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: dong.wei@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.

Reference to this paper should be made as follows: Chen, Z.B., Wang, J., Dong, W. and Qi, Z.C. (2007) 'Interface Theory based Formalization and Verification of Orchestration in BPEL4WS', *Int. J. Business Process Integration and Management*, Vol. 1, Nos. 1/2/3, pp.64–74.

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 (Leymann, 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

Copyright © 200x Inderscience Enterprises Ltd.

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 advance 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

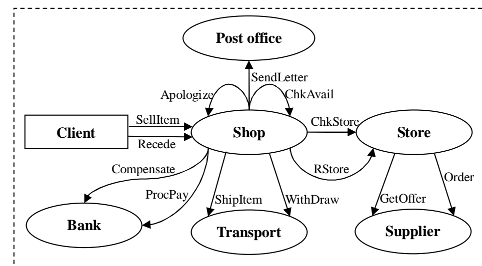


Figure 1: Supply chain management system.

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 $\langle SellItem, FAIL \rangle$ 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. $\langle SellItem, FAIL \rangle$ 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 $\mathcal{M}$ be a finite set of web methods, $\mathcal{O}$ be a finite set of outputs, $\mathcal{A} \subseteq \mathcal{M} \times \mathcal{O}$ 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 $\mathcal{S_C}$ and $\mathcal{S_F}$ that specify the fault handling and compensation behaviours. Signature interface is defined as follows.

Definition 1 (Signature Interface, SI)**.** *A signature interface $\mathcal{P}$ is a 4-tuple ($\mathcal{A}$, $\mathcal{S}$, $\mathcal{S_C}$, $\mathcal{S_F}$), where*

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

Signature interface describes the direct invocation relation of Web service interfaces. An action may have different *types*. An action $a \in \mathcal{A}$ is a *supported action* if $\mathcal{S}(a)$ is defined. A web method $m \in \mathcal{M}$ is a *supported method* if there exists a supported action $a = \langle m, o \rangle$. An action a is a *success action* if $\mathcal{S_C}(a)$ is defined. An action a is an *exception action* if $\mathcal{S_F}(a)$ 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:

$$\begin{aligned} required(a') = & (\exists a \in dom(\mathcal{S}).\ a' \in \mathcal{S}(a)) \vee \\ & (\exists a \in dom(\mathcal{S_C}).\ a' \in \mathcal{S_C}(a)) \vee \\ & (\exists a \in dom(\mathcal{S_F}).\ a' \in \mathcal{S_F}(a)). \end{aligned}$$

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.

$$\begin{gathered} \forall a \in \mathcal{A},\ \exists b \in \mathcal{A}.\ a = \langle m, o_1 \rangle \wedge required(a) \wedge b \in dom(\mathcal{S}) \\ \wedge b = \langle m, o_2 \rangle \Rightarrow a \in dom(\mathcal{S}), \\ \forall a, b \in dom(\mathcal{S_F}),\ \forall c \in dom(\mathcal{S_C}).\ a \notin \mathcal{S_F}(b) \wedge a \notin \mathcal{S_C}(c). \end{gathered}$$

Example 2.1 (Well-formed SI)**.** *The signature interface $\mathcal{P}_{shop}$=($\mathcal{A}_{shop}$, $\mathcal{S}_{shop}$, $\mathcal{S_C}_{shop}$, $\mathcal{S_F}_{shop}$) of the Web service* **Shop** *is defined as follows.*

$\mathcal{A}_{shop}$ = {
⟨SellItem,SOLD⟩,⟨SellItem,FAIL⟩,⟨ChkAvail,OK⟩,⟨ChkAvail,FAIL⟩,
⟨ProcPay,OK⟩,⟨ProcPay,FAIL⟩,⟨ShipItem,OK⟩,⟨ShipItem,FAIL⟩,
⟨ChkStore,OK⟩,⟨ChkStore,FAIL⟩,⟨Apologize,OK⟩,⟨SendLetter,OK⟩,
⟨Recede,OK⟩,⟨Compensate,OK⟩,⟨RStore,OK⟩,⟨WithDraw,OK⟩ }

