Interface Theory based Formalization and Verification of Orchestration in BPEL4WS

Zhenbang Chen, Ji Wang, Wei Dong, Zhichang Qi
National Laboratory for Parallel and Distributed Processing
Changsha, China
{z.b.chen, jiwang, dongwei}@mail.edu.cn

ABSTRACT

BPEL4WS (BPEL) is a Web service composition language in service-oriented computing. Service orchestration can be specified by the executable process in BPEL. However, it lacks of a formal foundation for specification and verification of service-oriented systems. In this paper, a formalization is presented for service orchestration in BPEL, based on the interface theory of the Web services with transactions. The transformational approach is proposed to translate BPEL processes to protocol interfaces. With the formalization, verification method is presented to ensure the correctness of BPEL. Specially, model checking method is used to verify the protocol property. One may check the compatibility and refinement between the BPEL processes as well as verifying some critical properties. A complete case study is demonstrated to illustrate our approach.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications; D.2.12 [Software Engineering]: Interoperability—Interface definition languages; H.3.5 [Information Systems]: Online Information Services—Web-based services

General Terms
Design, Reliability, Theory, Algorithms

Keywords
Service composition, Web services, Web service interfaces, Formal specification, BPEL4WS, Interface theory, Verification

1. INTRODUCTION

Service composition is an important theme in service-oriented computing. Many languages such as WSFL [1], WSCI [2], XLANG [4] and BPEL [3] are emerging as Web services composition languages. It is desired to build formal foundations for service composition in order to understand the service-oriented software systems well.

In general, service composition can be categorized into service orchestration and choreography. Orchestration refers to an executable business process that may interact with both internal and external Web services. Orchestration describes how Web services can interact at the message level, including the business logic and execution order of the interactions. These interactions may span applications and/or organizations, and result in a long-lived, transactional process. With orchestration, the process is always controlled by the orchestration, while the choreography tracks the sequence of messages that may involve multiple parties and multiple sources. It is associated with the public message exchanges that occur between multiple Web services [5].

Recently, the propose of Web service interface theory makes the description of orchestration in Web services become more natural. As a formal foundation of the component-based design, Alfar and Henzinger [6] proposed a theory of interface automata to specify the component interfaces. Beyer et al. [7, 8] presented a Web service interface description language, which can describe the interfaces in three levels, i.e. signature, consistency and protocol levels. However, the transaction feature is not considered in the existing interface theories, while it is one of the essential features in distributed computing, such as Web service systems. Web service-based transactions differ from traditional transactions in that they execute over long periods, require commitments to the transaction to be "negotiated" at runtime, and isolation levels must be relaxed [9]. We extend the formalism of Web service interfaces proposed in [7] to describe transaction information in all three levels of signature, conversation and protocol.

The aim of the paper is to formalize and verify service orchestration by Web service interface theory. The concerned language is BPEL, which has been accepted as a mainstream Web service composition language. The contribution of this paper is four-fold: (1) We propose an interface theory for orchestration description at three abstract levels of signature, conversation and protocol. (2) We present the translation rules and algorithms to translate from BPEL to protocol interface. (3) We propose the verification method, process and methodology to ensuring the high confidence in orchestration description. (4) A travel agency Web service without/with long running transaction is demonstrated to illustrate our formalism and the verification approach.

The rest of this paper is organized as follows: section 2 presents the framework of the interface theory for Web services with transactions, including the signature interface, conversation interface and protocol interface; section 3 proposes the translation methods from BPEL to protocol interface and the translation algorithms that use the translation methods; section 4 proposes the verifica-
tion method and its methodology; section 5 gives a travel agency Web service system to illustrate the formalization and verification; section 6 highlights some related works and section 7 concludes the paper and discusses some issues of future research.

2. WEB SERVICE INTERFACE THEORY

The basic activity in Web services is method call. A Web service may support or invoke some methods, and a method call may return different values. So a method attached by a return value can suitably depict the basic action in Web service interface. Whether an action is successful or exceptional will also be described in interface description. If an exception action is invoked, the fault handling for the action should be taken, and the successful actions that have been invoked before should be compensated. Let $M$ be a finite set of web methods, $O$ be a finite set of outcomes and $dom(f)$ denote the domain of the function $f$. Signature interface is defined as follows.

**Definition 1** (Signature Interface, SI). A signature interface $I$ is a 4-tuple $(A, S, C, F)$, where

- $A \subseteq M \times O$ is a set of actions that can appear in $P$;
- $S : A \rightarrow 2^A$ is a partial function that assigns to an action $a$ a set of actions that can be invoked by $a$;
- $S_c : A \rightarrow 2^A$ is a partial function that assigns to an action $a$ a set of actions that can be invoked by the compensation for $a$;
- $S_F : A \rightarrow 2^A$ is a partial function that assigns to an action $a$ a set of actions that can be invoked by the fault handling for $a$;
- $dom(S_c) \cap dom(S_F) = \emptyset$, and $dom(S_c) \cup dom(S_F) = dom(S)$.

