates BEPL with service component. Section V shows how to check the compatibility of two components. Section VI gives an example to illustrate our method. Section VII describes tool supported implementation of our technique. Section VIII is a discussion of the related work. Section IX concludes the paper.

II. THE DESCRIPTION OF THE SOLUTION

This section gives a general description of behavior based service selection technique.

Web service composition can be formed in the following

doi:10.4304/jsw.7.9.1950-1959

steps: web service providers publish their web service and invocation interfaces with WSDL and register the web services to UDDI; application programs then selects the needed web services and send the requests via invocation interfaces; finally a service composition is formed. The follows are the steps for the behavior based service selection.

STEP 1: Service Publishing. Based on the platform proposed by OASIS WSBPEL TC, we can publish service process to the UDDI registry. For the example, take a look at the following code describing part of the Deliver Agent service.

```
<process name="Delivery"
  targetNamespace="http://example.com/deliveragent"
  xmlns="http://schemas.../ws/.../business-process/"
  xmlns:taw="http://example.com/deliveragent/wsdl"
  abstractProcess="yes">
  ......
</process>
```

This service process can be published as a separate UDDI tModel, which is also named with the BPEL process. The corresponding UDDI tModels is in the following:

```
<tModel tModelKey="uuid:b1..." >
  <name>Delivery</name>
  <overviewDoc>
    <overviewURL>http://.../deliver.bpel<overviewURL>
  <overviewDoc>
  <categoryBag>
    <keyedReference
      tModelKey="uuid:d01987d1-...-9be2-2a66eb99d824"
      keyName="uddi-org:xml:namespace"
      keyValue="http://example.com/deliveragent" />
      ......
   </categoryBag>
</tModel>
```

Hence the service can be searched by service requirers.

STEP 2: Representation of Service Behavior. If we regard the service as a component [11], then the service behavior is the behavior of component. We use SCA as our component model, and the BPEL process is then transformed to a SCA model. The partner link, the port type, and the operation in BPEL form a port in the component, hence the composition between services is through ports of components. In our previous work [3], the dynamic behavior of component is described by port activities.

STEP 3: Checking the compatibility of service behavior.

The service selection is based on checking the compatibility of two components. For two given services A_s and B_s, assume that their corresponding components are A_c and B_c, respectively. If the behaviors of A_c and B_c are compatible, then services A_s and B_s can be composed.

Note: In the service selection, besides checking the dynamic service behavior, we may need to check the static signature of two components and consider other QoS matrices as well.

III. COMPONENT MODEL FOR SCA

In this section, we briefly describe our previous work that formalizes SCA assembly model. For the details, please see [3].

A. Service Component Model

In SCA, component provides and consumes services via ports. 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 support-ed by the port in SCA: synchronous communication and asynchronous communication. The formal definition of the port is given as follows.

Definition 3.1 (Port): A port p is a tuple (M, t, c), where M is a finite set of methods in p, 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 formed as op($\overline{Tx}$; $\overline{Ty}$), where op is the method name, $\overline{Tx}$ and $\overline{Ty}$ 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. For the sake of simplicity, we use •p to denote the synchronous port, and ♦p to denote the asynchronous port.

Definition 3.2 (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, $W \subseteq TP \times \cup_{C\in G}(C.P_p \cup C.P_r)$ is the port relation that is non-reflexive, where $TP = P_p \cup P_r \cup \cup_{C\in G} C.P_r$, $C.P_p$ and $C.P_r$ denote the provided and required port sets of the sub component C respectively.

The syntax of the component dynamic behavior 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)
| BBE (Basic behavior expression)
| BE ; BE (Sequence)
| BE$\lhd$ b$\rhd$ BE (Condition)
| b *BE (Loop)
| BE$\sqcap$ BE (Non-determinism)
| BE$||$BE (Parallel)
| BE[p_1/p_2] (Renaming)
| BE[$p_1 \rightarrow p_2$] (Wiring)

Here BBE represents the basic activities of skip that does nothing, and the assign activity p.x=e that assigns the value of expression e to the variable x of p. b represents the

Boolean expressions, 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. 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.

$$BE[p_1 \rightarrow p_2] = \begin{cases} p_2 \bullet_m p_1 & BE = p \bullet_m \wedge p = p_2 \\ p_1 \circ_m p_2 & BE = p \circ_m \wedge p = p_1 \\ p_2 \blacklozenge_m p_1 & BE = p \blacklozenge_m \wedge p = p_2 \\ p_1 \lozenge_m p_2 & BE = p \lozenge_m \wedge p = p_1 \\ BE_1[p_1 \rightarrow p_2]; BE_2[p_1 \rightarrow p_2] & BE = BE_1; BE_2 \\ BE_1[p_1 \rightarrow p_2] \triangleleft b \triangleright BE_2[p_1 \rightarrow p_2] & BE = BE_1 \triangleleft b \triangleright BE_2 \\ b * BE_1[p_1 \rightarrow p_2] & BE = b * BE_1 \\ BE_1[p_1 \rightarrow p_2] \sqcap BE_2[p_1 \rightarrow p_2] & BE = BE_1 \sqcap BE_2 \\ BE_1[p_1 \rightarrow p_2] \| BE_2[p_1 \rightarrow p_2] & BE = BE_1 \| BE_2 \\ BE & BE = p_1 \odot p_2 \end{cases}$$

B. Operational Semantics

We use the classical Labeled 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 $\langle BE, \varepsilon \rangle$, where BE is the behavior expression and E is the global asynchronous activity queue, which is unbounded. We use $\varepsilon ^\frown a$ to represent the resulted queue after adding activity a to the end of ε, $\varepsilon \backslash a$ to represent the resulted queue after removing the first activity a in ε, and $a \in \varepsilon$ to represent that the activity a is contained in the queue ε. In the following definition, to make the expressions simpler, we omit the asynchronous message queue ε from $\langle BE, \varepsilon \rangle$ if no conflict exists.