$\mathcal{S}_{shop}$ = { ⟨**SellItem**, **SOLD**⟩ → {⟨ChkAvail,OK⟩,⟨ProcPay,OK⟩, ⟨ShipItem,OK⟩},
⟨**SellItem**, **FAIL**⟩ → {⟨ChkAvail,FAIL⟩,⟨ChkAvail,OK⟩, ⟨ProcPay,FAIL⟩,⟨ProcPay,OK⟩, ⟨ShipItem,FAIL⟩},
⟨**ChkAvail**, **OK**⟩ → {⟨ChkStore,OK⟩},
⟨**ChkAvail**, **FAIL**⟩ → {⟨ChkStore,FAIL⟩},
⟨**Apologize**, **OK**⟩ → {⟨SendLetter,OK⟩},
⟨**Recede**, **OK**⟩ → {⟨Compensate,OK⟩,⟨RStore,OK⟩, ⟨WithDraw,OK⟩}}

$\mathcal{S_C}_{shop}$ = { ⟨**SellItem**, **SOLD**⟩ → {⟨Recede,OK⟩},
⟨**ChkAvail**, **OK**⟩ → ∅, ⟨**Apologize**, **OK**⟩ → ∅,
⟨**Recede**, **OK**⟩ → ∅}

$\mathcal{S_F}_{shop}$ = { ⟨**SellItem**, **FAIL**⟩ → {⟨Apologize,OK⟩},
⟨**ChkAvail**, **FAIL**⟩ → ∅}

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 $\mathcal{P}_{shop}$ supports all the required actions whose methods are

supported methods and no exception action is invoked in compensation or fault handling, $\mathcal{P}_{shop}$ 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 $\mathcal{P}_1 = (\mathcal{A}_1, \mathcal{S}_1, \mathcal{S}_{\mathcal{C}1}, \mathcal{S}_{\mathcal{F}1})$ and $\mathcal{P}_2 = (\mathcal{A}_2, \mathcal{S}_2, \mathcal{S}_{\mathcal{C}2}, \mathcal{S}_{\mathcal{F}2})$, $\mathcal{P}_1$ and $\mathcal{P}_2$ are compatible if the following conditions are satisfied:*

- $dom(\mathcal{S}_1) \cap dom(\mathcal{S}_2) = \emptyset$;
- $\mathcal{P}_c = \mathcal{P}_1 \cup \mathcal{P}_2 = (\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{S}_{\mathcal{C}1} \cup \mathcal{S}_{\mathcal{C}2}, \mathcal{S}_{\mathcal{F}1} \cup \mathcal{S}_{\mathcal{F}2})$ *is well-formed.*

If two signature interfaces $\mathcal{P}_1$ and $\mathcal{P}_2$ are compatible(denoted by $comp(\mathcal{P}_1, \mathcal{P}_2)$), their composition (denoted by $\mathcal{P}_1 \parallel \mathcal{P}_2$) is $\mathcal{P}_c$. The composition operator is commutative and associative.

Example 2.2 (Compatibility of SI). *The signature interface $\mathcal{P}_{store}$=($\mathcal{A}_{store}$, $\mathcal{S}_{store}$, $\mathcal{S}_{\mathcal{C}store}$, $\mathcal{S}_{\mathcal{F}store}$) of the Web service* **Store** *is defined as follows. $\mathcal{P}_{store}$ is well-formed and compatible with $\mathcal{P}_{shop}$ in Example 2.1.*

$$\begin{aligned}
\mathcal{A}_{store} &= \{ \langle \text{ChkStore,OK} \rangle, \langle \text{ChkStore,FAIL} \rangle, \langle \text{RStore,OK} \rangle, \\
&\quad \langle \text{GetOffer,OK} \rangle, \langle \text{Order,OK} \rangle \} \\
\mathcal{S}_{store} &= \{ \langle \mathbf{ChkStore}, \mathbf{OK} \rangle \rightarrow \{ \langle \text{GetOffer,OK} \rangle, \langle \text{Order,OK} \rangle \}, \\
&\quad \langle \mathbf{ChkStore}, \mathbf{FAIL} \rangle \rightarrow \emptyset, \langle \mathbf{RStore}, \mathbf{OK} \rangle \rightarrow \emptyset \} \\
\mathcal{S}_{\mathcal{C}store} &= \{ \langle \mathbf{ChkStore}, \mathbf{OK} \rangle \rightarrow \{ \langle \text{RStore,OK} \rangle \}, \\
&\quad \langle \mathbf{RStore}, \mathbf{OK} \rangle \rightarrow \emptyset \} \\
\mathcal{S}_{\mathcal{F}store} &= \{ \langle \mathbf{ChkStore}, \mathbf{FAIL} \rangle \rightarrow \{ \langle \text{GetOffer,OK} \rangle, \langle \text{Order,OK} \rangle \} \}
\end{aligned}$$