Signature interface describes the direct invocation relation of Web service interfaces. An action may have different types. An action $a \in A$ is a supported action if $S(a)$ is defined. A web method $m \in M$ is a supported method if there exists a supported action $a = (m, o)$. An action $a$ is a success action if $S_c(a)$ is defined. An action $a$ is an exception action if $S_F(a)$ is defined. An action $a$ is a required action if it can be invoked by a supported action or compensation or fault handling, which can be expressed by the formula defined as follows:

$$\text{required}(a') = (\exists a \in dom(S), a' \in S(a)) \lor$$
$$\text{required}(a') = (\exists a \in dom(S_c), a' \in S_c(a)) \lor$$
$$\text{required}(a') = (\exists a \in dom(S_F), a' \in S_F(a)).$$

Service registries often require service providers to publish solid description of interfaces. We use well-formed interfaces to ensure the integrity. A signature interface is well-formed if the following conditions hold: every required action whose method is a supported method is a supported action, and no exception action will be invoked in compensation or fault handling.

An action may invoke different action sets in different conditions. Signature interface can not describe this feature. Conversation interface is proposed to specify different cases for the invocation of an action. A conversation is a set of actions that are invoked together. Propositional formulas are used to represent different conversations.

**Definition 2** (Conversation Expression). The set of conversation expressions over an action set $A$ is given by the following grammar ($a \in A$):

$$\omega :: T | a | \omega_1 \sqcup \omega_2 | \omega_1 \sqcap \omega_2$$

$T$ is the propositional constant, which represents no action is needed to be invoked. Action $a$ represents a single action is needed to be invoked. The expression $\omega_1 \sqcup \omega_2$ represents that each conversation represented by $\omega_1$ or $\omega_2$ can be invoked. The expression $\omega_1 \sqcap \omega_2$ represents that one conversation must contain each conversation from $\omega_1$ and $\omega_2$. The set of all conversation expressions on action set $A$ is denoted by $\omega(A)$.

**Definition 3** (Conversation Interface, CI). A conversation interface $I$ is a 4-tuple $(A, \epsilon, E_c, E_f)$, where

- $A \subseteq M \times O$ is a set of actions that can appear in $I$;
- $\epsilon : A \rightarrow \omega(A)$ is a partial function that assigns to an action $a$ a conversation expression which represents all conversations that can be invoked by $a$;
- $E_c : A \rightarrow \omega(A)$ is a partial function that assigns to an action $a$ a compensation conversation expression which represents all conversations that can be invoked by the compensation for $a$;
- $E_f : A \rightarrow \omega(A)$ is a partial function that assigns to an action $a$ a fault handling conversation expression which represents all conversations that can be invoked by the fault handling for $a$;

$$dom(\epsilon) \cap dom(E_c) = \emptyset$$
$$dom(\epsilon) \cap dom(E_f) = \emptyset.$$
DEFINITION 4 (TERM). The set of terms over an action set $\mathcal{A}$ is given by the following grammar: $(a_i \in \mathcal{A}, i \in \mathbb{N}, n \geq 2, B \subseteq \mathcal{A}$, and $a \in \mathcal{A}$):

$$\text{term ::= } \tau \mid \ell \mid a \mid \bigcup B \mid \bigcap B \mid \bigodot B \mid a_1; \ldots; a_n \mid \text{term}$$

The set of all terms over $\mathcal{A}$ is denoted by $\text{Term}(\mathcal{A})$, and $|\text{term}| = |\text{term}|$. The term $\tau$ represents no action is needed to be invoked. The term $\ell$ represents a coordination action. The term $ae(m, o)$ represents a call to web method $m$ with the expected outcome $o$. The term $\bigcup B$ represents a nondeterministic choice in action set $B$. The term $\bigcap B$ represents parallel invocation of all actions in $B$, and the parent waits for all sides to return. If any side fails or is an exception action, the parent fails. The term $\bigodot B$ represents parallel invocation of all actions in $B$, while the return of any action will return the parent. Only when all sides are exception actions, the parent fails. $a_1; \ldots; a_n$ represents sequential calls. The term $|\text{term}|$ represents the term in the brackets is a transaction term, and any exception action invoked from the term will cause compensation or fault handling to the actions which are invoked before from the term. The term in transaction term must result exceptions.

The sequence of invocations between Web services can be specified through an automaton. For indicating the place where exceptions occur and the coordinations in Web services, coordination protocol automaton (CPA) is presented.

DEFINITION 5 (COORDINATION PROTOCOL AUTOMATA). A coordination protocol automaton $G$ is a 3-tuple $(\mathcal{A}, \mathcal{L}, \mathcal{D}, \delta)$, where

- $\mathcal{A} \subseteq \mathcal{M} \times \mathcal{O}$ is a set of actions;
- $\mathcal{L} \subseteq N \times \{0, \varnothing\}$ is a set of locations, where $N$ is a set of location names, and $\{0, \varnothing\}$ is the location type set, the default type of location is $\varnothing$, $(\perp, \varnothing) \in \mathcal{L}$ is the return location, $(\varnothing, 0) \in \mathcal{L}$ is the execution location;
- $\delta \subseteq \mathcal{L} \times \{\perp, \varnothing\} \times \text{Term}(\mathcal{A}) \times \mathcal{L}$ is the transition relation set.