The basic synchronous semi-message activity do not have the complete communicating partner information, which will be achieved in the future. We assume that the basic synchronous semi-message activity will always complete. After getting the complete port information, the basic synchronous message activity can complete after synchronization.

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

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

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.

$$\frac{}{p \blacklozenge_m \xrightarrow{p \blacklozenge_m} stop}$$

$$\frac{}{\langle p_1 \blacklozenge_m p_2, \varepsilon \rangle \xrightarrow{p_1 \blacklozenge_m p_2} \langle stop, \varepsilon ^\frown p_1 \blacklozenge_m p_2 \rangle}$$

$$\frac{}{p \lozenge_m \xrightarrow{p \lozenge_m} stop}$$

$$\frac{p_1 \blacklozenge_m p_2 \in \varepsilon}{\langle p_1 \lozenge_m p_2, \varepsilon \rangle \xrightarrow{p_1 \lozenge_m p_2} \langle stop, \varepsilon \backslash p_1 \blacklozenge_m p_2 \rangle}$$

There are some rules for the sequential behavior composition, choice expression, loop expression, non-determinism expression, and parallel expression. We omit the details here.

The **stop** represents the termination of the dynamic behavior, and it cannot be engaged in any action. The execution of skip activity always terminates successfully, leaving everything unchanged. For the execution of assign, we will add the variable assignment to the configuration, denoted as $\langle BE, \varepsilon\{p.x \mapsto e\} \rangle$.

C. Component Composition

A composite contains assemblies of service components. 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 a 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 resulted composite by promoting the remaining non-wired ports.

At the signature level, two component $Com_i = (P_p^i, P_r^i, G_i, W_i)$ ($i \in \{1, 2\}$) are composable if they do not provide the same port, that is, $P_{p1} \cap P_{p2} = \varnothing$. We define the automatically related required port set as follows:

$M_p(Com_1, Com_2) = (P_{r1}\ P_{p2}) \cup (P_{r2}\ P_{p1})$,

where $P_r\ P_p = \{p \mid p \in P_r \wedge (p.M, provided, p.c) \in P_p\}$.

The automatically generated wire set can be defined as follows:

$$M_w(Com_1, Com_2) = \{(p_1, p_2) \mid p_1 \in M_p(Com_1, Com_2) \wedge p_2 = (p_1.M, provided, p_1.c)\}.$$

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

$$BE[W] = \begin{cases} BE & W = \varnothing \\ BE[p_1 \rightarrow p_2][W \backslash \{(p_1, p_2)\}] & (p_1, p_2) \in W \end{cases}$$

Definition 3.3 (Automatic Composition): Given two composable components $Com_i = (P_{pi}, P_r^i, G_i, W_i)$ and their dynamic behavior 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) $P_p = (P_{p1} \cup P_{p2}) \setminus \{(p.M, provided, p.c) \mid p \in M_p\}$;

(2) $P_r = (P_r^1 \cup P_r^2) \setminus M_p$;

(3) $G = G_1 \cup G_2$;

(4) $W = M_w \cup \{(p, p) \mid p \in P_p \cup P_r\}$;

(5) $BE = (BE_1 \,||\, BE_2)[M_w]$, where M_p and M_w are the abbreviations for $M_p(Com_1, Com_2)$ and $M_w(Com_1, Com_2)$ respectively.

Given the composite Com = (P_p, P_r, G, W), and the dynamic behavior expressions of the sub components, the dynamic behavior expression BE of Com can be calculated as follows: $BE = || \{BE_C \mid C \in G\}[W_w]$, where $|| B = B_1 || \ldots || B_n$ if $B = \{B_1, \ldots, B_n\}$, $W_w = \{(p_1, p_2) \mid (p_1, p_2) \in W \wedge p_1.t \neq p_2.t\}$.

Definition 3.4: (Compatibility) Given two components $Com_1$and Com_2, and their dynamic behaviour expressions $BE_1$and BE_2, $Com_1$and $Com_2$are compatible if the following conditions hold: (1) $Com_1$and Com_2 are composable; (2) there exists an action trace tr such that $BE \xrightarrow{tr} stop$ and the asynchronous message queue E is empty, where BE is the dynamic behavior of $Com_1 \oplus Com_2$.

IV. FROM BPEL TO SCA

BPEL process is viewed as a series of activities which can be composed into complex activities by data flow and control flow. Here are some basic flows:

Control flow: ⟨sequence⟩, ⟨switch⟩, ⟨while⟩, ⟨ flow ⟩, ⟨pick⟩. Data flow: ⟨invoke⟩, ⟨receive⟩, ⟨reply⟩, ⟨assign⟩.

Since composition in BPEL is characterized by the partner link, the port type, and the operation involved in the two communicating partners. These information forms the business functionalities offered by the process. We may use Port to represent all these information.

By mapping control flows and data flows to port activities, we may use port activities to describe interface behavior of the service component. The follows are the mappings for the basic control flows and data flows. Let BE be an expression of activities and p be a port, then we have the following rules.

Rule 1: assign:

⟨assign⟩ ⟨ copy ⟩ ⟨from variable = ”exp”/⟩ ⟨ to variable = ”x” /⟩ ⟨ /copy⟩ ⟨ /assign⟩ $\Longrightarrow$ p.x = exp.

Rule 2: sequence:

⟨sequence. . .⟩ ⟨act_1⟩ ⟨act_2⟩⟨/sequence⟩$\Longrightarrow$ BE_1; BE_2, where $BE_1$and $BE_2$are the behavior expressions derived from $act_1$and act_2, respectively.

Rule 3: switch:

⟨switch⟩ ⟨case condition=”...” ⟩ ⟨act_1⟩ ⟨/act_1⟩⟨/case⟩ ⟨otherwise⟩ ⟨act_2⟩ ⟨/act_2⟩ ⟨/otherwise⟩⟨/switch⟩$\Longrightarrow$ $BE_1 \lhd b \rhd BE_2$, where b = condition = ”...”, and $BE_1$and BE_2 are the behavior expressions derived from act_1 and act_2, respectively.