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 $\mathcal{P}_1 = (\mathcal{A}_1, \mathcal{S}_1, \mathcal{S}_{\mathcal{C}1}, \mathcal{S}_{\mathcal{F}1})$ and $\mathcal{P}_2 = (\mathcal{A}_2, \mathcal{S}_2, \mathcal{S}_{\mathcal{C}2}, \mathcal{S}_{\mathcal{F}2})$, $\mathcal{P}_2$ refines $\mathcal{P}_1$ ($\mathcal{P}_2 \preccurlyeq \mathcal{P}_1$) if the following conditions are satisfied:*

- *for every $a \in \mathcal{A}$, if $\mathcal{P}_1$ supports a,then $\mathcal{P}_2$ supports a;*
- *for every $a \in \mathcal{A}$, if a is a success action in $\mathcal{P}_1$, then a is a success action in $\mathcal{P}_2$;*
- *for every $a \in \mathcal{A}$, if a is an exception action in $\mathcal{P}_1$, then a is an exception action in $\mathcal{P}_2$;*
- *for every $a, a' \in \mathcal{A}$, and $a \in dom(\mathcal{S}_1)$, if $a' \in \zeta_2(a)$, where $\zeta_2 \in \{\mathcal{S}_2, \mathcal{S}_{\mathcal{C}2}, \mathcal{S}_{\mathcal{F}2}\}$, then $a' \in \zeta_1(a)$;*
- *for every unsupported web method $m \in \mathcal{M}$ in $\mathcal{P}_2$, if $\langle m, o \rangle$ is a required action in $\mathcal{P}_2$, then $\langle m, o \rangle$ is a required action in $\mathcal{P}_1$.*

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 $\mathcal{P}_2$ is required by $\mathcal{P}_1$, they describe that $\mathcal{P}_2$ does not assume more actions which are supported by environment than $\mathcal{P}_1$. Given three signature interfaces $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$, if $comp(\mathcal{P}_1, \mathcal{P}_3)$, $comp(\mathcal{P}_2, \mathcal{P}_3)$, and $\mathcal{P}_2 \preccurlyeq \mathcal{P}_1$, then $\mathcal{P}_2 \parallel \mathcal{P}_3 \preccurlyeq \mathcal{P}_1 \parallel \mathcal{P}_3$.

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 $\mathcal{A}$ and a transaction block name set $\mathcal{B}_T$ is given by the following grammar ($a \in \mathcal{A}$, $\mathcal{B} \subseteq \mathcal{A}$, and $n, n_1, n_2 \in \mathcal{B}_T$):*

$$term :: \tau \mid \ell \mid a \mid \sqcup\mathcal{B} \mid \sqcap\mathcal{B} \mid \boxplus\mathcal{B} \mid (\!|n_1, n_2|\!) \mid \widehat{(\!|n|\!)}$$

The set of all terms over $\mathcal{A}$ and $\mathcal{B}_T$ is denoted by $Term(\mathcal{A}, \mathcal{B}_T)$. The term τ represents no action is needed to be invoked. The term ℓ 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 $\sqcup\mathcal{B}$ is a *choice term*, which represents a nondeterministic choice in the action set $\mathcal{B}$. The term $\sqcap\mathcal{B}$ is a *fork term*, which represents parallel invocations of all actions in $\mathcal{B}$, and the term waits for all actions to return. If any action fails or is an exception action, the term fails. The term $\boxplus\mathcal{B}$ is a *fork-choice term*, which represents parallel invocations of all actions in $\mathcal{B}$, whereas the return of any action will return the term. Only when all sides are exception actions, the term fails. $a_1\square...\square a_n = \square\{a_1, ..., a_n\}$, where $\square \in \{\sqcup, \sqcap, \boxplus\}$. 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 $\widehat{(\!|n|\!)}$ 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 Automata, CPA).

