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
E-mail: z.b.chen@mail.edu.cn
E-mail: jiwang@mail.edu.cn
E-mail: dongwei@mail.edu.cn
E-mail: qzc@nudt.edu.cn
*Corresponding author

Abstract: BPEL4WS (BPEL) is a Web service composition language for service-oriented computing. Service orchestration can be specified by executable processes in BPEL. However, it lacks of a formal foundation for specification and verification of service-oriented systems. This paper presents an improved protocol interface for Web services. Compared to the existing interface theory, the presented interface can describe some more advanced features of long running transaction such as nested transaction. An interface theory based formalization is presented for service orchestration in BPEL. The transformational approach is proposed for translating BPEL processes to protocol interfaces. With the formalization, a formal technique is presented for model checking of BPEL program with respect to the protocol properties. A set of case studies are demonstrated to illustrate our approach.

Keywords: Service composition; Web services; Transaction; BPEL4WS; Interface theory; Verification.


Biographical notes: Zhenbang Chen is a PhD candidate at National Laboratory for Parallel and Distributed Processing of China. His research interests are in the areas of service-oriented computing, formal methods and software engineering. Ji Wang is a professor at National Laboratory for Parallel and Distributed Processing of China. His research interests include high confidence software and systems, distributed computing and software engineering. Wei Dong is an associate professor at National Laboratory for Parallel and Distributed Processing of China. His research interest is high confidence software engineering. Zhichang Qi is a professor at National Laboratory for Parallel and Distributed Processing of China. His research topic is software engineering.

1 INTRODUCTION

Service composition is an important theme in service-oriented computing. Many languages such as WSFL (Leemann, 2001), WSCI (Arkin et al., 2002), XLANG (Thatte, 2001) and BPEL (Curbera et al., 2003) are emerging as Web service composition languages. It is desirable to build formal foundations for service composition in order to understand service-oriented software systems well.

In general, service composition can be categorized into service orchestration and choreography. Orchestration specifies service composition from the perspective of a single Web service (Peltz, 2003). The orchestration description may be an executable process that interacts with some outside Web services to achieve the composition and the business logic. Choreography specifies the service composition from the perspective of the global system. It describes the public message exchanges between a set of...
Web services (Peltz, 2003) and the global business logic of the system. Though the different perspectives exist in orchestration and choreography, there are some common aspects of them, such as business logic and description methods. Currently, some service composition languages can describe orchestration as well as choreography such as BPEL.

Recently, it is proposed to build a formal foundation for Web services by interface theory. As a formal foundation of the component-based design, the theory of interface automata (de Alfaro and Henzinger, 2001) is presented for specification of component interfaces. Beyer et al. (2005a,b) extend it and present a Web service interface description language, which can describe the interfaces in three levels, i.e. signature, consistency and protocol. However, the transaction feature is not considered in the existing interface theories, though 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 (Little, 2003). For this reason, it is desirable to have the interface theory for the Web services with long running transactions to facilitate the orchestration.

Inspired by the ideas of Aspect-Oriented Programming (AOP) (Kiczales et al., 1997), Chen et al. (2006a,b) extend the formalism of Web service interfaces proposed in Beyer et al. (2005a) to describe transaction information in all three levels, i.e. signature, conversation and protocol. In each level, the transaction behaviour is weaved into the normal interface behaviour. The model presented in Chen et al. (2006a,b) can capture some basic features of long running transactions such as fault handling and compensation. However, while being a formal foundation for the flexible transaction description mechanisms that exist in many Web service composition languages, it should be enriched with capability for specifying more advanced features such as nested transactions and user-defined fault handling.

Currently, most researches on the formal semantics of BPEL mainly take into account the non-transactional behaviour of BPEL. Therefore, the analysis methods based on those semantic models cannot deal with the transaction behaviour in BPEL. Furthermore, the scope-based transaction description mechanism of BPEL can flexibly depict the transaction behaviour through scope-based fault handling and compensation mechanisms, but it lacks of an effective formalism to interpret these mechanisms as well as the BPEL analysis method that can cover the transaction behaviour.

The contributions of this paper contain three parts. First, the protocol interface is improved to capture some more advanced features of long running transaction to facilitate the transaction description in Web service orchestration. Second, the formalization for BPEL is proposed based on the improved protocol interface, and the translation methods and algorithms for translating BPEL into protocol interface are presented. Third, the verification methodology is proposed for ensuring the correctness of the orchestration description.

Through our approach, the transaction behaviour in BPEL can be interpreted rigorously and nicely. The developer of the Web service orchestration can use our approach to specify the transaction behaviour flexibly, find errors in the development process and ensure the correctness.

The remainder of this paper is organized as follows. Section 2 presents the protocol interface for Web services with transactions. Section 3 proposes the translation methods from BPEL to protocol interface and the translation algorithms. Section 4 presents a model checking based verification methodology for BPEL description. Section 5 gives two examples to illustrate the formalization and verification. In Section 6, the related work is reviewed and compared. Section 7 concludes the paper and discusses some issues of future research.

### 2 Web Service Protocol Interface

![Figure 1: Supply chain management system.](image)

A classical Web service-based system, supply chain management system that is shown in Figure 1, is given for the demonstration of the interface theory. The system is composed of six Web services. Each labeled arrow from one service to another indicates the Web method call from the caller to the callee. Shop supports the Web method SellItem that can be called by the Client to start the selling process. When the selling process starts, the Shop will first check the availability of items to be sold by calling the method ChkAvail, which requires the Web method ChkStore implemented by Store to check whether the desirable items are in stock and deduct the number of items if the stock checking is successful. If the stock is inadequate, the selling process fails. If the stock is inadequate or the stock after deducting is below a certain amount, the Store department will make an order from the Supplier and get some new items. If the availability checking is successful, the Shop will parallelly process the payment by calling the method ProcPay and the delivery by calling the method ShipItem. ProcPay is implemented by the Bank and its success can be compensated by calling the method Compensate. ShipItem is implemented by Transport and its success can be compensated by calling the
method Withdraw. If all the above steps are successful, the selling process is successful, otherwise the successful steps before failure should be compensated and the failed steps should be handled. For instance, the Shop will call the method Apologize implemented by itself to send an apologetic letter to the Client because of the failure of the selling process.

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. For instance, SellItem is a method provided by the Web service Shop in Figure 1, and ⟨SellItem, FAIL⟩ is one of its actions. 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. ⟨SellItem, FAIL⟩ is an exception action that indicates the selling failure.

There are different detailed interface descriptions from Web service providers. For this reason, the interface theory for describing the transaction information is proposed at three different abstract levels of signature, conversation and protocol. Signature interface is the base for the other two. Because we only use the protocol interface to formalize BPEL, the conversation interface (Chen et al., 2006b) will be omitted in this paper.

Inspired by the ideas of AOP, we separate the descriptions of fault handling and compensation behaviour from those of normal behaviour in the interface description. In the interface semantics, the fault handling and compensation behaviour can be weaved into the normal behaviour to describe the transaction information.

2.1 Brief Description of Syntax

Let M be a finite set of web methods, O be a finite set of outputs, A ⊆ M × O denote the set of actions and dom(f) denote the domain of the function f. Based on the signature interface in Beyer et al. (2005a), we extend it with two partial functions Sc and Sf that specify the fault handling and compensation behaviours. Signature interface is defined as follows.

Definition 1 (Signature Interface, SI). A signature interface P is a 4-tuple (A, S, Sc, Sf), where

- A ⊆ M × O is a set of actions that can appear in P;
- S : A → 2A is a partial function that assigns to an action a a set of actions that can be invoked by a;
- Sc : A → 2A is a partial function that assigns to an action a a set of actions that can be invoked by the compensation for a;
- Sf : A → 2A 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(Sc) ∩ dom(Sf) = ∅, dom(Sc) ⊆ dom(S), and dom(Sf) ⊆ dom(S).

Signature interface describes the direct invocation relation of Web service interfaces. An action may have different types. An action a ∈ A is a supported action if S(a) is defined. A method m ∈ M is a supported method if there exists a supported action a = (m, o). An action a is a success action if Sc(a) is defined. An action a is an exception action if Sf(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:

\[ \text{required}(a') = (\exists a \in \text{dom}(S). \ a' \in S(a)) \lor (\exists a \in \text{dom}(Sc). \ a' \in Sc(a)) \lor (\exists a \in \text{dom}(Sf). \ a' \in Sf(a)). \]

Service registries often require service providers to publish solid interface descriptions. Well-formedness is used to describe 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. Following formulae can express the well-formedness of signature interface.

\[ \forall a \in A, \ b \in A. \ a = (m, o_1) \land \text{required}(a) \land b \in \text{dom}(S) \land b = (m, o_2) \Rightarrow a \in \text{dom}(S), \] \[ \forall a, b \in \text{dom}(Sf). \ \forall c \in \text{dom}(Sc). \ a \notin Sf(b) \land a \notin Sc(c). \]

Example 2.1 (Well-formed SI). The signature interface \( P_{\text{shop}} = (A_{\text{shop}}, S_{\text{Shop}}, S_{\text{SC_shop}}, S_{\text{SF_shop}}) \) of the Web service Shop is defined as follows.