Rule 4: while:

⟨while condition = ”cond-exp” ⟩ ⟨act_1⟩ ⟨/act_1⟩ ⟨/while⟩ $\Longrightarrow$b*BE, where $BE_1$is a behavior expression derived from act_1, and b = condition=”cond-exp”.

Rule 5: flow:

⟨flow. . . ⟩ ⟨act_1⟩ ⟨/act_1⟩ ⟨act_2⟩ ⟨/act_2⟩ ⟨/flow⟩$\Longrightarrow$ $BE_1 \,||\, BE_2$, where BE_1 and BE_2 are the behavior expressions derived from $act_1$and act_2, respectively.

Rule 6: receive:

⟨receive partnerLink = ”...” portT ype = ”...”
operation = ”SelectInsurance”
variable = ”InsuranceRequest”
createInstance = ”yes”/⟩
$\Longrightarrow p \circ_m$,
where p = (M, t, c),
M={SelectInsurance(T InsuranceRequest)}
t=required,
c=synchronous,
and m contains the data information.

Rule 7: reply:

⟨reply partnerLink = ”...” portT ype = ”...”
operation = ”SelectInsurance”
variable = ”InsurSelResp”/⟩
$\Longrightarrow p \bullet_m$,
where p = (M, t, c),
M={SelectInsurance(; T InsurSelResp)}
t= provided,
c= synchronous,
and m contains the data information.

Rule 8: (Asynchronous)invoke:

⟨invoke partnerLink = ”...” portT ype = ”...”
operation = ”CompInsurP rem”
inputV ariable = ”InsuranceRequest”/⟩
$\Longrightarrow p \diamond_m$,
where p = (M, t, c),
M={CompInsurP rem(T InsuranceRequest)}
t =required,
c=asynchronous,
and m contains the data information.

Rule 9: (Synchronous)invoke:

⟨invoke partnerLink = ”...” portT ype = ”...”
operation = ”CompInsurP rem”
inputV ariable = ”InsuranceRequest”
outputV ariable = ”InsuranceAResposne”/⟩
$\Longrightarrow p \bullet_m$,
where p = (M, t, c),
M= {CompInsurP rem(T InsuranceRequest; T InsuranceAResposne),
t=provided,
c=synchronous.
and m will contain the data information.

Rule 10: pick:

⟨pick... ⟩
  ⟨onM essage...variable = ”m_1” ⟩
    ⟨...act_1... ⟩
  ⟨/onM essage >
  ⟨onM essage...variable = ”m_2” ⟩
    ⟨...act_2... ⟩
  ⟨/onM essage⟩
⟨/pick⟩

$\Longrightarrow$ $(p \circ_{m1}; BE_1) \sqcap (p \circ_{m2}; BE_2)$, where BE_1 and BE_2 are the behavior expressions derived from $act_1$and act_2, respectively. onMessage can be regarded as a receive activity and is sequentially combined with the corresponding BE_i. If message m_1 is received, then BE_1 is performed and BE_2 is not performed. If message m_2 is received, then BE_2 is performed and BE_1 is not performed. In the case that both messages are received almost simultaneously, the choice of the behavior depends on the implementation of BPEL.

V. SERVICE SELECTION

Our service selection process is based on checking the static signature and dynamic behavior of the service component.

STEP 1: Static Signature Checking.

In SCA specification, there are some constraints on the static signature definition of composite, such as the wire compatibility. Consistency represents that the component is well statically composed.

A component Com = (P_p, P_r, G, W) is consistent (denoted by Cons(Com)) if the following conditions hold:

- The communication types of two ports in each port relation must be equal, which is formalized as follows:
  $$\forall (p_1, p_2) \in W \bullet (p_1.c = p_2.c).$$
- For each port p in the component, the types of the ports that are related to p must be equal, which can be formalized as follows:
  $$\forall (p, p_1), (p, p_2) \in W \bullet (p_1.t = p_2.t).$$
- The port types of the promote must be same, and those of the wire must be different, which can be formalized as follows:
  $$\forall (p_1, p_2) \in W \bullet ((p_1 \in P_p \cup P_r \Rightarrow p_1.t = p_2.t) \wedge (p_1 \notin P_p \cup P_r \Rightarrow p_1.t \neq p_2.t))$$
- For each promoting port, the methods in a promoting provided port must be contained by the methods provided by the promoted provided port of one sub component, and the methods in a promoting required port must contain all the methods required by the promoted required ports of some sub components, which can be formalized as follows:
  $$\forall (p, p_1), (p, p_2) \in W \bullet (p \in P_p \Rightarrow (p_1 = p_2 \wedge p.M \subseteq p_1.M)) \wedge$$
  $$\forall p_2 \in P_r \bullet \cup_{(p2,pi) \in W} p_i.M \subseteq p_2.M$$
- For each wire, the methods of the required port must be contained by the related provided ports, which can be formalized as follows:
  $$\forall (p_1, p_2) \in W \bullet (p_1 \notin P_p \cup P_r \Rightarrow p_1.M \subseteq p_2.M).$$
- All the sub components are consistent, that is,
  $$\forall C \in G \bullet Cons(C).$$

STEP 2: Dynamic Behavior Checking.

Based on the Definition 3.4, we can check if two component behaviors are compatible. An algorithm is given in the following to check the compatibility. Let Com_1 and Com_2 be two components and their dynamic behavior expressions are BE_1 and BE_2, respectively. For the simplicity, in the algorithm, we consider the case that both Com_1 and Com_2 have just one branch. If Compatibility = True, then these two component behaviors are compatible.