A coordination protocol automaton G is a 4-tuple $(\mathcal{A}, \mathcal{B}_T, \mathcal{L}, \delta)$, where

- *$\mathcal{A} \subseteq \mathcal{M} \times \mathcal{O}$ is a set of actions;*
- *$\mathcal{B}_T$ is a set of transaction block names;*
- *$\mathcal{L} \subseteq \mathcal{N} \times \{\varovee, \varowedge\}$ is a set of locations, where $\mathcal{N}$ is a set of location names, $\{\varovee, \varowedge\}$ is the location type set, and the default type of location is $\varovee$. $\langle \perp, \varovee \rangle \in \mathcal{L}$ is the return location, and $\langle \boxtimes, \varovee \rangle \in \mathcal{L}$ is the exception location;*
- *$\delta \subseteq \mathcal{L} \setminus \{\langle \perp, \varovee \rangle, \langle \boxtimes, \varovee \rangle\}) \times Term(\mathcal{A}, \mathcal{B}_T) \times \mathcal{L}$ is the transition relation set.*

We use $in(l) = \{t \mid t \in \delta \wedge t = (l_s, term, l)\}$ to represent all the transitions with the same target location l, and $out(l) = \{t \mid t \in \delta \wedge t = (l, term, l_t)\}$ 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 $\varovee$, then the transitions in $out(l)$ can be taken if there exists one transition in $in(l)$ that has been taken before. If the type of the location l is $\varowedge$, then the transitions in $out(l)$ can be taken if all transitions in $in(l)$ have been taken before. A location is terminating in *CPA* if there exists a trace starting from the location and ending with $\langle \perp, \varovee \rangle$ or $\langle \boxtimes, \varovee \rangle$. As a shorthand, we use location name to denote the location whose type is $\varovee$. For example, $\perp$ is used to denote $\langle \perp, \varovee \rangle$. Based on *CPA*, the protocol interface can be defined as follows.

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

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

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 $\mathcal{T} = (G, \mathcal{D}, \mathcal{R}, \mathcal{R_C}, \mathcal{R_F})$, the underlying signature interface of $\mathcal{T}$ (denoted by $psi(\mathcal{T})$) is $(\mathcal{A}_s, \mathcal{S}, \mathcal{S_C}, \mathcal{S_F})$, where $\mathcal{A}_s = \mathcal{A}$; $\mathcal{S}(a) = sigl(\mathcal{R}(a))$ if $\mathcal{R}(a)$ is defined, otherwise $\mathcal{S}(a)$ is undefined; $\mathcal{S_C}(a) = sigl(\mathcal{R_C}(a))$ if $\mathcal{R_C}(a)$ is defined, otherwise $\mathcal{S_C}(a)$ is undefined; $\mathcal{R_F}(a) = sigl(\mathcal{R_F}(a))$ if $\mathcal{R_F}(a)$ is defined, otherwise $\mathcal{S_F}(a)$ is undefined. The function $sigl : \mathcal{L} \to 2^{\mathcal{A}}$ is defined as follows:

$$sigl(\langle \perp, \varovee \rangle) = \emptyset,\ sigl(\langle \boxtimes, \varovee \rangle) = \emptyset,$$
$$sigl(q) = \bigcup_{\exists (q, term, q') \in \delta \wedge term \neq \ell} \varphi(term) \cup sigl(q'),$$
$$\varphi(\tau) = \emptyset, \varphi(a) = \{a\},\ \varphi(\Box \mathcal{B}) = \mathcal{B}, \text{ where } \Box \in \{\sqcup, \sqcap, \boxplus\},$$
$$\varphi(\llparenthesis n_1, n_2 \rrparenthesis) = \emptyset,\ \varphi(\llparenthesis \widehat{n} \rrparenthesis) = \emptyset.$$