\[ A_{\text{shop}} = \{ \text{SellItem, SOLD}, \text{SellItem, FAIL}, \text{ChkAvail, OK}, \text{ChkAvail, FAIL}, \text{ProcPay, OK}, \text{ProcPay, FAIL}, \text{ShipItem, OK}, \text{ShipItem, FAIL}, \text{ChkStore, OK}, \text{ChkStore, FAIL}, \text{Apologize, OK}, \text{SendLetter, OK}, \text{Recede, OK}, \text{Compensate, OK}, \text{RStore, OK}, \text{WithDraw, OK} \} \]

\[ S_{\text{Shop}} = \{ \text{SellItem, SOLD} \rightarrow \{ \text{ChkAvail, OK}, \text{ProcPay, OK}, \text{ShipItem, OK} \}, \text{SellItem, FAIL} \rightarrow \{ \text{ChkAvail, FAIL}, \text{ChkAvail, OK}, \text{ProcPay, FAIL}, \text{ProcPay, OK}, \text{ShipItem, FAIL} \}, \text{ChkStore, OK} \rightarrow \{ \text{ChkStore, OK}, \text{ChkStore, FAIL} \}, \text{Apologize, OK} \rightarrow \{ \text{SendLetter, OK} \}, \text{Recede, OK} \rightarrow \{ \text{Compensate, OK}, \text{RStore, OK}, \text{WithDraw, OK} \} \} \]

\[ S_{\text{SC_shop}} = \{ \text{SellItem, SOLD} \rightarrow \{ \text{Recede, OK} \}, \text{ChkAvail, OK} \rightarrow \emptyset, \text{Apologize, OK} \rightarrow \emptyset, \text{Recede, OK} \rightarrow \emptyset \} \]

\[ S_{\text{SF_shop}} = \{ \text{SellItem, FAIL} \rightarrow \{ \text{Apologize, OK} \}, \text{ChkAvail, FAIL} \rightarrow \emptyset \} \]

The supported action ⟨SellItem, SOLD⟩ will directly invoke three actions: ⟨ChkAvail, OK⟩, ⟨ProcPay, OK⟩ and ⟨ShipItem, OK⟩. ⟨ChkAvail, OK⟩ is a required action, whose method is supported by Shop itself, and it is a supported action too. ⟨SellItem, SOLD⟩ is a success action, and ⟨SellItem, FAIL⟩ is an exception action. Because \( P_{\text{shop}} \) supports all the required actions whose methods are
supported methods and no exception action is invoked in compensation or fault handling, $P_{\text{Shop}}$ is well-formed.

Given two Web service interfaces, we want to check whether they can cooperate properly. First, two Web services cannot support the same actions. Second, the new Web service interface, which is composed of them, should be well-formed. The compatibility of signature interface is given as follows.

**Definition 2 (SI Compatibility).** Given two signature interfaces $P_1 = (A_1, S_1, S_{\text{C}1}, S_{\text{F}1})$ and $P_2 = (A_2, S_2, S_{\text{C}2}, S_{\text{F}2})$, $P_1$ and $P_2$ are compatible if the following conditions are satisfied:

- $\text{dom}(S_1) \cap \text{dom}(S_2) = \emptyset$;
- $P_c = P_1 \cup P_2 = (A_1 \cup A_2, S_1 \cup S_2, S_{\text{C}1} \cup S_{\text{C}2}, S_{\text{F}1} \cup S_{\text{F}2})$ is well-formed.

If two signature interfaces $P_1$ and $P_2$ are compatible (denoted by $\text{comp}(P_1, P_2)$), their composition (denoted by $P_1 \parallel P_2$) is $P_c$. The composition operator is commutative and associative.

**Example 2.2 (Compatibility of SI).** The signature interface $P_{\text{Store}} = (A_{\text{Store}}, S_{\text{Store}}, S_{\text{C} \text{store}}, S_{\text{F} \text{store}})$ of the Web service Store is defined as follows. $P_{\text{Store}}$ is well-formed and compatible with $P_{\text{Shop}}$ in Example 2.1.

- $A_{\text{Store}} = \{ \text{ChkStore,OK}, \text{ChkStore,FAIL} \cup \text{RStore,OK} \}$
- $S_{\text{Store}} = \{ \text{ChkStore,OK} \rightarrow \{(\text{GetOffer,OK},\text{Order,OK})\}, \text{ChkStore,FAIL} \rightarrow \emptyset, \text{RStore,OK} \rightarrow \emptyset \}$
- $S_{\text{C} \text{store}} = \{ \text{ChkStore,OK} \rightarrow \{(\text{GetOffer,OK},\text{Order,OK})\} \}$
- $S_{\text{F} \text{store}} = \{ \text{ChkStore,FAIL} \rightarrow \{(\text{GetOffer,OK},\text{Order,OK})\} \}$

To enable top-down design, it is desirable to replace a Web service in a system (environment) with a new Web service without affecting the running of the system. After replacement, all parts of 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 of the environment. It is necessary to point out that the required actions may be supported by Web service itself or other Web services of its environment. The replacing Web service should guarantee more and assume fewer than the replaced Web service.

**Definition 3 (SI Substitutivity).** Given two signature interfaces $P_1 = (A_1, S_1, S_{\text{C}1}, S_{\text{F}1})$ and $P_2 = (A_2, S_2, S_{\text{C}2}, S_{\text{F}2})$, $P_2$ refines $P_1$ ($P_2 \Rightarrow P_1$) if the following conditions are satisfied:

- For every $a \in A$, if $P_1$ supports $a$, then $P_2$ supports $a$;
- For every $a \in A$, if $a$ is a success action in $P_1$, then $a$ is a success action in $P_2$;
- For every $a \in A$, if $a$ is an exception action in $P_1$, then $a$ is an exception action in $P_2$;
- For every $a, a' \in A$, and $a \in \text{dom}(S_1)$, if $a' \in \zeta_2(a)$, where $\zeta_2 \in \{S_2, S_{\text{C}2}, S_{\text{F}2}\}$, then $a' \in \zeta_1(a)$;
- For every unsupported web method $m \in M$ in $P_2$, if $\langle m, o \rangle$ is a required action in $P_2$, then $\langle m, o \rangle$ is a required action in $P_1$.

The first three conditions ensure that the replacing Web service guarantees every action and its types that are guaranteed by the replaced one. The last two conditions ensure that every required action in $P_2$ is required by $P_1$, they describe that $P_2$ does not assume more actions which are supported by environment than $P_1$. Given three signature interfaces $P_1, P_2$, and $P_3$, if $\text{comp}(P_1, P_3), \text{comp}(P_2, P_3)$, and $P_2 \Rightarrow P_1$, then $P_2 \parallel P_1 \parallel P_3$.

Signature interface does not have any sequence information. Beyer et al. (2005a) present protocol interface to indicate the sequences of action invocations. We extend the protocol interface in Beyer et al. (2005a) to enable transaction description (Chen et al., 2006a,b). In this paper, we improve the elements of the protocol interface in Chen et al. (2006b) to have a more flexible transaction description mechanism including nested transaction, user-defined fault-handling, etc.

In Web services, the modes of action invocations include thread creation, choice, parallel executions, transaction block beginning, etc. We use terms to represent these different modes.

**Definition 4 (Term).** The set of terms over an action set $A$ and a transaction block name set $B_T$ is given by the following grammar ($a \in A, B \subseteq A$, and $n_1, n_2 \in B_T$):

$$\text{term ::= } \tau \mid \ell \mid a \mid \cup B \mid \cap B \mid \exists B \mid [n_1, n_2] \mid [\sim n]$$

The set of all terms over $A$ and $B_T$ is denoted by $\text{Term}(A, B_T)$. The term $\tau$ represents no action is needed to be invoked. The term $\ell$ represents a coordination action, which can be used to control the invocation sequences of some parallelly invoked actions. The term $a = \langle m, o \rangle$ represents a call to the web method $m$ with the expected output $o$. The term $\cup B$ is a choice term, which represents a non-deterministic choice in the action set $B$. The term $\cap B$ is a fork term, which represents parallel invocations of all actions in $B$, and the term waits for all actions to return. If any action fails or is an exception action, the term fails. The term $\exists B$ is a fork-choice term, which represents parallel invocations of all actions in $B$, whereas the return of any action will return the term. Only when all sides are exception actions, the term fails. $a_1 \cap \ldots \cap a_n = \{a_1, \ldots, a_n\}$, where $\in \{\cup, \cap, \exists\}$. The term $[n_1, n_2]$ represents the beginning of transaction block whose name is $n_1$ and its parent transaction block name is $n_2$. The term $[\sim n]$ represents the invocation of default compensation for the transaction block $n$.

The sequence of invocations between Web services can be specified through automata. To indicate the place where exceptions occur and the coordinations between invocations, we propose coordination protocol automaton as follows.

**Definition 5 (Coordination Protocol Automaton, CPA).**
A coordination protocol automaton \( G \) is a 4-tuple \((A, B_T, L, \delta)\), where

- \( A \subseteq M \times O \) is a set of actions;
- \( B_T \) is a set of transaction block names;
- \( L \subseteq N \times \{q, \varnothing\} \) is a set of locations, where \( N \) is a set of location names, \( \{q, \varnothing\} \) is the location type set, and the default type of location is \( q \);
- \( \delta \subseteq L \times \{(\perp, q), (\varnothing, q)\} \times \text{Term}(A, B_T) \times L \) 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 with the same target location \( l \), and \( \text{out}(l) = \{t \mid t \in \delta \land t = (l, \text{term}, l)\} \) to represent all the transitions with the same source location \( l \). The transition of a location is determined by its type. If the type of the 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 the location \( l \) is \( q \), 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 \((\perp, q)\) or \((\varnothing, q)\). As a shorthand, we use location name to denote the location whose type is \( q \). For example, \( \perp \) is used to denote \((\perp, q)\). Based on CPA, the protocol interface can be defined as follows.