We use $\text{in}(l) = \{t \mid t \in \delta \land t = (l, \text{term}, l')\}$ to represent all the transitions whose target locations are $l$, and $\text{out}(l) = \{t \mid t \in \delta \land t = (l, \text{term}, l')\}$ to represent all the transitions whose source locations are $l$. The transition of a location is determined by its type. If the type of location $l$ is $\varnothing$, then the transitions in $\text{out}(l)$ can be taken if there exists one transition in $\text{in}(l)$ that has been taken before. If the type of location $l$ is $\perp$, then the transitions in $\text{out}(l)$ can be taken if all transitions in $\text{in}(l)$ have been taken before. A location is terminating in CPA if there exists a trace starting from the location and ending with $(\perp, \varnothing)$ or $(\varnothing, 0)$.

DEFINITION 6 (PROTOCOL INTERFACE, PI). A protocol interface $T$ is a 5-tuple $(G, D, \mathcal{R}, \mathcal{R}_c, \mathcal{R}_e)$, where

- $G$ is a coordination protocol automaton to specify interface behaviour;
- $D \subseteq \mathcal{A}$ is the provided action set;
- $\mathcal{R} : \mathcal{A} \rightarrow G$ is a partial function which assigns to a action the start location in $G$;
- $\mathcal{R}_c : \mathcal{A} \rightarrow \mathcal{L}$ is a partial function which assigns to a success action the start location in $G$ for compensation;
- $\mathcal{R}_e : \mathcal{A} \rightarrow \mathcal{L}$ is a partial function which assigns to an exception action the start location in $G$ for fault handling;
- $\text{dom}(\mathcal{R}_c) \cup \text{dom}(\mathcal{R}_e) = \text{dom}(\mathcal{R}_c), \text{dom}(\mathcal{R}_c) \cap \text{dom}(\mathcal{R}_e) = \emptyset$, and $D \subseteq \text{dom}(\mathcal{R}_c)$.

A location is terminating in $\text{PI}$ if it is terminating in CPA and the location of each action in the terminating trace is also terminating in $\text{PI}$. Given a protocol interface $\mathcal{T} = (G, D, \mathcal{R}, \mathcal{R}_c, \mathcal{R}_e)$, the underlying signature interface of $T$ (denoted by $\text{ps}i(T)$) is $(\mathcal{A}, \mathcal{S}, \mathcal{S}_c, \mathcal{S}_e)$, where $\mathcal{A} = \mathcal{A}$; $\mathcal{S}(a) = \text{sig}(\mathcal{R}(a))$ if $\mathcal{R}(a)$ is defined, otherwise $\mathcal{S}(a)$ is undefined; $\mathcal{S}_c(a) = \text{sig}(\mathcal{R}_c(a))$ if $\mathcal{R}_c(a)$ is defined, otherwise $\mathcal{S}_c(a)$ is undefined; $\mathcal{S}_e(a) = \text{sig}(\mathcal{R}_e(a))$ if $\mathcal{R}_e(a)$ is defined, otherwise $\mathcal{S}_e(a)$ is undefined. The function $\text{sig} : L \rightarrow 2^A$ is defined as follows:

$$\text{sig}(\langle \perp, \varnothing \rangle) = \emptyset, \text{sig}(\langle \varnothing, 0 \rangle) = \emptyset,$$

$$\varphi(T) = \bigcup_{t \in \mathcal{R}(T), \varnothing \notin \mathcal{L}_t} \varphi(\mathcal{S}_c(a)) \cup \varphi(\mathcal{S}_e(a)),$$

$$\varphi(\mathcal{D}) = \emptyset, \varphi(\bigcap B) = \emptyset, \varphi(\bigodot B) = \emptyset, \varphi(\bigcup B) = \bigcup \{\mathcal{L}_t \mid t \in \mathcal{R}(T), \varnothing \notin \mathcal{L}_t\},$$

$$\varphi(\text{term}) = \varphi(\text{term}).$$

A protocol interface $T$ is well-formed if the following conditions hold: $\text{ps}i(T)$ is well-formed; if $a \in \text{dom}(\mathcal{R}_c)$, then $\mathcal{R}_c(a)$ is terminating; if $a \in \text{dom}(\mathcal{R}_e)$, then $\mathcal{R}_e(a)$ is terminating. For the sake of simplicity, it is assumed that: no transaction term can be invoked by exception action, nor can it be invoked by compensation or fault handling; transaction term cannot be invoked recursively or parallel. The types of an action $a$ in a protocol interface $T$ are same as those of $a$ in $\text{ps}i(T)$.