Algorithm Check_The_Compatibility_Of_Components
Inputs: Com_1, Com_2;
Outputs: Compatibility {True, False};
Begin

Let $M_w = \{p_h \rightarrow p_k\}$; $E = \varnothing$; i= 1; j= 1;
// For the simplicity, we may assume
$BE_1 = x_1 \odot x_2 \odot \ldots x_{n1} \odot$;
$BE_2 = y_1 \odot y_2 \odot \ldots y_{n2} \odot$;
while ($i \leq n_1$)
  if $x_i = p_k$ and $\odot = \bullet_m$
    $BE_1[M_w] =_p p_k \bullet_m p_h; x_i+1 \odot \ldots x_n1 \odot [M_w]$;
    $//p_k \bullet_m p_h \xrightarrow{p_k \bullet_m ph}$ stop
    $BE_1[M_w] = stop; x_i+1 \odot \ldots x_n1 \odot [M_w]$
      $= x_{i+1} \odot \ldots x_{n1} \odot [M_w]$;
  if $x_i = p_k$ and $\odot = \blacklozenge_m$
    $BE_1[M_w] = p_k \blacklozenge m^p h; x_{i+1} \odot \ldots x_{n1} \odot [M_w]$;
    $// \langle p_k \blacklozenge_m p_h, \varepsilon \rangle \xrightarrow{p_k \blacklozenge_m p_h} \langle stop, \varepsilon \frown p_k \blacklozenge_m p_h \rangle$
    $BE_1[M_w] = stop; x_{i+1} \odot \ldots x_{n1} \odot [M_w]$
      $= x_{i+1} \odot \ldots x_{n1} \odot [M_w]$;
  if $x_i = p_h$ and $\odot = \circ_m$
    $BE_1[M_w] = p_h \circ_m p_k; x_{i+1} \odot \ldots x_n \odot [M_w]$;
    $// p_h \bullet_m p_k \xrightarrow{p_h \bullet_m p_k} stop$
    $BE_1[M_w] = stop; x_{i+1} \odot \ldots x_{n1} \odot [M_w]$
      $= x_{i+1} \odot \ldots x_{n1} \odot [M_w]$;
  if $x_i = p_k$ and $\odot = \lozenge_m$
    $BE_1[M_w] = p_h \lozenge_m p_k; x_{i+1} \odot \ldots x_{n1} \odot [M_w]$;
    $// \langle p_h \lozenge_m p_h, \varepsilon \rangle \xrightarrow{p_h \lozenge_m p_h} \langle stop, \varepsilon \frown p_k \blacklozenge_m p_h \rangle$
    $BE_1[M_w] = stop; x_{i+1} \odot \ldots x_{n1} \odot [M_w]$
      $= x_{i+1} \odot \ldots x_{n1} \odot [M_w]$;
  i + +;
end while
while ($j \leq n_2$)
  (repeat the above, change x to y and $n_1$to n_2.)
  j + +;
end while
if $BE_1[M_w] // BE_2[M_w]$ = stop and $E = \varnothing$, then
  Compatibility = True;

else
  Compatibility = False;

VI. AN EXAMPLE: ONLINE SHOP

To illustrate our method, we present an online shop as our example process [4].

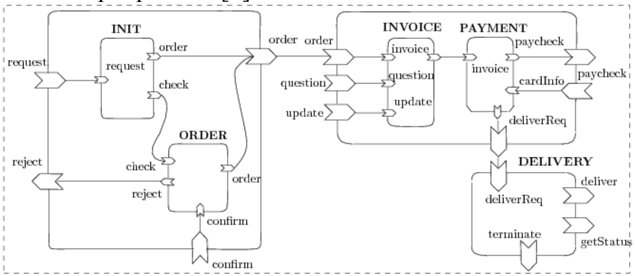

Fig 1. SCA of online shop

The process is structured into two concurrent activities: customers initial choice and order processing. But, 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 he got the confirmation. Assuming the customer has ordered a product, he either 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 to a delivery service provider. We assume that the process has no more interaction with the environment except asking for delivery service. The SCA of online shop is displayed in Fig. 1.

In the figure, ports

p^{11}request= ({request($\overline{\mathrm{Re}q}$, $\overline{\mathrm{Re}sp}$)},
provided, synchronous),

p^{11}check= ({check($\overline{check\,\mathrm{Re}q}$, $\overline{check\,\mathrm{Re}sp}$)},
required, synchronous),

p^{11}order= ({order($\overline{order}$)},
required, asynchronous),

form atomic-component C^{11}= INIT with dynamic behavior expression BE^{11}= p^{11}request◦; (p^{11}check •$\triangleleft$ $b_1$$\triangleright$ p^{11} order), where

b_1='check is required'. Ports

p^{12}check= ({check($\overline{check\,\mathrm{Re}q}$, $\overline{check\,\mathrm{Re}sp}$)},
provided, synchronous),

p^{12}reject= ({reject($\overline{orderData}$, $\overline{reject\,\mathrm{Re}sp}$)},
provided, synchronous),

p^{12}order= ({order($\overline{order}$)},
required, asynchronous),

p^{12}confirm= ({conf irm($\overline{orderData}$, $\overline{confirm\,\mathrm{Re}sp}$)},
provided, synchronous),

form atomic-component C^{12}= ORDER with dynamic behavior expression BE^{12}=p^{12}check◦; (p^{12}reject◦ $\triangleleft$ b_2 $\triangleright$ p^{12}confirm◦); p^{12}order, where b_2='the order is cancelled'.

Ports

p^{21}order= ({order($\overline{order}$)},
provided, asynchronous),

p^{21}question= ({question($\overline{question}$, $\overline{answer}$)},
provided, synchronous),

p^{21}update= ({update($\overline{update\,\mathrm{Re}q}$, $\overline{update}$)},
provided, synchronous),

p^{21}invoice= ({invoice($\overline{invoice}$, $\overline{back}$)},
required, synchronous),

form atomic-component C^{21}= INVOICE with dynamic behavior expression BE21= p^{21}order◊;(p^{21}question$\triangleleft$ $b_3$$\triangleright$ p^{21}update); p^{21}invoice•, where b_3='more questions are needed'.