**Definition 6 (Protocol Interface, PI).** A protocol interface \( T \) is a 5-tuple \((G, D, R, R_C, R_F)\), where

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

A location is terminating in \( PI \) if it is terminating in \( G \) and the location of each invoked action in the terminating trace is also terminating in \( PI \). Given a protocol interface \( T = (G, D, R, R_C, R_F) \), the underlying signature interface of \( T \) (denoted by \( \text{psi}(T) \)) is \((A_s, S, S_C, S_F)\), where \( A_s = A; S(a) = \text{sigl}(R(a)) \) if \( R(a) \) is defined, otherwise \( S(a) \) is undefined; \( S_C(a) = \text{sigl}(R_C(a)) \) if \( R_C(a) \) is defined, otherwise \( S_C(a) \) is undefined; \( S_F(a) = \text{sigl}(R_F(a)) \) if \( R_F(a) \) is defined, otherwise \( S_F(a) \) is undefined. The function \( \text{sigl} : L \rightarrow 2^A \) is defined as follows:

\[
\text{sigl}((\perp, q)) = \emptyset, \text{sigl}((\varnothing, q)) = \emptyset, \\
\text{sigl}(q) = \bigcup_{(l, \text{term}, q') \in \delta, \text{term} \neq \emptyset} \varphi(\text{term}) \cup \text{sigl}(q'), \\
\varphi(\tau) = \emptyset, \varphi(a) = \{a\}, \varphi(\sqcup) = \emptyset, \varphi(\sqcap) = \emptyset.
\]

A protocol interface \( T \) is well-formed if the following conditions hold: \( \text{psi}(T) \) is well-formed; if \( a \in \text{dom}(R) \), then \( R(a) \) is terminating; if \( a \in \text{dom}(R_C) \), then \( R_C(a) \) is terminating; if \( a \in \text{dom}(R_F) \), then \( R_F(a) \) is terminating. The types of an action \( a \) in a protocol interface \( T \) are same as those of \( a \) in \( \text{psi}(T) \). In addition, an action \( a \in D \) is a provided action, which can be invoked by the client, and the other actions in \( \text{dom}(R) \) are private actions, which can not be invoked by the client.

**Example 2.3 (Well-formed Protocol Interface).** The protocol interface \( T_{shop} = (G_{shop}, D_{shop}, R_{shop}, R_{C_{shop}}, R_{F_{shop}}) \) of the Web service \( Shop \) is defined as follows. As a shorthand, we use the set whose elements are formed in \((l, \text{term}, l)\) to represent the transition relation set in CPA \( G_{shop} \), where \( l \) and \( l' \) are the locations in \( G_{shop} \). The partial functions \( R, R_C, R_F \) are indicated by adding an action before a transition, and the corresponding location of the action is simply the source location of the transition at the head position \((A_{shop} \text{ is the same as that in Example 2.1}) \).

\[
\{ \\
\langle \text{SellItem, SOLD} \rangle \rightarrow_R (q_0, \langle \text{SellItem, CheckAvail} \rangle, q_1), \\
(q_1, \langle \text{ChkAvail, OK} \rangle, q_2), \\
(q_2, \langle \text{SellItem, ProcPay} \rangle, q_3), \\
(q_3, \langle \text{ProcPayOK} \rangle, q_4), \\
(q_4, \langle \text{SellItem, Shipped} \rangle, q_5), \\
(q_5, \langle \text{ShipItem, OK} \rangle, \perp), \\
\langle \text{SellItem, FAIL} \rangle \rightarrow_R (q_6, \langle \text{SellItem, CheckAvail} \rangle, q_7), \\
(q_7, \langle \text{ChkAvail, FAIL} \rangle, q_8), \\
(q_8, \langle \text{ChkAvail, OK} \rangle, q_9), \\
(q_9, \langle \text{SellItem, ProcPay} \rangle, q_{10}), \\
(q_{10}, \langle \text{ProcPayOK} \rangle, q_{11}), \\
(q_{11}, \langle \text{ProcPayOK} \rangle, q_{12}), \\
(q_{12}, \langle \text{SellItem, Shipped} \rangle, q_{13}), \\
(q_{13}, \langle \text{ShipItem, FAIL} \rangle, q_{14}), \\
\langle \text{ChkAvail, OK} \rangle \rightarrow_R (q_{14}, \langle \text{ChkStore, OK} \rangle, \perp), \\
\langle \text{ChkAvail, FAIL} \rangle \rightarrow_R (q_{15}, \langle \text{ChkStore, FAIL} \rangle, q_{16}), \\
\langle \text{Apologize, OK} \rangle \rightarrow_R (q_{16}, \langle \text{SendLetter, OK} \rangle, \perp), \\
\langle \text{Recede, OK} \rangle \rightarrow_R (q_{17}, \langle \text{WithDraw, OK} \rangle, q_{18}), \\
(q_{18}, \langle \text{Compensate, OK} \rangle, q_{19}), \\
\langle \text{SellItem, SOLD} \rangle \rightarrow_R (q_{20}, \langle \text{Recede, OK} \rangle, \perp), \\
\langle \text{Recede, OK} \rangle \rightarrow_R (q_{21}, \langle \text{ChkAvail} \rangle, \perp), \\
\langle \text{ChkAvail, FAIL} \rangle \rightarrow_R (q_{22}, \langle \text{ChkAvail} \rangle, \perp), \\
\langle \text{SellItem, FAIL} \rangle \rightarrow_R (q_{23}, \langle \text{SellItem} \rangle, q_{24}), (q_{24}, \langle \text{Apologize, OK} \rangle, q_{25}) 
\}
\]

\( T_{shop} \) models the interface behaviour of \( Shop \), and \( \text{psi}(T_{shop}) \) is the signature interface \( \mathcal{T}_{shop} \) in Example 2.1. Because \( \mathcal{T}_{shop} \) is well-formed and \( T_{shop} \) satisfies all the other conditions, \( T_{shop} \) is well-formed. Action \( \langle \text{SellItem, SOLD} \rangle \) can invoke three transaction blocks in sequence. Action \( \langle \text{SellItem, FAIL} \rangle \) can invoke different sequences of actions, and each sequence needs compensation and fault handling because of the exception. The fault handling for \( \langle \text{SellItem, FAIL} \rangle \) will first invoke the default compensation for transaction block \( \text{SellItem} \) by \( \langle \text{SellItem} \rangle \), after which action \( \langle \text{Apologize, OK} \rangle \) will be invoked to send the apologetic letter. The provided action set \( D_{shop} \) is \{\langle \text{SellItem, OK} \rangle, \langle \text{SellItem, FAIL} \rangle \}. \]
successful executions of \( B \) bit, the fault handling must first execute for \( B \) with the long-running transaction model.

also the sequence of compensation invocations should agree

rectness, which will be presented at Section 3.1.

process can continue only after the completion of every

Definition 7 (PI Compatibility). Given two protocol interfaces \( T_1 = (G_1, D_1, R_1, R_{C1}, R_{F1}) \) and \( T_2 = (G_2, D_2, R_2, R_{C2}, R_{F2}) \), \( T_1 \) and \( T_2 \) are compatible if the following conditions are satisfied:

- \( \psi(T_1) \) and \( \psi(T_2) \) are compatible;
- \( L_1 \cap L_2 = \{(\bot, \emptyset), (\emptyset, \emptyset)\} \);
- \( B_{F1} \cap B_{F2} = \emptyset \);
- \( T_c = T_1 \cup T_2 = (G_1 \cup G_2, D_1 \cup D_2, R_1 \cup R_2, R_{C1} \cup R_{C2}, R_{F1} \cup R_{F2}) \) is well-formed, where \( G_1 \cup G_2 = (A_1 \cup A_2, B_{T1} \cup B_{T2}, L_1 \cup L_2, \delta_1 \cup \delta_2) \).

If \( T_1 \) and \( T_2 \) are compatible (denoted by \( \text{comp}(T_1, T_2) \)), their composition (denoted by \( T_1 \parallel T_2 \)) is \( T_c \). The composition operator is commutative and associative. The substitutivity relation between protocol interfaces should be defined based on the semantics to ensure the temporal correctness, which will be presented at Section 3.1.

Protocol interface describes temporal invocations in Web service interfaces. It must ensure that not only the successful transaction block invocations should be recorded, but also the sequence of compensation invocations should agree with the long-running transaction model.

2.2 Protocol Interface Semantics

The execution of protocol interface will be either successful or exceptional. If an exception occurs, compensation or fault handling will be taken. The process is a long-running transaction model, whose invocation process can be exemplified in Figure 2. Transaction block \( B \) contains two sequential transaction blocks \( B_1 \) and \( B_2 \). After successful executions of \( B_1 \) and \( B_2 \), an exception occurs. The transaction block \( B \) is failed. Supposing the fault handling for \( B \) is to compensate the successful transaction blocks in it, the fault handling must first execute \( C_2 \) that is the compensation for \( B_2 \), and execute \( C_1 \) that is the compensation for \( B_1 \) after completing \( C_2 \).

Definition 8 (Tree). A tree over a finite set of labels \( \mathcal{L} \) is a partial function \( t : \mathbb{N}^* \rightarrow \mathcal{L} \), where \( \mathbb{N}^* \) denotes the word set over the natural number set.

We use \( \rho \) to denote empty word. \( p \cdot j \) denotes the concatenation of the word \( p \) with \( j \in \mathbb{N} \). For the sake of simplicity, we use \( p/j \) to denote \( p \cdot j \). The set of leaf nodes of tree \( t \) is \( \text{leaf}(t) = \{ p \in \text{dom}(t) \mid \forall j \in \mathbb{N}, pj \notin \text{dom}(t) \} \). For a node \( p \) in \( \text{dom}(t), \text{child}(p) = \{ q \mid \exists j \in \mathbb{N}, q = pj \land q \in \text{dom}(t) \} \), and \( \text{parent}(p) = \{ q \mid \exists j \in \mathbb{N}, p = qj \land q \in \text{dom}(t) \} \). Let \( \mathcal{T}(\mathcal{L}) \) denote all trees on a finite label set \( \mathcal{L} \).

Definition 9 (Stack). A stack over a finite set of labels \( \mathcal{L} \) is a partial function \( s(m) : \mathbb{N} \rightarrow \mathcal{L} \), where \( \mathbb{N} \) is the natural number set, and \( \text{dom}(s(m)) = \{ n \mid n < m \land n \in \mathbb{N} \} \).

\( s(0) \) is the empty stack. \( s(m)(m{-}1) \) is the top element of stack \( s(m) \). Let \( \mathcal{S}(\mathcal{L}) \) denote all stacks on a finite label set \( \mathcal{L} \). We use \( \text{Len}(s(m)) \) to denote the number of the elements in stack \( s(m) \), apparently \( \text{Len}(s(m)) = m \). \( \text{Push}(s(m), l) \) to denote pushing \( l \) into stack \( s(m) \) and \( \text{Pop}(s(m)) \) to denote returning and popping the top element in stack \( s(m) \).