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

Example 2.3 (Well-formed Protocol Interface). *The protocol interface $\mathcal{T}_{shop} = (G_{shop}, \mathcal{D}_{shop}, \mathcal{R}_{shop}, \mathcal{R_C}_{shop}, \mathcal{R_F}_{shop})$ of the Web service* **Shop** *is defined as follows. As a shorthand, we use the set whose elements are formed in $(l, term, l')$ to represent the transition relation set in CPA G_{shop}, where l and l' are the locations in G_{shop}. The partial functions $\mathcal{R}, \mathcal{R_C}, \mathcal{R_F}$ 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 ($\mathcal{A}_{shop}$ is the same as that in Example 2.1).*

$$
\begin{array}{l}
\{ \\
\langle \textbf{SellItem}, \textbf{SOLD} \rangle \to_{\mathcal{R}} (q_0, \llparenthesis \text{SellItem, CheckAvail} \rrparenthesis, q_1), \\
\quad (q_1, \langle \text{ChkAvail,OK} \rangle, q_2),\ (q_2, \llparenthesis \text{SellItem, ProcPay} \rrparenthesis, q_3), \\
\quad (q_3, \langle \text{ProcPay,OK} \rangle, q_4),\ (q_4, \llparenthesis \text{SellItem, ShipItem} \rrparenthesis, q_5), \\
\quad (q_5, \langle \text{ShipItem,OK} \rangle, \perp), \\
\langle \textbf{SellItem}, \textbf{FAIL} \rangle \to_{\mathcal{R}} (q_6, \llparenthesis \text{SellItem, CheckAvail} \rrparenthesis, q_7), \\
\quad (q_7, \langle \text{ChkAvail,FAIL} \rangle, \boxtimes), (q_6, \llparenthesis \text{SellItem, CheckAvail} \rrparenthesis, q_8), \\
\quad (q_8, \langle \text{ChkAvail,OK} \rangle, q_9), (q_9, \llparenthesis \text{SellItem, ProcPay} \rrparenthesis, q_{10}), \\
\quad (q_{10}, \langle \text{ProcPay,FAIL} \rangle, \boxtimes),\ (q_9, \llparenthesis \text{SellItem, ProcPay} \rrparenthesis, q_{11}) \\
\quad (q_{11}, \langle \text{ProcPay,OK} \rangle, q_{12}),\ (q_{12}, \llparenthesis \text{SellItem, ShipItem} \rrparenthesis, q_{13}) \\
\quad (q_{13}, \langle \text{ShipItem, FAIL} \rangle, \boxtimes), \\
\langle \textbf{ChkAvail}, \textbf{OK} \rangle \to_{\mathcal{R}} (q_{14}, \langle \text{ChkStore,OK} \rangle, \perp), \\
\langle \textbf{ChkAvail}, \textbf{FAIL} \rangle \to_{\mathcal{R}} (q_{15}, \langle \text{ChkStore,FAIL} \rangle, \boxtimes), \\
\langle \textbf{Apologize}, \textbf{OK} \rangle \to_{\mathcal{R}} (q_{16}, \langle \text{SendLetter,OK} \rangle, \perp), \\
\langle \textbf{Recede}, \textbf{OK} \rangle \to_{\mathcal{R}} (q_{17}, \langle \text{WithDraw,OK} \rangle, q_{18}), \\
\qquad (q_{18}, \langle \text{Compensate,OK} \rangle, q_{19}), (q_{19}, \langle \text{RStore,OK} \rangle, \perp), \\
\langle \textbf{SellItem}, \textbf{SOLD} \rangle \to_{\mathcal{R_C}} (q_{20}, \langle \text{Recede,OK} \rangle, \perp), \\
\langle \textbf{ChkAvail}, \textbf{OK} \rangle \to_{\mathcal{R_C}} (q_{21}, \llparenthesis \widehat{ChkAvail} \rrparenthesis, \perp), \\
\langle \textbf{Recede}, \textbf{OK} \rangle \to_{\mathcal{R_C}} \perp, \langle \textbf{Apologize}, \textbf{OK} \rangle \to_{\mathcal{R_C}} \perp \\
\langle \textbf{ChkAvail}, \textbf{FAIL} \rangle \to_{\mathcal{R_F}} (q_{22}, \llparenthesis \widehat{ChkAvail} \rrparenthesis, \boxtimes), \\
\langle \textbf{SellItem}, \textbf{FAIL} \rangle \to_{\mathcal{R_F}} (q_{23}, \llparenthesis \widehat{SellItem} \rrparenthesis, q_{24}), (q_{24}, \langle \text{Apologize,OK} \rangle, \boxtimes) \\
\}
\end{array}
$$