Signature interface and conversation interface depict the static invocation relations of Web service interfaces. Protocol interface describes dynamic invocations in Web service interfaces. The interface behaviour in protocol interface is complicated. It should ensure not only the invocation process be recorded for compensation or fault holding, but also the sequence of which should agree to the long-running transaction model. Intuitively, the invocation process is pushdown, and the process can continue only after the completion of every invoked action. The sequence of compensation and fault handling should be reverse of the sequence of the previous invocations, so the recorded actions should be first in last out. For precise definition of the semantics, we can use the model which is a tree nested by a stack to interpret protocol interface execution.

An execution of a protocol interface is an alternating sequence of states and the sets of actions, which is $\{v_0, A_0, v_1, A_1, \ldots\}$ with $\forall i \in \{0, 1, 2, \ldots\}, \{v_i, A_i\}$ the execution is finite if it can reach an end state with the empty stack and the single node tree. A trace of a protocol interface is the projection of an execution to its action sets. Given a protocol interface $T$ and a provided action $a \in D$, the interface behaviour invoked by $a$ can be transformed into a labeled transition system (LTS), which is denoted by $\text{LTS}(T, a)$.

The theory of Web service interface provides the foundation to check the compatibility and refinement of Web services. Given two Web service interfaces, one may check whether they are compatible and can work together properly. On the other hand, it is expected to replace a Web service in a system (environment) with another Web service without affecting the running of system. After replacement, all components in the system can still cooperate properly as before. Intuitively, the supported, success and exception actions are the guarantees of the Web service, and the required actions are the assumptions to the environment. The replacing Web service should guarantee more and assume fewer than the replaced one. Therefore, it is desired to formalize the realistic service composition language such as BPEL in the theory of Web service interface, so as to support the service-oriented development of software systems.

3. TRANSLATION FROM BPEL TO PI

BPEL can describe orchestration and choreography information of Web service compositions. A BPEL orchestration description for a Web service contains two parts: the description of provided methods and the connection types with other services, which is a
The activities in the sequence can be translated recursively. The activities in the fork term can suitably describe it, and the activities in each branch can be translated recursively. While can be mapped to a self loop in CPA, but the while conditions cannot be described. Links in flow activities are used for coordination between parallel activities. The joinCondition describes different join policy. According to the link semantics in BPEL, we can use the coordination action and location type of protocol interface to specify it as follows: because the target of link can be taken only after the source activity of link has been taken and the joinCondition is satisfied, we can map the target activity to a transition whose start location type is ⊥; when the source activity ends, it should invoke a coordination action and reach a middle location that will determine the invocation of target activity; we can use the middle location to specify the joinCondition of the target activity, which may require that all the incoming links should reach or a single reach of the incoming links is enough.

3.2 Structured BPEL activities

BPEL structured activities, such as sequence, switch, flow and while, are different invocation modes of interface behaviour. Figure 2 lists the translation rules for sequence, switch, flow, and while. Sequence denotes the sequential calls of activities, which can be mapped to a CPA transition whose term is an sequential call term. The activities in the sequence can be translated recursively. Switch can be mapped to a transition whose term is a nondeterministic choice term, and the activities of each case can be translated recursively. Flow represents that the included activities will be processed in parallel. The fork term can suitably describe it, and the activities in each branch can be translated recursively. While can be mapped to a self loop in CPA, but the while conditions cannot be described. Links in flow activities are used for coordination between parallel activities. The joinCondition describes different join policy. According to the link semantics in BPEL, we can use the coordination action and location type of protocol interface to specify it as follows: because the target of link can be taken only after the source activity of link has been taken and the joinCondition is satisfied, we can map the target activity to a transition whose start location type is ⊥; when the source activity ends, it should invoke a coordination action and reach a middle location that will determine the invocation of target activity; we can use the middle location to specify the joinCondition of the target activity, which may require that all the incoming links should reach or a single reach of the incoming links is enough.

3.3 Fault and Compensation handlers

The fault handling or compensation in BPEL appears within the scope. The BPEL process is the global scope, and the basic activities may have their own scopes. Figure 3 lists the translations for the scope and the corresponding translations for the activities in it. Every catch in faultHandlers represents one case of the activity invocation in the scope. If the invocation process is successful, the cancel invocation is defined in compensationHandler. In Figure 3 every catch is mapped to an exception action, and the activities in catch are mapped recursively to the fault handling behaviour in protocol interface. The translation of compensationHandler is similar.

The translation can support multi-scope compensation or fault handling. For the global process, we should attach the provided actions with the global faultHandlers and compensationHandler. If a local activity does not have an explicit fault or compensation...