Definition 10 (Stack Group). A stack group of a finite set of names \( N_s \) over a finite set of labels \( \mathcal{L} \) is a partial function \( g(n) : N_s \rightarrow \mathcal{S}(\mathcal{L}) \), where \( \mathcal{S}(\mathcal{L}) \) is the set of all stacks on \( \mathcal{L} \).

We use \( \mathcal{G}(N_s, \mathcal{L}) \) to denote all stack groups of a finite set of names \( N_s \) over a finite label set \( \mathcal{L} \). Given a protocol interface \( T = (G, D, R, R_C, R_F) \), its semantics is defined by a labeled transition system. The set of states is \( \mathcal{T}(Q_t) \times \mathcal{G}(B_T, \text{Term}(A, B_T)) \times 2^d \), that is, the Cartesian products of trees over \( Q_t = Q \times A^* \times \phi \times B_T, \) stack groups over \( B_T \) and \( \text{Term}(A, B_T) \), and power set of transition set \( \delta \), where \( Q \) is the location set of the coordination protocol automaton \( G, A \) is the action set in \( G, \phi = \{ \circ, \emptyset, \ast \} \) is the node type set, \( B_T \) is the transition block name set in \( G, \) and \( \delta \) is the transition set in \( G \). The underlying transition relation of \( T \) is a transition relation \( \mathcal{T}(Q_t) \times \mathcal{G}(B_T, \text{Term}(A, B_T)) \times 2^d \), where the state transition is the set of elements from \( A \cup \{ \text{ret, exp, cpb} \} \). We write \( \nu \xrightarrow{\nu'} for (\nu, B, \nu') \in \rightarrow \), where \( \nu = (t, g, s) \) and \( B \subseteq A \cup \{ \text{ret, exp, cpb} \} \). The transition can be given by a set of transition rules.

The beginning is the invocation of a provided action. Supposing we start from invoking a provided action \( a = (m, o) \), the initial state is \( \nu_{\text{initial}} = (l_\text{initial}, g_\text{initial}, s_\text{initial}) = (\{(\rho, (R(a), a, \circ, m))\}, \{(m, s_0(0))\}, \emptyset) \). The definition of transition rules are given as follows, where

- for a transition, the source state is defined as \( \nu = (t, g, s) \), and the target state as \( \nu' = (t', g', s') \);
- we use \( q(w)\beta/n \) to represent \( (q, w, \beta, n) \) in \( Q_t \), and \( w = p, q\beta/n \) is used to represent it. For example, \( q\beta/n \) represents \( (q, p, \emptyset, n) \);
- \( \delta(q) = \text{term}(q', q) \) denotes that there exists a transition \( (q, \text{term}, q') \) in the coordination protocol automaton \( G \), and the transition can be taken now, which means that if the type of \( q \) is \( \emptyset \), then \( i_\delta(q) \subseteq s \);
if action $c$ is supported by $\mathcal{R}$, $q_c = \mathcal{R}(c)$, otherwise $q_c = \bot$;
if $a$ is supported by $\mathcal{R}_c$, $\mu(a) = \mathcal{R}_c(a)$, otherwise if $a$ is supported by $\mathcal{R}_x$, $\mu(a) = \mathcal{R}_x(a)$, otherwise $\mu(a) = \bot$;
when applying all rules, if the transition $(q, \text{term}, q')$ is taken and the type of $q$ is $\sqcup$, then $s' = (s' \setminus in(q))$ after applying.

(Empty)
If there exists a node $p$ such that $p \in leaf(t)$, $t(p) = q \circ n$, and $\delta(q) = (\tau, q')$, then $t' = (t \setminus \{p, q \circ n\}) \cup \{(p, q' \circ n)\}$, $g' = g$, and $s' = s \cup \{(q, \tau, q')\}$.

(Coordination)
If there exists a node $p$ such that $p \in leaf(t)$, $t(p) = q \circ n$, and $\delta(q) = (t, q')$, then $t' = (t \setminus \{p, q \circ n\}) \cup \{(p, \bot \circ n)\}$, $g' = g$, and $s' = s \cup \{(q, t, q')\}$.

(Pushdown) $\nu \xrightarrow{\Delta} \nu'$
If there exists a node $p$ such that $p \in leaf(t)$, $t(p) = q \circ n$, and $\delta(q) = (r, q')$:

- $r = a$, then $t' = (t \setminus \{p, q \circ n\}) \cup \{(p, q' \circ n)\}$, $(\rho_0, q_a \circ n)$, and $\mathcal{M} = \{a\}$;
- $r = \bot \sqcup \mathcal{B}$, then $t' = (t \setminus \{p, q \circ n\}) \cup \{(p, q' \circ n)\}$, $(\rho_0, q_0 \circ n)$, where $c \in \mathcal{B}$, and $\mathcal{M} = \{c\}$;
- $r = \sqcap \mathcal{B}$, and $\mathcal{B} = \{a_0, ..., a_n\}$, then $t' = (t \setminus \{p, q\circ n\}) \cup \{(p, q' \circ n), (p, q_o \circ n), ..., (p, q_n \circ n)\}$, and $\mathcal{M} = \mathcal{B}$;
- $r = \sqcup \mathcal{B}$, and $\mathcal{B} = \{a_0, ..., a_n\}$, then $t' = (t \setminus \{p, q\circ n\}) \cup \{(p, q' \circ n), (p, q_0 \circ n), ..., (p, q_n \circ n)\}$, and $\mathcal{M} = \mathcal{B}$.

In all conditions, $g' = g$, and $s' = s \cup \{(q, r, q')\}$.

(TB-Begin) $\nu \xrightarrow{\{p\theta\}} \nu'$
If there exists a node $p$ such that $p \in leaf(t)$, $t(p) = q \circ n$, and $\delta(q) = (p\theta, q')$, and $g(n_1) = \text{defined}$:

- if $\text{Len}(g(n_1)) = 0$, then $t' = (t \setminus \{p, q \circ n\}) \cup \{(p, q' \circ n)\}$, and $g' = g$;
- if $\text{Len}(g(n_1)) > 0$, then $a = \text{Pop}(g(n_1))$, and $t' = (t \setminus \{p, q \circ n\}) \cup \{(p, q' \circ n), (p, \bot \circ n_1), (p, 00, \mu(a) \circ n_1)\}$.

In all conditions, $s' = s \cup \{(q, p\theta, q')\}$.

(Return) $\nu \xrightarrow{\{\tau\}} \nu'$
If there exists a node $p\theta$ such that $p\theta \in leaf(t)$, $t(p\theta) = \bot \circ n_1$, $t(p) = q \circ n_2$, where $\theta \in \mathbb{N}$:

- $\alpha = \circ$, then $t' = (t \setminus \{(p\theta, \bot \circ (w \circ n_1)\})$;
- $\alpha = \sqcap$, then $t' = (t \setminus \{(p\theta, \bot \circ n_1)\}) \cup \{(p, \bot \circ n_2)\}$;
- $\alpha = \circ$, and $\text{Len}(g(n_2)) > 0$, then $a = \text{Pop}(g(n_2))$, $t' = (t \setminus \{(p\theta, \bot \circ n_1)\}) \cup \{(p\theta, \mu(a) \circ n_1)\}$;
- $\alpha = \circ$, and $\text{Len}(g(n_2)) = 0$, then $t' = (t \setminus \{(p\theta, \bot \circ n_1)\}) \cup \{(p, q \circ n_2)\}$, and $g = g \setminus \{(n_2, g(n_2))\}$.

In all conditions, if $n_1 \neq n_2$ and $w \neq \rho$, then $\text{Push}(g(n_2), w)$, else $g' = g$.

(Permission) $\nu \xrightarrow{\{\exp\}, \nu'}$
If there exists a node $p\theta$ such that $p\theta \in leaf(t)$, $t(p\theta) = \bigotimes(w) \circ n_1$, $t(p) = q(w) \circ n_2$, where $\theta \in \mathbb{N}$:

- $\alpha = \circ$ and $w = \rho$, then $t' = (t \setminus \{(pp', res) | p' \in N' \land res = t(pp')\}) \cup \{(p, (\mathcal{S}(w_1) \circ n_2))\}$, if $p = \rho$ and $w = \rho$, then $t' = (t \setminus \{(pp', res) | p' \in N' \land res = t(pp')\}) \cup \{(p, (\mathcal{S}(w_1) \circ n_2))\};$
- $n_1 \neq n_2$ and $w \neq \rho$, then $t' = (t \setminus \{(p\theta, \mathcal{S}(w) \circ n_1)\}) \cup \{(p\theta, \mu(w) \circ n_1)\}$;
- $\alpha = \sqcup$ and $w = \rho$, then if all nodes in $\{(p, p' \circ \text{child}(p) \land p' \neq p\theta\}$ have no child, and the locations of all nodes are same to $\sqcup$, then $t' = (t \setminus \{(pp, res) | p' \in N' \land res = t(pp')\}) \cup \{(p, (\mathcal{S}(w_1) \circ n_2))\}$.

In all conditions, $g' = g$.

![Image of operation diagrams](image-url)
compensation will invoke the next compensation process if there exist other successful blocks that needs compensations. If no successful block that need compensation exists, the compensation stack should be removed from the stack group. If the return of transaction block is successful, then the corresponding transaction block action should be recorded.

- 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 fork 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 fork-choice 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 returns of all the other branches. If exception also occurs in each of the other branches, the global exception occurs. If the return of a transaction block is an exception location, the corresponding fault handling process should be taken.

An execution of a protocol interface is an alternating sequence of states and the sets of actions, which is $v_0, A_0, v_1, A_1, \ldots$, where $v_i \xrightarrow{\text{alt}} v_{i+1}$ for all $i \in \{0, 1, \ldots, n\}$. A trace of a protocol interface is the projection of an execution to its action sets.

Based on the transition rules, we can use the LTS simulation relation to define the substitutivity relation of protocol interfaces.

**Definition 11 (Labeled Transition System, LTS).** A labeled transition system is a 4-tuple $(S, I, L, \Delta)$, where $S$ is the set of states, $I \subseteq S$ is the set of initial states, $L$ is the set of labels, and $\Delta \subseteq S \times L \times S$ is the transition relation set.