Ports

p^{22}invoice= ({invoice($\overline{invoice}$, $\overline{back}$)},
provided, synchronous),

p^{22}paycheck=({paycheck($\overline{check\,\mathrm{Re}q}$, $\overline{check\,\mathrm{Re}sp}$)},
required, synchronous),

p^{22}deliverReq= ({deliverReq($\overline{invoice}$, $\overline{\mathrm{Re}sp}$)},
required, synchronous)

p^{22}cardInf o= ({enterCardInf o($\overline{cardInfo}$ o)},
provided, asynchronous)

p^{22}orderStatus= ({orderStatus($\overline{orderStatus}$)},
required, asynchronous)

form atomic-component C^{22}= PAYMENT with dynamic behavior expression B^{22}= p^{22}invoice◦;b_4∗ (p^{22}cardInfo ◊ ;p^{22}paycheck•); p^{22}deliverReq•;p^{22}orderStatus , where b_4='the payment checking is failed'. Assume that b_4( ) has the maximum three loops depending on the user input.

Ports

p^{3}deliverReq= ({deliverReq($\overline{invoice}$, $\overline{\mathrm{Re}sp}$)},
provided, synchronous),

p^{3}deliver= ({deliverReq($\overline{checkedInvoice}$)},
required, asynchronous),

p^{3}getStatus= ({getStatus($\overline{status\,\mathrm{Re}q}$, $\overline{status\,\mathrm{Re}sp}$)},
required, synchronous),

p^{3}terminate= ({terminate($\overline{ter\min ate}$)},
required, asynchronous),

form atomic-component C^{3}= DELIVERY with dynamic behavior expression B^{3}= p^{3}deliverReq◦; p^{3}deliver; p^{3}getStatus•; p^{3}terminate♦.

Service Compability Checking At Signature Level.
We omit this step and focus on the behavior checking.

Service Compability Checking At Behaviour Level.

Let C^1 be the automatic composition of components C^{11} and C^{12}, then $C^1 = C^{11} \oplus C^{12} = (P_p1_{,\,P}r^1, G_1, W_1)$, where

$P_p^1 = P_p^{11} \cup P^{12} \setminus \{(p.M, provided, p.c) | p \in M_p)\}$

$= \{p^{11}request, p^{12}reject\}$,

$P_r^1 = P_r^{11} \cup P_r^{12} \setminus M_p(C^{11}, C^{12})$

$= \{p^{12}confirm, p^{11}{}_{order}, p^{12}{}_{order}, p^{11}{}_{check}\} \setminus \{p^{11}{}_{check}\}$

$= \{p^{12}{}_{confirm}, p^{11}{}_{order}, p^{12}{}_{order}\}$,

$G_1 = \{\}$,

$W_1 = M_w \cup \{(p, p) | p \in P_p^1 \cup P_r^1\}$

$= \{(p^{11}{}_{check}, p^{12}{}_{check})\} \cup \{(p, p) | p \in P_p^1 \cup P_r^1\}$

$= \{(p^{11}{}_{check}, p^{12}{}_{check})\} \cup \{(p^{11}{}_{request}, p^{11}{}_{request})$,

$(p^{11}{}_{order}, p^{11}{}_{order}), (p^{12}{}_{reject}, p^{12}{}_{reject}), (p^{12}{}_{confirm}, p^{12}{}_{confirm})$,

$(p^{12}{}_{order}, p^{12}{}_{order})\}$,

$BE^1 = (BE^{11} \,|\,|\, BE^{12})[\{(p^{12}check \to p^{11}check)\}]$.

Let C^2 be the automatic composition of components C^{21} and C^{22}, then $C^2 = C^{21} \oplus C^{22} = (P_p2_{,\,P}r^2, G_2, W_2)$, where

$P_p^2 = P_p^{21} \cup P_p^{22} \setminus \{(p.M, provided, p.c) | p \in M_p)\}$

$= \{p^{21}order, p^{22}deliver\}$,

$P_r^2 = P_r^{21} \cup P_r^{22}{}_{\setminus M}p(C^{21}, C^{22})$

$= \{p^{21}{}_{question}, p^{21}{}_{update}, p^{22}{}_{paycheck}, p^{21}{}_{invoice}\} \setminus \{p^{21}{}_{invoice}\}$

$= \{p^{21}{}_{question}, p^{21}{}_{update}, p^{22}{}_{paycheck}\}$,

$G_2 = \{\}$,

$W_2 = M_w \cup \{(p, p) | p \in P_p^2 \cup P_r^2\}$

$= \{(p^{21}{}_{invoice}, p^{22}{}_{invoice})\} \cup \{(p, p) | p \in P_p^2 \cup P_r^2\}$

$= \{(p^{21}{}_{invoice}, p^{22}{}_{invoice})\} \cup \{(p^{21}{}_{order}, p^{21}{}_{order})$,

$(p^{21}{}_{question}, p^{21}{}_{question}), (p^{21}{}_{update}, p^{21}{}_{update})$;

$(p^{22}{}_{paycheck}, p^{22}{}_{paycheck}), (p^{22}{}_{deliver}, p^{22}{}_{deliver})\}$,

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

Assume that port relation set is

$W = W_1 \cup W_2 \cup \{(p^1{}_{order}, p^2{}_{order})\} \cup \{(p^2{}_{deliverReq}, p^3{}_{deliverReq})\}$, then a semantic model of dynamic behavior of online shop, denoted as SCA_{online}, would be

$SCA_{online} = C^1 \oplus_W C^2 \oplus_W C^3 = (P_p, P_r, G, W)$, where

$P_p = \{p_1 | (!p_1, !p_2) \in W\}$,

$P_r = \{p_1 | (?p_1, ?p_2) \in W\}$,

$G = \{G_1, G_2\}$,

$W_c = W$,

$BE = (BE^1 \,|\,|\, BE^2 \,|\,|\, BE^3)[W_w] \langle W_p \rangle$,

Where $W_w = \{(p_1, p_2) | (p_1, p_2) \in W \wedge p_1.t \neq p_2.t\}$, and $W_p = \{(p_1, p_2) | (p_1, p_2) \in W \wedge p_1.t = p_2.t\}$.