---

### Table: Sample Code Translation

<table>
<thead>
<tr>
<th>BPEL</th>
<th>Sample Code</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortType</td>
<td><code>&lt;portType …&gt;</code></td>
<td>Add actions of methods to D Normal Invocations</td>
</tr>
<tr>
<td>Basic activities</td>
<td><code>&lt;… receive …&gt;</code></td>
<td>act</td>
</tr>
<tr>
<td>Doing nothing</td>
<td><code>&lt;… empty …&gt;</code></td>
<td>act</td>
</tr>
<tr>
<td>Signaling faults</td>
<td><code>&lt;… throw …&gt;</code></td>
<td>act</td>
</tr>
<tr>
<td>Terminating Instance</td>
<td><code>&lt;… terminate …&gt;</code></td>
<td>act</td>
</tr>
<tr>
<td>No more activity</td>
<td><code>&lt;… act1 …&gt;</code></td>
<td>Reached act</td>
</tr>
</tbody>
</table>

**Figure 1:** Translating the basic BPEL units.

**Figure 2:** Translating the structured BPEL activities.

**Figure 3:** Translating the fault and compensation handlers.
compatibility and substitutivity of some BPEL processes. The presented language (XML), we can use XML elements as the input. The requirements and explanations of Algorithm 1 and 2 are given as follows:

- we are only concerned with the control flow in BPEL;
- the Condition in BPEL is specified in XPath language, this feature is simply described by adding multiple outgoing transitions in CPA;
- in Algorithm 1, we start from each provided action in \( D \). After translating the corresponding normal behaviour part of action, if the global process has no fault handler or compensation handler, the action is defined to \( \bot \) in \( S_C \) or \( S_F \). If the global process has fault handler or compensation handler, the compensation or corresponding fault handler part should be translated using Algorithm 2;
- the input parameter \( s_n \) in Algorithm 2 is responsible for deciding the resulted actions of the input XML element;
- because the translation of link is straightforward, so it is not reflected in the algorithms;
- the translation can directly call TranslateBPEL2PI with the wsdl file and the toplevel process element of bpel XML file.

4. VERIFICATION

For ensuring the high confidence of Web service system, we want to ensure the correctness of BPEL description. The presented Web service interface theory provides the foundation for verification. After translating from BPEL to protocol interface, the LTS interface behaviour model can be generated. Some verification operations can be taken on the LTS model. Besides that, the compatibility and substitutivity of some BPEL processes can also be checked.

Algorithm 1 TranslateBPEL2PI(\( \text{wsdl, process} \))

Input : The input process of BPEL
Output : The target protocol interface \( T = (G, D, \mathcal{R}, S_C, S_F) \)

Variables : boolean \( \text{fhandler, Chandler, action a, a}_1 \)

Element \( \text{elem}_t \), location \( q_1, q_2 \)

1. Set the elements of protocol interface \( T \) to \( \{ \} \)
2. \( D = D \cup \{ \text{all the resulted actions from all provided methods in wsdl} \} \)
3. \( \text{fhandler} = \text{F, Chandler} = \text{F} \)
4. \( \text{elem}_t \) := the activity element of the input process
5. If process has a \( \text{faultHandlers} \) then
6. \( \text{fhandler} := \text{T} \)
7. If process has a \( \text{compensationHandler} \) then
8. \( \text{Chandler} := \text{T} \)
9. For each action \( a \in D \) do
10. \( a_1 := \text{the action resulted from elem}_t \) according to \( a \)
11. \( \mathcal{R} = \mathcal{R} \cup \{ (a, q_1), (a, q_2) \} \)
12. \( \delta = \delta \cup \{ (q_1, a_1), (q_1, a_2) \} \)
13. TranslateElement2PI(\( \text{elem}_t, T, a_1, q_2 \))
14. If \( a \) is successful then
15. If \( \text{fhandler} \) then
16. \( R_C = R_C \cup \{ (q_2) \} \)
17. \( \text{elem}_t \) := \( \text{compensationHandler} \) activity
18. TranslateElement2PI(\( \text{elem}_t, T, \text{none}, q_2 \))
19. Else
20. \( R_C = R_C \cup \{ (\bot) \} \)
21. If \( a \) is exceptional then
22. If \( \text{fhandler} \) then
23. \( R_F = R_F \cup \{ (a, q_2) \} \)
24. \( \text{elem}_t := \text{the activity of corresponding faultHandler for a} \)
25. TranslateElement2PI(\( \text{elem}_t, T, \text{none}, q_2 \))
26. Else
27. \( R_F = R_F \cup \{ (\bot) \} \)
28. End
29. End
30. Return \( T \)

Algorithm 2 TranslateElement2PI(\( \text{elem}_t, T, q, a) \))

Input : The input element of BPEL process, the target protocol interface \( T = (G, D, R, S_C, R_F) \)

Output : none

Variables : boolean \( \text{fhandler, Chandler, action a, a}_1 \)

Element \( \text{elem}_t \), location \( q_1 \)