Given a protocol interface $T$ and a provided action $a = (m, o) \in D$, the interface behaviour invoked by $a$ can be transformed into a labeled transition system, which is denoted by $\text{LTS}(T, a)$. The transformation procedure of $\text{LTS}(T, a) = (S_a, I_a, L_a, \Delta_a)$ can be given as follows:

- $S_a = T(Q_t) \times G(B_T, \text{Term}(A, B_T)) \times 2^L$;
- $I_a = \{\{(p, (R(a), a, o, m))\}, \{(m, s_m(0))\}, \emptyset\}$;
- $L_a = 2^{\text{alt}(\text{ret, exp, cpb})}$;
- $\Delta_a$ contains the underlying transition relations using the protocol interface transition rules staring from the invocation of $a$.

Because ret, exp, cpb are not external Web service actions, the transitions labeled by them do not assume the environment. The simulation relation of the underlying labeled transition systems can be extended to relax the conditions that substitutivity should satisfy. We denote $\langle s_1, a, s_1' \rangle \in \Delta$ as $s_1 \xrightarrow{a} s_1'$. If $t = a_1 a_2 \ldots a_n \in L^*$, $s_1 \xrightarrow{a_1} \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_1'$ is denoted as $s_1 \xrightarrow{t} s_1'$.

**Definition 12 (LTS Weak Simulation).** Given two LTSs $M_1 = (S_1, I_1, L_1, \Delta_1)$ and $M_2 = (S_2, I_2, L_2, \Delta_2)$ and a label set $W$, $M_2$ is weakly simulated by $M_1$ over the label set $W$ if there exists a relation $\bowtie W \subseteq S_1 \times S_2$ such that:
- for every $s_1 \in M_1, s_2 \in M_2$, if $s_2 \bowtie W s_1$, then for every $(s_2, a, s_2') \in \Delta_2$, there exists $s_1 \xrightarrow{a} s_1'$ in $\Delta_1$, such that $s_2' \bowtie W s_1'$, where $s_1 \xrightarrow{a} s_1'$ represents $s_1 \xrightarrow{t} s_1'$, in which $t = a_1 a_2 \ldots a_n$, and there exists only one $a_i$ that $a_i = a$, and all the other labels are all in $W$;
- for every $s_2 \in M_2$, there exists $s_1 \in M_1$ such that $s_2 \bowtie W s_1$.

The substitutivity relation between protocol interfaces can be given as follows.

**Definition 13 (PI Substitutivity).** Given two protocol interfaces $T_1 = (G_1, D_1, R_1, \text{RC}_1, \text{RF}_1)$ and $T_2 = (G_2, D_2, R_2, \text{RC}_2, \text{RF}_2)$, $T_2$ refines $T_1$ ($T_2 \bowtie T_1$) if the following conditions are satisfied:

- $\psi(T_2) \bowtie \psi(T_1)$;
- for every $a \in D_1$, LTS($T_2, a$) is weakly simulated by LTS($T_1, a$) over $2^{\text{ret, exp, cpb}}$.

Given three protocol interfaces $T_1, T_2$, and $T_3$, if $\text{comp}(T_1, T_3), \text{comp}(T_2, T_3)$, and $T_2 \bowtie T_1$, then $T_2 \bowtie T_3 \bowtie T_1 \bowtie T_3$.

### 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 the provided methods and the connection types with other services, and this part is contained in a wsdl file; the description of the invocation process for the provided methods, and this part is contained in a bpe1 file. Orchestration information can be deployed to some BPEL engines such as BPWS4J (IBM, 2004). Many orchestrated Web services can be composed into a composite Web service or system.

In BPEL specification document, there exist many reasons for exception such as network blocking and calling throw activity. For the sake of simplicity, we only take into account throw activity and assume that no exception can be thrown from faultHandlers and compensationHandler. Because we are only concerned with the control flow in BPEL description, data handling is omitted in this paper. Due to the these assumptions, we give the BPEL syntax paradigm as follows, where $n$ is the activity name.

**Process ::=** process($n$, Activity, Fault, Comp)

**Activity ::=** receive($n$)

| invoke($n$) |
| reply($n$) |
| assign($n$) |
| throw($n$) |
| compensate($n$) |
According to different types of the syntax elements in BPEL, we can construct the corresponding Web service protocol interface. For each activity, the translation will be given by a specific procedure whose input parameters will be the translated activity, the start location for the translation, the enclosing scope name of the activity and the activity result to the enclosing scope. We use translate(Activity, q, s, r) to denote the procedure whose return is the reached new location. The meanings of the sub translating procedures used in translate(Activity, q, s, r) are given in Section 3.4.

3.1 Basic BPEL activities

The translation procedures of the basic BPEL activities are listed in Table 1. The basic activities of BPEL include receive, invoke, reply and assign. These activities are atomic units of BPEL description. They can be mapped to the transitions whose terms are single actions. It is necessary to point out that the exception may be handled by one of the exception handlers and when the exception is detected, the activity will be terminated without performing any other activities after the exception. Moreover, the translation for switch activity can suitably describe it. While activity can be mapped to a self loop in CPA at the moment. However, the translation gives a conservative representation. The child activities of structured activity can be translated recursively. The translation operation diagram is shown in Figure 4.

3.2 Structured BPEL activities

BPEL structured activities, such as sequence, switch, flow and while, are different invocation modes of interface behaviour. Table 2 lists the translations for sequence, switch, flow, and while. Sequence represents the sequential calls of activities, and it can be mapped to the sequential transitions in CPA. The activities in the sequence can be translated recursively. It is necessary to point out that the number of the translated child activities in sequence may be different from m because of the exception, which may also exist in the translation for switch activity. Switch can be mapped to the transition whose term is a nondeterministic choice term. Flow represents that the included activities will be processed in parallel and fork term can suitably describe it. While can be mapped to a self loop in CPA at the moment. However, the translation gives a conservative representation. The child activities of structured activity can be translated recursively. The translation operation diagram is shown in Figure 4.

| empty | sequence(n, Activity1,..., Activitym) |
| switch(n, Activity1,..., Activitym) | flow(n, Activity1,..., Activitym) |
| link(n, Activity1, Activity2) | while(n, Activity) |
| scope(n, Activity, Fault, Comp) | Comp := compensationHandle(Activity) |
| Fault := faultHandlers catchError(n1, Activity1),... | |
| catchError(nm, Activitym)) | faultHandlers catchError(n1, Activity1),... |
| catchError(nm, Activitym), | catchAll(Activityn)) |

Table 1: Translating the basic BPEL activities.

3.3 Scope activity

Both of fault handling and compensation in BPEL appear within a scope. Table 3 lists the translations for the scope, faultHandles and compensationHandles. The translation for the scope can be divided into the following three conditions: (1) the scope result is OK and the activity in the scope will not result in any exception; (2) the scope result is OK and the activity in the scope will result in some exceptions. Some of these exceptions may be han-
### Table 2: Translating the structured BPEL activities.

<table>
<thead>
<tr>
<th>BPEL</th>
<th>Translation Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sequence</strong></td>
<td></td>
</tr>
<tr>
<td>( r = \text{GetResult}(sequence, r) )</td>
<td></td>
</tr>
<tr>
<td>( \delta = \delta \cup {(q_0, \text{sequence}, r'), q_0'} )</td>
<td></td>
</tr>
<tr>
<td><strong>end</strong></td>
<td></td>
</tr>
<tr>
<td>( q_1 = \text{translate}(\text{Activity}1, q_0, s, r') )</td>
<td></td>
</tr>
<tr>
<td>( q_2 = \text{translate}(\text{Activity}2, q_1, s, r') )</td>
<td></td>
</tr>
<tr>
<td>( ... )</td>
<td></td>
</tr>
<tr>
<td>( q_n = \text{translate}(\text{Activity}<em>n, q</em>{n-1}, s, r') )</td>
<td></td>
</tr>
</tbody>
</table>

| **switch** |
| \( r = \text{GetResult}(\text{switch}, r) \) |
| \( \delta = \delta \cup \{(q, \{\text{switches}, r', \} q') \} \) |
| \( R = R \cup \{(\text{switches}, r'), q_0\} \) |
| \( a_1 = \text{branch}_1, \text{GetResult}(\text{Activity}_1, r') \) |
| **end** |
| \( \delta = \delta \cup \{(q_0, \{a_1, ..., a_n\}, q_0') \} \) |

| **flow** |
| \( r = \text{GetResult}(\text{flow}, r) \) |
| \( \delta = \delta \cup \{(q, \{\text{flows}, r'\}, q') \} \) |
| \( R = R \cup \{(\text{flows}, r'), q_0\} \) |
| \( a_1 = \text{branch}_1, \text{GetResult}(\text{Activity}_1, r') \) |
| **end** |
| \( \delta = \delta \cup \{(q_0, \{a_1, ..., a_m\}, q_0') \} \) |

| **while** |
| \( r = \text{GetResult}(\text{while}, r) \) |
| \( \delta = \delta \cup \{(q, \{\text{while}.., r'\}, q') \} \) |
| \( R = R \cup \{(\text{while}.., r'), q_0\} \) |
| \( a_1 = \text{branch}_1, \text{GetResult}(\text{Activity}_1, r') \) |
| **end** |
| \( \delta = \delta \cup \{(q_0, \{a_1, ..., a_m\}, q_0') \} \) |

---

![Translation Diagram](image)

**Figure 4:** Translation diagrams of the structured BPEL activities.

---

According to the BPEL specification, if there is no explicit `faultHandler` for the `scope`, the default `faultHandler` will run all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the corresponding scopes and rethrowing the fault to the next enclosing scope (Curbera et al., 2003), which can be mapped to the translation \( \delta = \delta \cup \{\{q, [n], \emptyset\}\} \) suitably. If there is no explicit `compensationHandler` for the `scope`, the default `compensationHandler` is same as that of default `faultHandlers` without rethrowing the exception.