Our interest is located in computing the dynamic behavior BE. We are not going to compute all the behaviors, instead, we only compute just one behavior

request → check → conf irm → order → question → invoice → paycheck → deliver

to illustrate our method. Since we have

$SCA_{online} = C^1 \oplus C^2 \oplus C^3$.

thus, the dynamic behavior becomes

$BE = (BE^1 \,|\,|\, BE^2 \,|\,|\, BE^3)[M_w]$

$= BE^1[M_w] \,|\,|\, BE^2[M_w] \,|\,|\, BE^3[M_w]$,

Where

$M_w = \{p^{12}{}_{check} \to p^{11}{}_{check}, p^{22}{}_{invoice} \to p^{21}{}_{invoice}, p^{21}{}_{order} \to p^{12}{}_{order}, p^3{}_{deliverReq} \to p^{22}{}_{deliverReq}$,

$I = User - Inputs\}$.

Now we first compute BE^1:

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

where

$BE^{11}[M_w] = (p^{11}{}_{request}\circ; (p^{11}{}_{order}\blacklozenge \triangleleft b \triangleright p^{11}{}_{check}\bullet))[M_w]$

$\Longrightarrow p^{11}{}_{request} \circ [\{I, \ldots\}]; (p^{11}{}_{order}\blacklozenge \triangleleft b \triangleright p^{11}{}_{check}\bullet)$

$[\{p^{12}check \to p^{11}{}_{check}, \ldots\}]$

$\Longrightarrow stop; (p^{11}{}_{order}\blacklozenge \triangleleft b \triangleright p^{11}{}_{check}\bullet)$

$[\{p^{12}{}_{check} \to p^{21}{}_{check}, \ldots\}]$

$\Longrightarrow (p^{11}{}_{order}\blacklozenge [\{\ldots\}] \triangleleft b \triangleright p^{11}{}_{check}\bullet)$

$[\{p^{12}{}_{check} \to p^{11}{}_{check}, \ldots\}]$

$\Longrightarrow p^{11}{}_{check} \bullet [\{p^{12}{}_{check} \to p^{11}{}_{check}, \ldots\}]$

$\Longrightarrow p^{11}{}_{check} \bullet p^{12}{}_{check}$

and

$BE^{12}[M_w]$

$= (p^{12}{}_{check}\circ; (p^{11}{}_{reject} \bullet \triangleleft b \triangleright p^{11}{}_{conf\,irm}\circ); p^{12}{}_{order}\blacklozenge)$

$[\{p^{12}{}_{check} \to p^{11}{}_{check}, I, \ldots\}]$

$\Longrightarrow p^{12}{}_{check} \circ [\{p^{12}{}_{check} \to p^{11}{}_{check} \ldots\}]$;

$(p^{11}r_{eject} \bullet \triangleleft b \triangleright p^{11}{}_{confirm}\circ)[\{I, \ldots\}]$;

$p^{12}{}_{order}\blacklozenge)[\{p^{21}{}_{order} \to p^{12}{}_{order} \ldots\}]$

$\Longrightarrow p^{12}{}_{check} \circ p^{11}{}_{check}$;

$p^{11}{}_{confirm}\bullet [\{I, \ldots\}]$; // choose confirm

$p^{12}{}_{order}\blacklozenge [\{p^{21}{}_{order} \to p^{12}{}_{order}, \ldots\}]$.

Thus

$BE^1 \Longrightarrow (p^{11}{}_{check} \bullet p^{12}{}_{check}) \,|\,|$

$(p^{12}{}_{check} \circ p^{11}{}_{check}; p^{11}{}_{confirm} \bullet [\{I, \ldots\}]$;

$p^{12}{}_{order}[\{(p^{21}{}_{order} \to p^{12}{}_{order}, \ldots\}])$

$\Longrightarrow stop \,|\,|\, (stop; p^{11}{}_{confirm}\bullet [\{I, \ldots\}]$;

$p^{12}{}_{order}\blacklozenge [\{p^{21}order \to p^{12}order, \ldots\}])$

$\Longrightarrow p^{11}{}_{confirm} \bullet [\{I, \ldots\}]$;

$p^{12}{}_{order}\blacklozenge [\{p^{21}{}_{order} \to p^{12}{}_{order}, \ldots\}]$

$\Longrightarrow stop; p^{12}{}_{order}\blacklozenge [\{p^{21}{}_{order} \to p^{12}{}_{order}, \ldots\}]$

$\Longrightarrow p^{12}{}_{order}\blacklozenge [\{p^{21}{}_{order} \to p^{12}{}_{order}, \ldots\}]$

$\Longrightarrow p^{12}{}_{order}\blacklozenge p21order$.

Similar to BE^1, we can compute BE^2 as

$BE^2 = (BE^{21} \,|\,|\, BE^{22})[M_w]$

$= BE^{21}[M_w] \,|\,|\, BE^{22}[M_w]$,

where

$BE^{21}[M_w]$

$= (p^{21}{}_{order}\lozenge; (p^{21}{}_{question} \bullet \triangleleft b \triangleright p^{21}{}_{update}\bullet); p^{21}{}_{invoice}\bullet)[M_w]$

$\Longrightarrow p^{21}{}_{order}\lozenge [\{p^{21}{}_{order} \to p^{12}{}_{order}, \ldots\}]$;

$(p^{21}{}_{question} \bullet \triangleleft b \triangleright p^{21}{}_{update}\bullet)[\{I, \ldots\}]$;

$p^{21}{}_{invoice} \bullet [\{p^{22}{}_{invoice} \to p^{21}{}_{invoice}, \ldots\}]$

$\Longrightarrow p^{21}{}_{order}\lozenge p^{12}{}_{order}$;

$(p^{21}_{question}\bullet \triangleleft\, b \triangleright\, p^{21}_{update}\bullet)[\{I, \ldots\}];$