1. \( \text{fhandler} := \text{F, Chandler} := \text{F} \)
2. If \( \text{elem}_t \) has a \( \text{faultHandlers} \) then
3. \( \text{fhandler} := \text{T} \)
4. If \( \text{elem}_t \) has a \( \text{compensationHandler} \) then
5. \( \text{Chandler} := \text{T} \)
6. Do the translation for the \( \text{activity} \) in the input \( \text{element} \) starting from \( q \) according to \( s_n \) using the translation rules above
7. For each resulted action \( a \) from above translation, where \( a \notin \text{dom}(\mathcal{R}) \)
8. \( \mathcal{R} = \mathcal{R} \cup \{ (q_1, a_1) \} \)
9. \( \text{elem}_t := \text{the corresponding for a} \)
10. TranslateElement2PI(\( \text{elem}_t, T, q, a \))
11. End
12. If \( s_n \) is a success action and \( s_n \notin \text{dom}(\mathcal{S}_C) \) then
13. If \( \text{fhandler} \) then
14. \( \mathcal{S}_C = \mathcal{S}_C \cup \{ (q_3, q_1) \} \)
15. Elements := \( \text{compensationHandler} \) activity
16. TranslateElement2PI(\( \text{elem}_t, T, \text{none}, q_1 \))
17. Else
18. \( \mathcal{S}_C = \mathcal{S}_C \cup \{ (q_3, \bot) \} \)
19. If \( s_n \) is an exception action and \( s_n \notin \text{dom}(\mathcal{S}_F) \) then
20. If \( \text{fhandler} \) then
21. \( \mathcal{S}_F = \mathcal{S}_F \cup \{ (q_3, q_1) \} \)
22. \( \text{elem}_t := \text{activity in faultHandlers according to} s_n \)
23. TranslateElement2PI(\( \text{elem}_t, T, \text{none}, q_1 \))
24. Else
25. \( \mathcal{S}_F = \mathcal{S}_F \cup \{ (q_3, \bot) \} \)
The first subsection presents the translation procedure from protocol interface to LTS. The second subsection presents the model checking approach to verifying protocol property. The third subsection presents our verification process and the methodology.

4.1 Translation

The translation from protocol interface to LTS model can be taken using the transition rules of protocol interfaces. For precise definition for the transitions of protocol interface, we can use the model which is a tree nested by a stack to interpret its execution. The main ideas of transition rules can be briefly explained as follows:

- The beginning is the invocation of an action in the provided action set \( D \). The initial state is a tree which has only one node and a stack whose content is decided by the type of the supported action. If the action is a success action, the stack is empty; else if it is an exception action, it will be pushed into the stack.
- The operations in transitions can be divided into two parts: tree operations and stack operations. The tree operations depict a pushdown system. Only leaf nodes of the tree can be operated. The coordination protocol automaton decides the terms to invoke.
- The terms \( a, \sqsubseteq B \) and \( a_1; \ldots; a_n \), lead to pushing down the leaf node. The terms \( \sqsubseteq B \) and \( \sqsubseteq B \) lead to branching the leaf node. The reaching of \( \langle \bot, \varnothing \rangle \) or \( \langle \varnothing, \varnothing \rangle \) leads to cutting nodes. Transaction term leads to pushing down the leaf node with a transaction node.
- For pushdown operations, if the location of current leaf node is reached from a supported success action, or the leaf node is not pushed from a transaction term node, no stack operation is needed. If current node location is reached from a supported exception action, or the node is branched from a transaction term node and the transaction node is not under compensation or fault handling, some stack operations are needed for recording the operation term.
- When an exception location is reached, some coordination should be taken. There are two complicate cases. The first case is that the exception location is reached from a \( \exists B \) term, and it will cause the global exception and the other branches should be terminated. The second case is that the exception location is reached from a \( \exists B \) term, and whether it can cause the global exception is determined by the other branches. If one of the other branches returns successfully, the parent will be successful. If one of the other branches does not return, this branch should wait until the return of all of the other branches. If exception also occurs in each of the other branches, the global exception occurs.

Given a protocol interface \( \mathcal{T} \), and a supported action \( a \), the transformation procedure of \( LTS(\mathcal{T}, a) = (S_a, I_a, L_a, \Delta_a) \) can be briefly given as follows:

- \( S_a \) contains the states appeared in the transitions because of the invocation of \( a \);
- if \( a \) is a success action, \( I_a \) contains only one state which is a single node tree nested by a empty stack; if \( a \) is an exception action, \( I_a \) contains only one state which is a single node tree nested by a stack, which only contains \( a \);
- \( L_a \) contains the transition labels appeared in the transitions;
- \( \Delta_a \) contains the underlying transition relations using the protocol interface transition rules from the invocation of \( a \).