The translation procedure for `faultHandler` is `translate(fault, q, s, r1, r2)`. The meanings of `q` and `s` are same as those of `translate(seq(Activity, q, s, r), r1)` is the result of the activity in the `scope`, and `r2` is the actual result of the `scope`. `r1 ∈ N` represents that the exception can be handled by a `catch` handle. `r1 ∉ N ∧ r2 = OK` represents that the exception will be handled by the `catchAll` handle. `r1 ∉ N ∧ r2 ≠ OK` represents that the exception will be diffused to the outside `scope`.

Table 4 lists the translation procedure for the `BPEL` process that can be seen as the `global scope`, and translation procedure `translate(seq(Process))` is similar to that of `scope`. The generated action of the `process` should be added not only to the domain of `R`, but also to the provided action set `D`. 


3.4 Translation Algorithm

Based on the translating procedures for different types of BPEL activities, we present the translation algorithms as follows. The requirements and explanations of algorithms are given as follows.

- **BPEL** description is formed in EXtensible Markup Language (XML), parsing the input BPEL XML is omitted in this paper. We suppose that the BPEL inputs of the algorithms are formed in the syntax format at the beginning of this Section. It needs a preprocessing procedure to parse the input BPEL XML. For a short hand, this procedure will not be given.

- The **Condition** in BPEL is specified in XPath language. Because data handling is omitted in this paper, so this feature is not discussed in this paper.

- The details of the translation procedures *translate*, *translate_f* and *translate_p* have been presented in the preceding subsections. Only procedure skeleton will be presented in algorithms 1, 2 and 3.

- **L** and **T** used in the preceding translation procedures are global variables. **L** is a set and **T** is a protocol interface.

### Table 3: Translating the scope activity.

<table>
<thead>
<tr>
<th>BPEL</th>
<th>Translation Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>scope</strong></td>
<td>r = GetResult(scope, r)</td>
</tr>
<tr>
<td></td>
<td>R = GetResult(Activity)</td>
</tr>
<tr>
<td></td>
<td>R_c = GetResult(scope)</td>
</tr>
<tr>
<td></td>
<td>E = GetException(scope)</td>
</tr>
<tr>
<td>r = OK ∧ #E = 0</td>
<td>δ = δ ∪ {(q_i, [n_i, q_i], q_i), (q_i, (scope_n, r'), q_i)}</td>
</tr>
<tr>
<td>R = R ∪ {(scope_n, r'), q_0}</td>
<td>R_c = R_c ∪ {(scope_n, r'), q_i}</td>
</tr>
<tr>
<td>if Comp exists then</td>
<td>translate(Comp, q_s, n, r')</td>
</tr>
<tr>
<td>else</td>
<td>δ = δ ∪ {(q_i, [n_i, q_i], \perp)}</td>
</tr>
<tr>
<td>B = B ∪ {(scope_n, r')}</td>
<td>end</td>
</tr>
<tr>
<td>δ = δ ∪ {(q_i, [n_i, q_i], q_i), (q_i, \perp, B, q_i')}</td>
<td>end</td>
</tr>
</tbody>
</table>

| Fault | N = \{m_1, ..., m_n\} |
|       | if r_1 \in N then |
|       | translate(Activity_r, q_s, OK) |
|       | if r_1 \notin N ∧ r_2 = OK then |
|       | translate(Activity_r_2, q_s, OK) |
|       | if r_1 \notin N ∧ r_2 ≠ OK then |
|       | δ = δ ∪ \{(q_i, \perp, \perp)\} |

### Table 4: Translating the process.

<table>
<thead>
<tr>
<th>BPEL</th>
<th>Translation Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process</strong></td>
<td>r_p = GetProcessResult(process)</td>
</tr>
<tr>
<td></td>
<td>R_p = GetResult(Activity)</td>
</tr>
<tr>
<td></td>
<td>E = GetException(Activity)</td>
</tr>
<tr>
<td></td>
<td>For each r_i \in R_p</td>
</tr>
<tr>
<td></td>
<td>δ = δ ∪ {(q_i, [n_i, q_i], \perp)}</td>
</tr>
<tr>
<td></td>
<td>R = R ∪ {(n, r_i), q_0}</td>
</tr>
<tr>
<td></td>
<td>D = D ∪ {(n, r_i)}</td>
</tr>
<tr>
<td></td>
<td>translate(Activity, q_s, n, r_i)</td>
</tr>
<tr>
<td></td>
<td>translate(Activity, q_0, n, r_i)</td>
</tr>
<tr>
<td></td>
<td>if Comp exists then</td>
</tr>
<tr>
<td></td>
<td>translate(Comp, q_s, n, r_i)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>δ = δ ∪ {(q_i, [n_i, q_i], \perp)}</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

| Fault | N = \{m_1, ..., m_n\} |
|       | if r_1 \in N then |
|       | translate(Activity_r, q_s, OK) |
|       | if r_1 \notin N ∧ r_2 = OK then |
|       | translate(Activity_r_2, q_s, OK) |
|       | if r_1 \notin N ∧ r_2 ≠ OK then |
|       | δ = δ ∪ \{(q_i, \perp, \perp)\} |
Algorithm 1 translate($Activity, q, s, r$)

**Input**: The input $Activity$, start location $q$,
the inclosing scope $s$, and the supposed result $r$;
**Output**: The reached new location after translation;
**Variables**: Set $R, R_e, E$. Result $r′, r, i, boolean b_r$,
location $q′, q_0, q_e, q_r, q_{ef}, q_{r}, q_1, q_2$;
1: if $Activity$ is the source of some links and
   $Activity$ parent is sequence then
2: do the translation procedures in Table 2;
3: return $q'$;
4: end
5: $b_r = false$;
6: if $Activity$ is basic then
7: do the translation procedures in Table 1;
8: if $Activity$ is structured then
9: do the translation procedures in Table 2;
10: if $Activity$ is scope then
11: do the translation procedures in Table 3;
12: if $Activity$ is the source of a link then
13: do the translation procedures in Table 2;
14: $b_r = true$;
15: end
16: if $Activity$ is the target of a link then
17: generate middle location $q_m$ for the $q$;
18: do the translation procedures in Table 2;
19: end
20: if the $Activity$ parent is not sequence and
   $b_r = false$ and $q' ≠ ∅$ then
21: $δ = δ ∪ \{(q', r, ∅)\}$;
22: return $q'$;

Algorithm 2 translate($Fault, q, s, r_1, r_2$)

**Input**: The input $Fault$, start location $q$,
the inclosing scope $s$, the result $r_1$, the actual result $r_2$;
**Output**: none;
**Variables**: Set $N$
1: $N = \{n_1, ..., n_m\}$;
2: if $r_1 \in N$ then
3: translate($Activity_{n_i}, q, s, OK$);
4: if $r_1 \notin N$ and $r_2 = OK$ then
5: translate($Activity_{n_i}, q, s, OK$);
6: if $r_1 \notin N$ and $r_2 ≠ OK$ then
7: $δ = δ ∪ \{(q, r, ∅)\}$;
8: return $R$;

Algorithm 3 translate($Process$)

**Input**: The input $Process$;
**Output**: none;
**Variables**: Set $E, R_p, R_p$; Output $r_i, r_j$,
location $q_0, q_e, q_r, q_{ef}, q_{r}, q_1, q_2$;
1: do the translation procedures in Table 4;

Algorithm 4 GetResult($Activity$)

**Input**: The input $Activity$;
**Output**: The set of possible results of $Activity$;
**Variables**: Set $R, R_e, F$; boolean $r_f$;
1: $R = \emptyset$;
2: if $Activity$ is the basic activity and
   $Activity ≠ throw$ then
3: $R = \{OK\}$;
4: if $Activity = throw(n)$ then
5: $R = \{n\}$;
6: if $Activity$ is the structured activity and
   $Activity ≠ while$ then
7: $r_f = false$;
8: for each $i = 1$ to $n$ do
9: $R_i = GetResult(Activity_{i})$;
10: $R = R \cup R_i$;
11: if OK $\notin R_i \land Activity = sequence$ then
12: $R = R \setminus \{OK\}$;
13: break;
14: end
15: if OK $\notin R_i \land Activity = flow$ then
16: $r_f = true$;
17: end
18: if $r_f = true \land Activity = flow$ then
19: $R = R \setminus \{OK\}$;
20: end
21: if $Activity = while(n, Activity_{j})$ then
22: $R = GetResult(Activity_{j})$;
23: if $Activity = scope(n, Activity_{j}, ...)$ then
24: if $Fault$ has catchAll then
25: $R = \{OK\}$;
26: else
27: $R_i = GetResult(Activity_{j})$;
28: $F = \{n_1, ..., n_m\}$;
29: $R = R_i \setminus F$;
30: if $R < \#R$, then
31: $R = R \cup \{OK\}$;
32: end
33: end
34: return $R$;

- Supposing that the set operation can be done in $O(1)$ time, the maximum complexity of algorithm 1 is $O(k \ast n \ast n)$, where $n$ is the number of XML elements in the corresponding XML of the input $Activity$, and $k$ is the maximum number of results from $Activity$, and this maximum complexity also exists in the algorithms 2 and 3.
- Algorithm 4 is used for getting the result set of the input $Activity$. We suppose that the successful result is $OK$ and all the others are exception names. For the sequence activity, if one of its child activities only results in exceptions, the sequence activity will result in exceptions only, and the rest sequential child activities need not to be translated. If one child activity of the flow activity only results exceptions, the flow activity will not result in $OK$. For the scope activity, if the faultHandles has catchAll handle, the result will be $OK$; if no catchAll handle exists, then the result set will contain the exceptions that cannot be handled, and if some exceptions can be handled, the
result set will contain OK. The maximum complexity of algorithm 4 is \(O(n)\) time, where \(n\) is the number of XML elements of the corresponding XML of the input Activity.