$p^{21}_{invoice}\bullet[\{p^{22}_{invoice} \rightarrow p^{21}_{invoice}, \ldots\}]$

$BE^{22}[M_w]= (p^{22}_{invoice}\circ; p^{22}_{paycheck}\bullet; p^{22}_{deliverReq}\blacklozenge)^{[}M_w]$

$\Longrightarrow p^{22}_{invoice}\circ[\{p^{22}_{invoice}\rightarrow p^{21}_{invoice}, \ldots\}];$

$p^{22}_{paycheck}\bullet[I, \ldots]; p^{22}_{deliverReq}\blacklozenge [I,\ldots]$

$\Longrightarrow p^{22}_{invoice}\circ p^{21}_{invoice};$

$p^{22}_{paycheck}\bullet[I, \ldots]; p^{22}_{deliverReq}\blacklozenge [I,\ldots]$

Thus,

$(BE^1)[M_w] \mid\mid (BE^2)[M_w]$

$\Longrightarrow p^{12}_{order}\blacklozenge p^{21}_{order} \parallel (BE^2)[M_w]$

$\Longrightarrow p^{12}_{order}\blacklozenge p^{21}_{order}\parallel$

$(p^{21}_{order}\lozenge p^{12}_{order};$

$(p^{21}_{question}\bullet \triangleleft\, b \triangleright\, p^{21}_{update}\bullet)[I, \ldots];$

$p^{21}_{invoice}\bullet[\{p^{22}_{invoice} \rightarrow p^{21}_{invoice}, \ldots\}] \parallel (BE^{22})[M_w]$

$\Longrightarrow stop \parallel Stop;$

$p^{21}_{question}\bullet[I, \ldots];$ //choose question

$p^{21}_{invoice}\bullet[\{p^{22}_{invoice}\rightarrow p^{21}_{invoice}, \ldots\}]$

$\parallel(BE^{22})[M_w]$

$\Longrightarrow stop; p^{21}_{invoice}\bullet p^{22}_{invoice} \parallel (BE^{22})[M_w]$

$\Longrightarrow p^{21}_{invoice}\bullet p^{22}_{invoice}\parallel p^{22}_{invoice}\circ p^{21}_{invoice};$

$p^{22}_{paycheck}\bullet[I, \ldots]; p^{22}_{deliverReq}\blacklozenge [I,\ldots]$

$\Longrightarrow stop \mid\mid stop;$

$p^{22}_{paycheck}\bullet[I, \ldots];$

$p^{22}_{deliverReq}\blacklozenge [I,\ldots]$

$\Longrightarrow stop \mid\mid op; stop; p^{22}_{deliverReq}[I, \ldots])$

$\Longrightarrow p^{22}_{deliverReq}[I, \ldots].$

Finally,

$BE = (BE^1)[M_w] \mid\mid (BE^2)[M_w] \mid\mid E^3)[M_w]$

$= p^{22}_{deliverReq}\blacklozenge [p^3_{deliverReq}\rightarrow p^{22}_{deliverReq}, \ldots] \mid\mid$

$(p^3_{deliverReq}\lozenge ; p^3_{deliver}; \blacklozenge p^3_{getStatus}\bullet;$

$P^3_{terminate}\blacklozenge)[p^3_{deliverReq}\rightarrow p^{22}_{deliverReq}, \ldots]$

$= p^{22}_{deliverReq}\blacklozenge p^3_{deliverReq}\parallel p^3_{deliverReq}\lozenge\ p^{22}_{deliverRe;}p^3_{deliver}\blacklozenge;$

$p^3_{getStatus}\bullet; P^3_{terminate}\blacklozenge$

$= stop \mid\mid stop; p^3_{deliver}\blacklozenge; p^3_{getStatus}\bullet; p^3_{terminate}\blacklozenge$

$= p^3_{deliver}\blacklozenge; p^3_{getStatus}\bullet; p^3_{terminate}\blacklozenge.$

Now the asynchronous message queue ε is empty and the last line is the behaviour of component DELIVERY. Hence, if we want to find a shipping service that is compatible with SCA_{online}, it is required to find a shipping service that is compatible with DELIVERY.

VII. TOOL SUPPORTED SERVICE SELECTION

IBM recently released IBM WebSphere Process Server (Process Server) V6.0, which is based on the Service Component Architecture (SCA). IBM WebSphere Integration Developer V6.0 is an Eclipse-based tool used for developing and assembling applications targeted for IBM WebSphere Process Server V6.0. We will use Integration Developer to develop automatic service binding for BPEL.

The following is the sketch of our developing process for service selection. We first build components, define interfaces for the components, import interfaces from existing services, and wire components together; then we implement the Process component with the BPEL editor; finally we deploy the application by mapping every SCA module to a complete J2EE application package. Currently the existing IBM software sets service binding manually. For example, to bind the delivery service, the software needs to select the Browser to choose the WSDL file in the local file system. After integrating with our selection method, the service binding can be automatically completed by the system after the service is selected. At this moment the service component is wired to the application module. As Figure 2 shows, service component Delivery is wired to createEshop module.

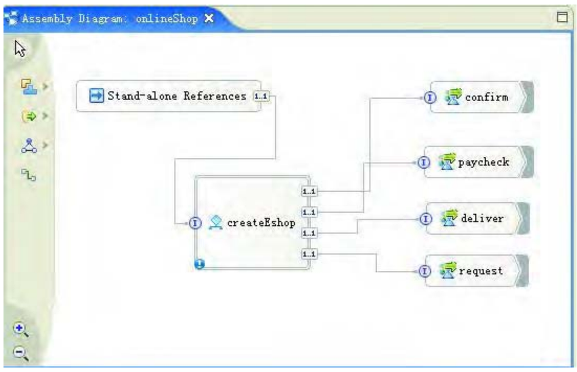

Fig 2. Service component is wired to application module for Online Shop.

The key point in the service binding is to assign the service end points to the partner links, so that the Endpoint URL (hostname, port, and path), Target service name, and Port name will be changed.