$\mathcal{T}_{shop}$ models the interface behaviour of **Shop**, and $psi(\mathcal{T}_{shop})$ is the signature interface $\mathcal{P}_{shop}$ in Example 2.1. Because $\mathcal{P}_{shop}$ is well-formed and $\mathcal{T}_{shop}$ satisfies all the other conditions, $\mathcal{T}_{shop}$ is well-formed. Action $\langle$**SellItem**,**SOLD**$\rangle$ can invoke three transaction blocks in sequence. Action $\langle$**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 $\langle$**SellItem**,**FAIL**$\rangle$ will first invoke the default compensation for transaction block *SellItem* by $\llparenthesis \widehat{SellItem} \rrparenthesis$, after which action $\langle$**Apologize**,**OK**$\rangle$ will be invoked to send the apologetic letter. The provided action set $\mathcal{D}_{shop}$ is $\{\langle SellItem, OK \rangle, \langle SellItem, FAIL \rangle\}$.

Definition 7 (PI Compatibility). *Given two protocol interfaces $\mathcal{T}_1 = (G_1, \mathcal{D}_1, \mathcal{R}_1, \mathcal{R}_{\mathcal{C}1}, \mathcal{R}_{\mathcal{F}1})$ and $\mathcal{T}_2 = (G_2, \mathcal{D}_2, \mathcal{R}_2, \mathcal{R}_{\mathcal{C}2}, \mathcal{R}_{\mathcal{F}2})$, $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible if the following conditions are satisfied:*

- *$psi(\mathcal{T}_1)$ and $psi(\mathcal{T}_2)$ are compatible;*
- *$\mathcal{L}_1 \cap \mathcal{L}_2 = \{\langle \bot, \ovee \rangle, \langle \boxtimes, \ovee \rangle\}$;*
- *$\mathcal{B}_{T_1} \cap \mathcal{B}_{T_2} = \emptyset$;*
- *$\mathcal{T}_c = \mathcal{T}_1 \cup \mathcal{T}_2 = (G_1 \cup G_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{R}_{\mathcal{C}1} \cup \mathcal{R}_{\mathcal{C}2}, \mathcal{R}_{\mathcal{F}1} \cup \mathcal{R}_{\mathcal{F}2})$ is well-formed, where $G_1 \cup G_2 = (\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{B}_{T_1} \cup \mathcal{B}_{T_2}, \mathcal{L}_1 \cup \mathcal{L}_2, \delta_1 \cup \delta_2)$.*

If $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible (denoted by $comp(\mathcal{T}_1, \mathcal{T}_2)$), their composition (denoted by $\mathcal{T}_1 \parallel \mathcal{T}_2$) is $\mathcal{T}_c$. 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.

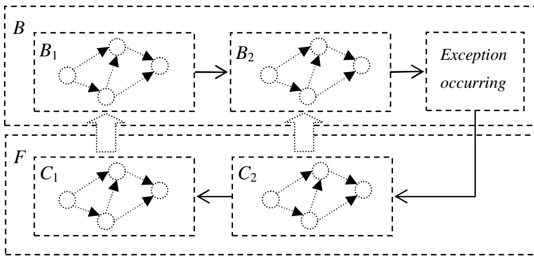


Figure 2: Long-running transaction example.

Intuitively, the invocation process is pushdown, and the process can continue only after the completion of every invoked action. The sequence of compensation should be reverse of the sequence of the previous invocations, so the recorded successful transaction block 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 group to interpret protocol interface execution.