- Algorithm 5 is used for getting the set of actual exceptions that can be caused by the input Activity. For the sequence activity, if one of its child activities only results in exceptions, the exceptions caused by the rest sequential child activities will not be considered. The maximum complexity of algorithm 5 is same as that of algorithm 4.

\[
\text{Algorithm 5 GetException(}\text{Activity}\text{)}
\]

**Input**: The input Activity;
**Output**: The set of exceptions that can be resulted from the Activity;
**Variables**: Set \(E, Ri\);
1. \(E = \emptyset\);
2. if Activity is the basic activity and Activity \(\neq\) throw then
3. \(E = \emptyset\);
4. if Activity = throw(n) then
5. \(E = \{n\}\);
6. if Activity is the structured activity and Activity \(\neq\) while then
7. for each \(i = 1\) to \(m\) do
8. \(Ri = \text{GetResult(}\text{Activity}_i\text{)}\);
9. \(E = E \cup (Ri \setminus \{\text{OK}\})\);
10. if \(\text{OK} \notin Ri \land \text{Activity} = \text{sequence}\) then
11. break;
12. end
13. end
14. end
15. if Activity = while(n, Activity)_1 then
16. \(E = \text{GetException(}\text{Activity}_1\text{)}\);
17. if Activity = scope(n, Activity)_1 then
18. \(E = \text{GetException(}\text{Activity}_1\text{)}\);
19. return \(E\);

- Algorithm 6 is used for getting the result set of the input Process. The maximum complexity of algorithm 6 is \(O(n)\), where \(n\) is the number of XML elements in the corresponding XML of the input Process.

\[
\text{Algorithm 6 GetProcessResult(}P\text{rocess)}
\]

**Input**: The input Process;
**Output**: The set of possible results of Process;
**Variables**: Set \(R, Ri\);
1. if Fault has catchAll then
2. \(R = \{\text{OK}\}\);
3. else
4. \(Ri = \text{GetResult(}\text{Activity}\text{)}\);
5. \(F = \{n_1, ..., n_m\}\);
6. \(R = R \setminus F\);
7. if \(\#R < \#Ri\) then
8. \(R = R \cup \{\text{OK}\}\)
9. end
10. return \(Ri\);

- Algorithm 7 is used for getting the result of the input Activity according to the outside result. The maximum complexity of algorithm 7 is \(O(k)\), where \(k = \max\{1, n \ast j\}\), \(n\) is the number of XML elements in the corresponding XML of the input Activity, and if \(r\) is OK, then \(j = 0\), else \(j = 1\).

\[
\text{Algorithm 7 GetResult(}A, r\text{)}
\]

**Input**: The input Activity and the outside result \(r\);
**Output**: The actual result of Activity;
**Variables**: Set \(R, ri\);
1. if \(r = \text{OK}\) then
2. \(ri = \text{OK}\)
3. else
4. \(R = \text{GetResult(}\text{Activity}\text{)}\);
5. if \(r \in R\) then
6. \(ri = r\);
7. else
8. \(ri = \text{OK}\);
9. end
10. return \(ri\);

4 Verification

For ensuring the high confidence of a Web service system, we want to verify its BPEL description with respect to some critical properties. 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 substitutivity of BPEL processes can also be checked.

4.1 Verification Method

Based on the transformation, some temporal properties can be verified on protocol interface. The protocol property must be formed in \(a \rightarrow \varphi\), where \(\varphi\) is the formula in Action Set Computation Tree Logic (ASCTL) and \(a\) is a provided action.

**Definition 14** (Action Set Computation Tree Logic, ASCTL). The ASCTL formula \(\varphi\) over an action set \(A\) must meet the following syntactic rules, where \(D \subseteq A\).

\[
\chi ::= \text{true} \mid \text{false} \mid D \mid \neg\chi \mid \chi \land \chi' \\
\varphi ::= \text{true} \mid \text{false} \mid \neg\varphi \mid \varphi \land \varphi' \mid E_\gamma \mid A_\gamma \\
\gamma ::= [\varphi(\chi) \cup (\chi') \varphi'] \mid [\varphi(\chi) \cup (\chi') \varphi']
\]

Compared to Action Computation Tree Logic (ACTL) in Meolie et al. (2000), the syntax and semantics of ASCTL are same except the following differences:

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

In this section, we only give some key points of ASCTL semantics. The detailed semantics definitions can be referred to appendix.

- The semantics model is the LTS that does not contain any transition whose labeled action is internal action.
The γ in ASCTL formula ϕ is the path formula. Given a LTS model $M = (S, I, L, δ)$ and a path $p = s_1A_1s_2A_2s_3...$ of $M$, where $s_i ∈ S$, $A_i ∈ L$, $i$ is the natural number and $i ≥ 1$, the semantics of path formula can be given as follows.

$$p \models [ϕ](χ) \cup (ϕ')(χ') \iff \exists i ≥ 1, ∀ j ∈ [1, i) \bullet (A_i \models χ ∧ s_i \models ϕ)$$

$$p \models [ϕ](χ) \cup (ϕ')(χ') \iff p \models [ϕ](χ) \cup (ϕ')(χ') \vee (∀ i ≥ 1 \bullet (A_i \models χ ∧ s_i \models ϕ))$$

The until connective $U$ in ASCTL is strict until (Reynolds, 2003), from which the next operator $X$ can be derived.

Many other ASCTL operators can be derived from the basic ones, e.g. $EF$, $AF$, $EG$ and $AG$, and the derivations are same as Meolic et al. (2000).

### 5.1 Example 1

The BPEL script skeleton is shown in Figure 7. The process $p_1$ has a $scope$ activity $s_1$ in which $sequence$ activity $se$ contains five sequential activities. There is no explicit $faultHandlers$ for $s_1$, and the process has a $faultHandler$ that contains $catchAll$ handler only.

After translation, the protocol interface for the BPEL in Figure 7 is given as follows.

$$(p_1, e) \rightarrow_R (q_0, s_1, p_1), q_1, (q_1, s_1, e), \emptyset),$$
$$(s_1, e) \rightarrow_R (q_3, (se, e), \emptyset),$$
$$(se, e) \rightarrow_R (q_5, (receive, OK), q_6),$$
$$(q_6, (s_1, s_1), q_7), (q_7, (scope, s_1), OK), q_8),$$
$$(q_8, (s_1, s_1), q_{13}), (q_{13}, (scope, s_1), OK), q_{14}),$$
$$(q_{14}, τ, \emptyset),$$
$$(scope, s_1, OK) \rightarrow_R (q_9, (invoke, e, OK), q_{10}), (q_{10}, σ, \bot).$$

This section demonstrates the formalization and verification by two examples. The first example is used to exemplify the multi-scoped, default compensation and default fault handling translation. The second example is 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 the client.
volve the services of different airline companies according to the BPEL process behind the travel agency Web service. When a client requests for a travel reservation, the travel agency will invoke the airline, car rental, weather forecast service and do the assign activities in parallel. A successful result indicates that the reservation has been completed, the car rental service is contacted, and it provides the weather forecast of the client destination. After the reservation has been completed, the client will be notified of the successful completion.

According to the preceding 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 10. The solid arrow lines represent the sequential invocations. The middle big rectangle represents flow activity. The first activity in the flow is a switch activity. The dotted arrow lines represent links in the flow.

5.2 Example 2

The interaction architecture of the travel agency Web service is shown in Figure 9. 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 Web service begins to proceed the request. The travel agent will invoke the services of different airline companies according to the client destination, car rental company and weather forecast department to finish the reservation.

The provided action set is \{\{p1,e\}\}. From the above protocol interface, we can get the BPEL executions. After inspecting the protocol interface behaviour, we find that the fault handling behaviour of the catchAll handler in process is equal to an empty activity. The reason is that the scope s1 results in exception e and no successful completed scope is installed in the stack of process p1. In addition, the default faultHandler for the scope s1 will call the compensationHandler for s12 and s11 in sequence.

The meaning of the executable process is given as follows. After receiving the reservation request from a client, the travel agency will invoke the airline, car rental, weather forecast service and do the assign activities in parallel. According to the destination the client wants to go, the travel agent invokes different airline services. If client wants to travel to US, then usAirlines Web service will be invoked. If client wants to travel to Canada, then canadaAirlines Web service will be invoked, otherwise only britishAirlines Web service will be invoked. At the same time as the reservation takes place, the weather Web service is contacted, and it provides the weather forecast of the client destination. After the reservation has been completed, the car rental Web service might be invoked, and it happens only if the destination city 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 4, we can translate the BPEL description which has no compensation or fault handler into the following protocol interface. Because there is no compensation or fault handler, and there is no information about exceptions, no action is exception action. The provided action set D is \{(BookTravel,OK)\}.

\[
\begin{align*}
(BookTravel,OK) & \rightarrow (q_0, (bookTravel,OK), q_1), (q_1, 5, 1), \\
(TravelSeq,OK) & \rightarrow (q_2, (bookReceive,OK), q_3), (q_3, (flow1,OK), q_4), \\
& (q_4, (assign2,OK), q_5), (q_5, (BookReply,OK), 5), \\
& (flow1,OK) \rightarrow (q_7, 5\{branch1,OK\}, branch2,OK), \\
& (branch3,OK), (branch4,OK)), 5), \\
& (branch1,OK) \rightarrow (q_6, (switch1,OK), q_7), (q_7, 5, 1), \\
& (switch1,OK) \rightarrow (q_8, 5\{branch5,OK\}, branch6,OK), \\
& (branch7,OK)), 5), \\
& (branch5,OK) \rightarrow (q_9, (invokecc,OK), q_{10}).
\end{align*}
\]
Figure 8: BPEL specification of the travel agency service.