VIII. RELATED WORK

The ideas presented in this article are most closely related to the following research topics: service component modeling, service selection and integration of BPEL and service component.

Service Component Modeling

Because SCA is a new proposed specification, there are few existing work in formalization and verification for SCA. In [5], a small core formal language (SRML) 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. It addresses high-level design and supports a shift of emphasis from programming to business modeling. However, our method addresses low-level design in the sense that it provides an assembly model and binding mechanisms for service components and clients programed in specific languages, e.g. Java, C++, BPEL, or PHP. From this point of view, our method is more proper for the development in the real world.

Service Selection

A critical issue in automatic service binding is service selection.

Many methods have been developed such as: 1) querying a knowledge base of DAML-S service profiles[7], 2) semantic matching between advertisements and requests[9], 3) predefining carriers to create a single business process at design time and then using semantic technology to pick and instantiate the most appropriate process at runtime[6]. Our method to select required service is based on the signature and dynamic behavior of the service component. This can protect us from obtaining ”bad” services since UDDI registry contains checked and unchecked services. Thus our method can perform dynamic checking.

Integration of BPEL and Service Component

At least three ways are proposed to integrate BEPL and service component. One is based on Yang's work[11]. In [11], BPEL is converted into an equivalent object-oriented class. Any kind of service (composite or not) can be represented and stored as a service component class and can be used in the development of distributed applications. The second way is from IBM WebSphere Studio Integration Edition. It automatically creates a component file for BPEL process as well as for the invoked Web service. This file describes the interface and implementation of the process or Web service. When deployment code is generated for the BPEL process, the component files of the process and the service are wired together. The third way is from recently released IBM WebSphere Process Server (Process Server) V6.0, which is based on the new Service Component Architecture (SCA). Integration Developer V6.0 is used for developing and assembling applications. All these methods do not have a language independent dynamic behavior model to specify the interface behavior of the service component. In this paper, we adopt the third way as our blueprint, and enhance it by introducing dynamic behavior model.

IX. CONCLUSION

In this paper, we presented a behavior based service selection technique. This technique can protect us from obtaining ”bad” services. By integrating this technique with IBM WebSphere Process Server, the web services binding can be automatically completed in the running time. As the future work, we will further improve the service selection method for the composition of intelligent services, and strengthen the laws in the operational semantics so that they are powerful enough to do any reasoning such as calculating the minimal required interface between two components.

X. ACKNOWLEDGMENTS

This work is supported by NSF under Grant No. 90818013, No. 90718014 and Zhejiang NSF under Grant No.Z1090357.

REFERENCES

[1] A. Ankolekar et al., *DAML-S: Web Service Description for the Semantic Web*, In International Semantic Web Conference, Sardinia, Italy, 2002,pp.348-363.

[2] M. Beisiegel and et al, *Service Component Architecture Specification*. In http://www.osoa.org/display /Main/Home, 2007.

[3] Z. Ding, Z. Chen and J. Liu, *A Rigorous Model of Service Component Architecture*, In ENTCS, vol.207, pp.33-48, 2008.

[4] Z. Ding, M. Jiang, A. Kandel, *Port Based Reliability Computing For Service Composition*, In IEEE Transactions on Services Computing, 16 Mar. 2011, http: //doi. ieeecomputersociety. org/ 10.1109/TSC.2011.17.

[5] J. Fiadeiro, A. Lopes, L. Bocchi, *A Formal Approach to Service Component Architecture*, In Lecture Notes in Computer Science, vol.4184, pp.193-213, 2006.

[6] J. Lemcke, C. Drumm, *Semantic Business Automation*, In Proceedings of the 3rd European Semantic Web Conference's Industry Forum, Budva, Montenegro, 2006.

[7] D. J. Mandell, S. A. McIlraith, *Adapting BPEL4WS for the Semantic Web: The Bottom-up Approach to Web Service Interoperation*, In Proceedings of ISWC'03, Florida, 2003.

[8] T. Margaria and B. Steffen, *Service Engineering: Linking Business and IT. In Computer*, vol.39, no.10, pp.45-55, 2006.

[9] M. Paolucci, T. Kawmura, T. Payne and K. Sycara, *Semantic Matching of Web Services Capabilities*, In Proceedings of ISWC'02, Sardinia, Italy,2002.

[10] M. A. Rodríguez, M. J. Egenhofer, *Determining Semantic Similarity Among Entity Classes From Different Ontologies*, In IEEE Transactions on Knowledge and Data Engineering, vol.15, no.2, pp.442-456, 2003.

[11] J. Yang, *Web Service Componentization,* In Communication of ACM, vol.46, no.10, pp.35-40, 2003.

Zuohua Ding received his PhD (1996) in Mathematics (advisor: Professor Athanassios G. Kartsatos) and MS (1998) of Computer Science (advisor: Professor Abraham Kandel) all from University of South Florida, USA.

He is currently a professor and the director of Lab of Scientific Computing and Software Engineering, Zhejiang Sci-Tech University, Hangzhou, China, and he has been a research professor of National Institute for Systems Test and Productivity, USA since 2001. From 1998 to 2001, he was a senior software engineer in Advanced Fiber Communication, USA.

His research interests include software testing, program analysis, service computing and formal method, subjects on which he has authored and coauthored more than 70 papers.

Mingyue Jiang is an assistant professor of Center of Math Computing and Software Engineering, Zhejiang Sci-Tech University, Hangzhou, China.

Her research interests include program analysis and web engineering. So far, she has published over 10 papers

Jing Liu received her Ph.D in computer science from Shanghai University in 2005, and now she is a professor in Software Engineering Institute of East China Normal University.From 1996 to 2000, she was working in IBM Raleigh Lab and IBM Toronto Lab for software testing. From 2003 to 2004, she was working as a researcher in International Institute of Software Technology, United Nations University.

Her current research interests include software modeling, formal methods, and service computing, subjects on which she has authored and coauthored more than 65 papers.