4.2 Model Checking

Based on the transformation, some temporal properties can be verified on protocol interface. The protocol property must be formed in a \( \varphi \), where \( \varphi \) is the formula in Action Set Computation Tree Logic (ASCTL) and \( a \in \text{dom}(R) \). The ASCTL formula set over an action set \( A \) can be given by the following grammar, where \( D \subseteq A \).

\[
\begin{align*}
\chi &::= \text{true} \mid \text{false} \mid D \mid \chi \land \chi' \\
\varphi &::= \text{true} \mid \text{false} \mid \neg \varphi \mid \varphi \land \varphi' \mid E \gamma \mid A \gamma \\
\gamma &::= [\varphi(x)] U [\chi(x)] \varphi' \mid [\varphi(x)] U \{\chi(x)\} \varphi'
\end{align*}
\]

Compared to ACTL in [11], the syntax and semantics of ASCTL are similar except the following differences:

- for the semantics of ASCTL, the labels of transitions in LTS model are action sets;
- \( A \models D \) if \( A \cap D \neq \emptyset \), where \( A \) is the transition label and \( D \) is the action set in ASCTL formula;

Many other ASCTL operators can be derived from the basic ones, e.g. \( EF, AF, EG \) and \( AG \), the derivations are same as [11]. Model checking technique is used for protocol property verification. The method for model checking is same as [11], which used symbolic model checking method based on fixed point calculation [20] for ACTL verification.

![Figure 4: The verification process of protocol property.](image)

The verification process of protocol property is shown in Figure 4. First, the corresponding LTS of the property should be generated, and the LTS only contains the transitions whose labels are external action sets. Next, we can use the presented method to verify the ASCTL formula in the property.

4.3 Verification Process and Methodology

The interface theory based verification process for BPEL description can be taken using the above methods. The verification process and our main methodology are shown in Figure 6. The Web service development progress can be divided into many steps, in some of which verification can be taken to ensure the high confidence.

After getting requirements, the designer can begin to design the service model according to the requirements. During these steps, the designer can specify some protocol properties that the implementation must satisfy. When finishing the design, the designer can give it to the implementor, who can implement the actual Web services by BPEL. After finishing implementation, the implementor can input the BPEL description from which the corresponding protocol interfaces can be translated. After generating protocol interfaces, the LTS behaviour models can be generated from the protocol properties specified by the designer and the generated protocol interfaces. Using the model checking technique, the protocol properties can be verified on the implementation model to ensure...
the correctness and consistency. If some requirement properties are not satisfied, the designer or implementor should modify its design or implementation to ensure the satisfaction. After above all steps, the verified BPEL description can be deployed to some BPEL engines such as BPEL4J. Then the actual Web services can be established and provide services to the client.

5. A CASE STUDY

This section demonstrates the formalization and verification of a travel agency Web service implemented in BPEL, and it can provide airline reservation, car rental and weather forecast according to the country and city chosen by client. The interaction architecture of the travel agency service is shown in Figure 7.

In wsdl description, the provided port of the travel agency is travelPT, which contains a method bookTravel. Client can call the method to reserve a travel. When a client requests for a travel reservation, the BPEL process behind the travel agency service begins to proceed the request. The travel agent will invoke the services of different airline companies according to the destination of client, car rental company and weather forecast service and do the reservation.

According to above description, the BPEL process is an executable process. As a shorthand, the BPEL specification is shown in Figure 8. The execution structure of the process is shown in Figure 5. The execution structure of the process is shown in Figure 8. The solid arrow represent the sequential invocations. The
client wants to go, the travel agent invokes different airline services. If client wants to travel to US, then usAirlines service will be invoked. If client wants to travel to Canada, then canadaAirlines service will be invoked, otherwise only britishAirlines service will be invoked. At the same time as the reservation takes place, the weather service is contacted, and it provides the weather forecast of the client destination. After the reservation has been completed, the car rental service might be invoked, which happens only if the city of destination is in Canada or US and is not New York City. After the flow activity completes, the reservation result will be assigned and replied to the client.

Based on the translation methods in section 3, we can translate the BPEL description which has no compensation or fault handler into the corresponding protocol interface, which is shown in Figure 9. For short, we use the set whose element is formed in \( \{ l, \text{term}, l' \} \) to represent the transition relation in CPA, where \( l \) and \( l' \) are the locations in CPA. The partial functions \( R, R_1, R_3 \) are indicated by adding an action before a transition, and the resulted location of the action is simply the source location of the transition at the head position. Because there is no compensation or fault handler, and there is no information about exceptions, so all the actions are success actions, and the start locations of the compensation for all actions are same to \( l \), which are not shown in Figure 9. The provided action set \( D \) is \( \{ \text{bookTravel,OK} \} \).

To consider the case further, we make the modifications to illustrate the multi-scope compensation and fault handling. Supposing that the car rental company may report that there is no car for renting, the reservation process will fail in this situation. The global fault handler for the reservation exception will send an apologetic letter to the client. The compensation in child scope for airline switch activity is to recede the airline reservation. The fault handler for the failure of car rental invocation is an empty activity. The new protocol interface is shown in Figure 10. It is necessary to say that the translated orchestration process in Figure 10 needs a service for sending letters.