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

We use ρ 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 pj to denote $p \cdot j$. The set of leaf nodes of tree t is $leaf(t) = \{p \in dom(t) \mid \forall j \in \mathbb{N}, pj \notin dom(t)\}$. For a node p in $dom(t)$, $child(p) = \{q \mid \exists j \in \mathbb{N}, q = pj \wedge q \in dom(t)\}$, and $parent(p) = \{q \mid \exists j \in \mathbb{N}, p = qj \wedge q \in dom(t)\}$. Let $\mathbb{T}(\mathcal{L})$ 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 $s(m) : \mathbb{N} \to \mathcal{L}$, where $\mathbb{N}$ is the natural number set, and $dom(s(m)) = \{n \mid n < m \wedge n \in \mathbb{N}\}$.*

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

We use $\mathbb{G}(N_s, \mathcal{L})$ to denote all stack groups of a finite set of names N_s over a finite label set $\mathcal{L}$. Given a protocol interface $\mathcal{T} = (G, \mathcal{D}, \mathcal{R}, \mathcal{R}_{\mathcal{C}}, \mathcal{R}_{\mathcal{F}})$, its semantics is defined by a *labeled transition system.* The set of states is $\mathbb{T}(Q_t) \times \mathbb{G}(\mathcal{B}_T, Term(\mathcal{A}, \mathcal{B}_T)) \times 2^{\delta}$, that is, the Cartesian products of trees over $Q_t = Q \times \mathcal{A}^* \times \wp \times \mathcal{B}_T$, stack groups over $\mathcal{B}_T$ and $Term(\mathcal{A}, \mathcal{B}_T)$, and power set of transition set δ, where Q is the location set of the *coordination protocol automaton* G, $\mathcal{A}$ is the action set in G, $\wp = \{\circ, \boxplus, \odot\}$ is the node type set, $\mathcal{B}_T$ is the transaction block name set in G, and δ is the transition set in G. The *underlying transition relation* of $\mathcal{T}$ is a transition relation $(\mathbb{T}(Q_t) \times \mathbb{G}(\mathcal{B}_T, Term(\mathcal{A}, \mathcal{B}_T)) \times 2^{\delta}) \times 2^{\mathcal{A} \cup \{ret, exp, cpb\}} \times (\mathbb{T}(Q_t) \times \mathbb{G}(\mathcal{B}_T, Term(\mathcal{A}, \mathcal{B}_T)) \times 2^{\delta})$. The label of state transition is the set of elements from $\mathcal{A} \cup \{ret, exp, cpb\}$. We write $\nu \xrightarrow{\mathcal{B}} \nu'$ for $(\nu, \mathcal{B}, \nu') \in \to_{\mathcal{T}}$, where $\nu = (t, g, s)$ and $\mathcal{B} \subseteq \mathcal{A} \cup \{ret, exp, cpb\}$. 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 $a = \langle m, o \rangle$, the initial state is $\nu_{inital} = (t_{inital}, g_{inital}, s_{intial}) = (\{(\rho, (\mathcal{R}(a), a, \circ, m))\}, \{(m, s_m(0))\}, \emptyset)$. 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, β, n) in Q_t, and if $w = \rho$, $q\beta n$ is used to represent it. For example, $q \boxplus n$ represents $(q, \rho, \boxplus, n)$;
- $\delta(q) = (term, q')$ denotes that there exists a transition $(q, term, q')$ in the *coordination protocol automaton* , and the transition can be taken now, which means that if the type of q is $\owedge$, then $in(q) \subseteq s$;

- if action c is supported by $\mathcal{R}$, $q_c = \mathcal{R}(c)$, otherwise $q_c = \bot$;
- if a is supported by $\mathcal{R}_{\mathcal{C}}$, $\mu(a) = \mathcal{R}_{\mathcal{C}}(a)$, otherwise if a is supported by $\mathcal{R}_{\mathcal{F}}$, $\mu(a) = \mathcal{R}_{\mathcal{F}}(a)$, otherwise $\mu(a) = \bot$;
- when applying all rules, if the transition $(q, term, q')$ is taken and the type of q is $\oslash$, 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 $\delta(q) = (\ell, q')$, then $t' = (t \setminus \{(p, q \circ n)\}) \cup \{(p, \bot \circ n)\}$, $g' = g$, and $s' = s \cup \{(q, \ell, q')\}$.