Figure 11: BPEL specification of the travel agency service with transaction.
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 new BPEL description is shown in Figure 11. 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 ticket. The new protocol interface is given as follows, and the provided action set $\mathcal{D}$ is \{(BookTravel,OK),(BookTravel,NOCAR)\}. It is necessary to point out that the translated orchestration process needs a new Web service for sending letters.

\[
\begin{align*}
\text{BookTravel,OK} & \rightarrow_{\tau} \langle q_{24}, \{\text{BookTravel}\}, \bot \rangle \\
\text{TravelSeq,NOCAR} & \rightarrow_{\tau} \langle q_{54}, \{\text{travelSeq,NOCAR}\}, \bot \rangle \\
\text{scope1,NOCAR} & \rightarrow_{\tau} \langle q_{55}, \{\text{scope1,NOCAR}\}, \bot \rangle \\
\text{invokeam,OK} & \rightarrow_{\tau} \langle q_{51}, \{\text{invokeam,OK}\}, q_{12} \rangle \\
\text{BookTravel,OK} & \rightarrow_{\tau} \langle q_{11}, \{\text{BookTravel}\}, \bot \rangle \\
\text{switch2,OK} & \rightarrow_{\tau} \langle q_{46}, \{\text{switch2,OK}\}, \bot \rangle \\
\text{branch3,OK} & \rightarrow_{\tau} \langle q_{47}, \{\text{branch3,OK}\}, q_{48} \rangle \\
\text{branch4,OK} & \rightarrow_{\tau} \langle q_{50}, \{\text{assign1,OK}\}, q_{51} \rangle \\
\text{assign1,OK} & \rightarrow_{\tau} \langle q_{52}, \{\text{assign2,OK}\}, \bot \rangle \\
\text{BookTravel,OK} & \rightarrow_{\tau} \langle q_{53}, \{\text{BookTravel}\}, \bot \rangle
\end{align*}
\]

After translation, some protocol properties can be verified on the travel agency Web service system. The protocol properties and the corresponding verification results are shown in Table 5. The meanings of the properties can be given as follows: 1) the success of travel reservation will always lead to the success of ticket reservation; 2) the success of travel reservation will always lead to the ticket reservation to the United States, and the reason of the property unsatisfying is that 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 the client for the booking failure resulted from the car rental failure; 4) the ticket return must not occur before the ticket reservation; 5) the apology must not occur before the ticket return to US airline company, and the reason of the property unsatisfying is same as that of the second property; 6) if the ticket will return to US airline company, the apology must not occur before it.
6 Related Work

There are several works on formalizing business transactions (Boci et al., 2003; Butler et al., 2005; Butler and Ripon, 2005; Bruni et al., 2005). Boci et al. (2003) propose $\pi$-calculus that extends asynchronous $\pi$-calculus with some transaction calculi. Butler et al. (2005) propose STAC languages to specify compensating business transactions. Butler and Ripon (2005) extend communicating sequential process (CSP) to enable the description of long-running transactions. Bruni et al. (2005) propose an enhanced Sagas language for specifying compensations in flow composition languages. All of these work use process algebra as fundamental theory. We share the ideas of the parallel exception synchronization policy in their work. Compared with these approaches, protocol interface is more flexible in its fault handling and compensation definition method, through which one can define not only the user-defined fault handling behaviour but also the invocation of the default fault handling behaviour.

There are many research on formalization and verification of BPEL, which can be divided by the underlying semantic theory as follows.

- **Petri nets**: Verbeek and van der Aalst (2005) use WF-nets (workflow nets) to formalize the BPEL description, and a mapping from BPEL process model to WF-net was proposed. Ouyang et al. (2005) present a mapping from BPEL constructs into Petri nets structures, and the mapping covers almost all the constructs in BPEL such as fault handling, compensation, link, etc.;

- **Process Algebra**: Foster et al. (2004) use Finite State Process (FSP) to formalize BPEL, and the specifications are verified on LTSA WS-Engineer, which can perform safety and liveness analysis, and interface compatibility checking. In Koshkina (2003), BPEL-Calculus is proposed to formalize BPEL, and the description can be verified on Concurreny WorkBench (CWB) by a syntax compiler plug-in for BPEL-Calculus;

- **Automata**: Fu et al. (2004) use guarded finite state automata (GVSA) to describe service composition and formalize BPEL, and the GVSA description can be translated to Promela which can be verified on SPIN. In Pistore et al. (2005), State Transition System (STS) is used to formalize the BPEL abstract process, and some requirement formulae formed in EaGLE can be verified on the STS model using the model checker NuSMV. In Yan et al. (2005), Discrete-Event Systems (DES) is used to formalize BPEL description for monitoring:

  - **Abstract State Machine (ASM)**: Fahland (2005) gives a complete abstract operational semantics for BPEL using ASM. The work in Fahland (2005) incorporates most elements of BPEL such as data handling, fault handling and compensation based on the work in Farahbod (2004).

Except for the work in Ouyang et al. (2005) and (Fahland, 2005; Farahbod, 2004), the above work is deficient in modeling transaction behaviour of orchestration. The multiple scope, fault handling and compensation features in BPEL are not well supported. These analysis or monitoring methods cannot deal with the transaction behaviour in BPEL. In this paper, we propose a generic underlying formalism that can deal with those BPEL constructs nicely as well as a corresponding verification method, and bridge the gap between BPEL and our formalism. The verification method is automatic and can effectively verify the transaction behaviour of BPEL description. Ouyang et al. (2005) do not take into account the default fault handling behaviour that can be nicely formalized by our formalism. Compared with the work in Fahland (2005), only the control flow in BPEL is considered in our work at the moment, and our approach provides the transaction support in the underlying formalism directly.

In addition, there are several works on the formalism customizing for BPEL (Butler et al., 2005; Pu et al., 2006; Qiu et al., 2005). Butler et al. (2005) extend STAC with indexed compensation task (StAC,) to formalize the named compensation behaviour. Qiu et al. (2005) propose a formal operational semantics for a subset of constructs in BPEL by an intermediate language. The work in Pu et al. (2006) extends the language in Qiu et al. (2005) to enable the semantic interpretation for the link activity in BPEL. Compared with these approaches, our approach is based on the Web service interface theory.

7 Conclusion and Future Work

This paper presents an improved formal foundation and a formalization and verification method for service orchestration in BPEL. Through the protocol interface, one can specify the nested transaction block, which can be weaved with the user-defined fault handling behaviour based on the ideas of AOP. Especially, the user-defined fault handling behaviour can also invoke the default compensation behaviour of the transaction block. With the protocol interface, a formalization of orchestration in BPEL is presented in a transformational way. For a Web service imple-

---

Table 5: The protocol properties and the corresponding verification results.

<table>
<thead>
<tr>
<th>Protocol Property</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BookTravel,OK) $\rightarrow$ AF[{(switch,OK)}]</td>
<td>True</td>
</tr>
<tr>
<td>(BookTravel,OK) $\rightarrow$ AF[{{apologize,OK}}]</td>
<td>False</td>
</tr>
<tr>
<td>(BookTravel,NOCAR) $\rightarrow$ AF[{{apologize,OK}}]</td>
<td>True</td>
</tr>
<tr>
<td>(BookTravel,NOCAR) $\rightarrow$ $\neg E$ [{{returnam,OK}}]</td>
<td>False</td>
</tr>
<tr>
<td>(BookTravel,NOCAR) $\rightarrow$ $\neg E$ [{(returnam,OK)}]</td>
<td>True</td>
</tr>
</tbody>
</table>


mented in BPEL, the description of invocation process can be translated into the Web service protocol interface automatically. Based on the formalism and transformation, protocol properties specified in ASCTL can be verified and a verification methodology is proposed to ensure the high confidence in development.

With the help of the Web service protocol interface and the translation algorithms, scope-based fault handling and compensations mechanism in BPEL can be formalized nicely and rigorously, 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 the service orchestration and choreography, and to improve our formalism for specifying more elements of BPEL such as data handling.

ACKNOWLEDGMENT

Supported by the National Natural Science Foundation of China under Grant No.60233020, 60673118, 90612009; the National High-Tech Research and Development Plan of China under Grant No.2005AA113130, 2006AA01Z429; the National Grand Fundamental Research 973 Program of China under Grant No.2005CB321802; Program for New Century Excellent Talents in University under Grant No.NCET-04-0996.

REFERENCES


Appendix: ASCTL Semantics

The ASCTL formula set over an action set $A$ can be given by the following grammar, where $D \subseteq A$.

$$\chi ::= \text{true} | \text{false} | D | \neg \chi | \chi \land \chi'$$

$$\varphi ::= \text{true} | \text{false} | \neg \varphi | \varphi \land \varphi' | E_{\gamma} | A_{\gamma}$$

$$\gamma ::= [\varphi(\chi) \cup \{\chi'\}\varphi'] \cup [\varphi(\chi) \cup \{\chi'\}\varphi']$$

The formula $\chi$ is action set formula, $\varphi$ is state formula and $\gamma$ is path formula. Given an LTS $M = (S, I, L, \delta)$, an action set $A \in L$, a state $s \in S$ and a path $p = s_1A_1s_2A_2s_3...$, the satisfaction of action formula $\chi$ by $A$ (denoted by $A \models \chi$), state formula $\varphi$ by $s$ (denoted by $s \models \varphi$) and path formula $\gamma$ by $p$ (denoted by $p \models \gamma$) is defined by the following semantic rules.

$$A \models \text{true} \quad \text{always}$$

$$A \models \text{false} \quad \text{never}$$

$$A \models D \quad \text{iff} \ A \cap D \neq \emptyset$$

$$A \models \neg \chi \quad \text{iff} \ A \not\models \chi$$

$$A \models \chi \land \chi' \quad \text{iff} \ A \models \chi \land A \models \chi'$$

$$s \models \text{true} \quad \text{always}$$

$$s \models \text{false} \quad \text{never}$$

$$s \models \neg \varphi \quad \text{iff} \ s \not\models \varphi$$

$$s \models \varphi \land \varphi' \quad \text{iff} \ s \models \varphi \land s \models \varphi'$$

$$s \models E_{\gamma} \quad \text{iff there exists a path}$$

$$p = s_1A_1s_2A_2s_3... \quad \text{where}$$

$s_1 = s \land p \models \gamma$

$p \models [\varphi(\chi) \cup \{\chi'\}\varphi'] \cup [\varphi(\chi) \cup \{\chi'\}\varphi']$

As mentioned in Section 5.1, the syntax and semantics of ASCTL are same as ACTL in Meolic et al. (2000) except few differences, the ASCTL derivations from the above operators and connectives can be referred to Meolic et al. (2000) too.