After composing the protocol interfaces of different parts, Some protocol properties can be verified on the travel agency service system. The protocol properties and the corresponding verification results are listed in Table 1. The meanings of the properties can be given as follows: 1) the success of travel booking will always lead to the success of ticket reservation; 2) the success of travel booking will always lead to the ticket reservation to the United States. The reason of the verification failure is because the travel to the city outside of the United States will not reserve the United States traveling tickets; 3) the travel agency will always apologize to client for the booking failure resulted from the car rental failure; 4) the ticket return will always occur because of the booking failure resulted from the car rental failure. The reason of the verification failure is the ticket reservation may not occur when the car rental failure occurs; 5) the ticket return must not occur before the ticket reservation; 6) the apology must not occur before the ticket return. The reason of the verification failure is same with that of the fourth property; 7) if the ticket return will occur, the apology must not occur before the ticket return.

Table 1: The protocol properties and the corresponding verification results.

<table>
<thead>
<tr>
<th>Protocol Property</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{bookTravel,OK} \rightarrow \text{AF} (\text{airline,OK}) )</td>
<td>True</td>
</tr>
<tr>
<td>( \text{bookTravel,OK} \rightarrow \text{AF} (\text{USReservation,OK}) )</td>
<td>False</td>
</tr>
<tr>
<td>( \text{bookTravel,NOCAR} \rightarrow \text{AF} [(\text{apologize,OK})] )</td>
<td>True</td>
</tr>
<tr>
<td>( \text{bookTravel,NOCAR} \rightarrow \text{EF} [\text{Recede,OK}] )</td>
<td>False</td>
</tr>
<tr>
<td>( \text{bookTravel,NOCAR} \rightarrow \sim E { \sim { \text{airline,OK} } } \cup { \text{Recede,OK} } )</td>
<td>True</td>
</tr>
<tr>
<td>( \text{bookTravel,NOCAR} \rightarrow \sim E { \sim { \text{Recede,OK} } } \cup { \text{apologize,OK} } )</td>
<td>False</td>
</tr>
<tr>
<td>( \text{bookTravel,NOCAR} \rightarrow \sim E { \sim { \text{Recede,OK} } } \cup { \text{apologize,OK} } )</td>
<td>True</td>
</tr>
</tbody>
</table>

6. RELATED WORK

There are many research on formalization and verification for Web service composition languages [13, 14, 15, 16]. [13] used Petri net to formalize the Web service composition. Foster[14] used finite state process (FSP) to formalize BPEL and verified the description on LTSA. In [15], guarded finite state machine (GFSA)
is used to describe service composition, and the description can be translated to Promela which can be verified on SPIN. In [16], state transition system (STS) are used to formalize the BPEL abstract process. Some requirement formula formed in EaGLe can be verified on the STS model using the model checker NuSMV.

The approaches in [13, 14, 15, 16] are deficient in modeling transaction behaviour of orchestration. The multiple scope, fault handling and compensation features in BPEL are not well supported, and the dynamic behaviour for fault handling or compensation when exceptions occur are not clearly discussed. In this paper, we propose a new approach to model and verify BPEL executable process behaviour with transactions from the perspective of the interface theory, which matches the service-oriented computing naturally and can deal with the transaction behaviour rigorously.

Besides that, there are several works on formalizing business process with transactions and transaction features in BPEL. [17] extended communicating sequential process (CSP) to enable the description of long-running transactions. [18] proposed enhanced Sagas language to specify compensations in flow composition languages. [19] gave a complete abstract operational semantics for BPEL using abstract state machine (ASM). The work in [19] is based on the work in [20], which also use ASM to formalize BPEL and incorporate most syntax elements in BPEL such as data handling, fault handling and compensation.

[17] and [18] only presented a theory supporting compensation mechanism, the application on BPEL or some other description languages was not discussed. To my knowledge, the gap between BPEL and the foundation theory is a problem. [19] and [20] only presented the operational semantics for BPEL and no other further mechanism was presented, such as verification or testing. In this paper, we propose the underlying formalization theory as well as verification method, and bridge the gap between BPEL and our formalism. The verification method is automatic and can effectively verify the transaction behaviour in BPEL.

7. CONCLUSION AND FUTURE WORK
This paper presents a formal foundation and a verification method for service orchestration in BPEL. The formal theory is introduced to specify Web service interfaces, in which the transaction information has been modeled. With a set of interface descriptions, Web services can be described from the views of signature, conversation and protocol. With the interface theory, a formalization of orchestration in BPEL is presented in a transformational way. For a Web service, the description of provided methods and the invocation process can be translated into the Web service protocol interface automatically. Based on the formalism, some verification operations can be taken and the compatibility and refinement can be checked between the services. In addition, with the help of the Web service interface theory dealing with transactions well, fault handling and compensations in BPEL can be formalized nicely in the presented approach, and the confidence of BPEL description can be improved through the automatic verification process.

The ongoing and future work are to investigate an integrated formalism for both of service orchestration and choreography by Web service interface theory, and to improve our formalism for specifying more syntax elements in BPEL such as scoped variable.

8. REFERENCES