(**Pushdown**) $\nu \xrightarrow{\mathcal{M}} \nu'$
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 $t' = (t \setminus \{(p, q \circ n)\}) \cup \{(p, q' \circ n), (p0, q_a \circ n)\}$, and $\mathcal{M} = \{a\}$;
- $r = \sqcup\mathcal{B}$, then $t' = (t \setminus \{(p, q \circ n)\}) \cup \{(p, q' \circ n), (p0, q_c \circ n)\}$, where $c \in \mathcal{B}$, and $\mathcal{M} = \{c\}$;
- $r = \sqcap\mathcal{B}$, and $\mathcal{B} = \{a_0, ..., a_n\}$, then $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\circ), (p0, q_{a_0}\circ), ..., (pn, q_{a_n}\circ)\}$, and $\mathcal{M} = \mathcal{B}$;
- $r = \boxplus\mathcal{B}$, and $\mathcal{B} = \{a_0, ..., a_n\}$, then $t' = (t \setminus \{(p, q\circ)\}) \cup \{(p, q'\boxplus), (p0, q_{a_0}\circ), ..., (pn, q_{a_n}\circ)\}$, and $\mathcal{M} = \mathcal{B}$.

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

(**TB-Begin**) $\nu \xrightarrow{\mathcal{M}} \nu'$
If there exists a node p such that $p \in leaf(t)$, $t(p) = q \circ n$, $\delta(q) = (\llparenthesis n_1, n \rrparenthesis, q_1)$, and $\delta(q_1) = (term, q')$, then $t' = (t \setminus \{(p, q \circ n)\}) \cup \{(p, q' \circ n), (p0, q_c(c) \circ n_1)\}$, $\mathcal{M} = \{c\}$, $g' = g \cup \{(n_1, s_{n_1}(0))\}$, and $s' = s \cup \{(q, \llparenthesis n_1, n \rrparenthesis, q_1), (q_1, term, q')\}$, where $c = a$ if $term = a$, and $c \in \mathcal{B}$ if $term = \sqcup\mathcal{B}$.

(**CP-Begin**) $\nu \xrightarrow{\{cpb\}} \nu'$
If there exists a node p such that $p \in leaf(t)$, $t(p) = q \circ n$, $\delta(q) = (\widehat{\llparenthesis n_1 \rrparenthesis}, q')$, and $g(n_1)$ is defined:

- if $Len(g(n_1)) = 0$, then $t' = (t \setminus \{(p, q \circ n)\}) \cup \{(p, q' \circ n)\}$, and $g' = g$;
- if $Len(g(n_1)) > 0$, then $a = Pop(g(n_1))$, and $t' = (t \setminus \{(p, q \circ n)\}) \cup \{(p, q' \circ n), (p0, \bot \odot n_1), (p00, \mu(a) \circ n_1)\}$.

In all conditions, $s' = s \cup \{(q, \widehat{\llparenthesis n_1 \rrparenthesis}, q')\}$.

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

- $\alpha = \circ$, then $t' = t \setminus \{(p\theta, \bot(w) \circ n_1)\}$;
- $\alpha = \boxplus$, then $t' = (t \setminus \{(pp', res) \mid p' \in \mathbb{N}^* \wedge res = t(pp')\}) \cup \{(p, q \circ n_2)\}$;
- $\alpha = \odot$, and $Len(g(n_2)) > 0$, then $a = Pop(g(n_2))$, $t' = (t \setminus \{(p\theta, \bot(w) \circ n_1)\}) \cup \{(p\theta, \mu(a) \circ n_1)\}$;
- $\alpha = \odot$, and $Len(g(n_2)) = 0$, then $t' = t \setminus \{(p\theta, \bot(w) \circ n_1), (p, q\alpha n_2)\}$, and $g = g \setminus \{(n_2, g(n_2))\}$.

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

(**Exception**) $\nu \xrightarrow{\{exp\}} \nu'$
If there exists a node $p\theta$ such that $p\theta \in leaf(t)$, $t(p\theta) = \boxtimes(w) \circ n_1$, $t(p) = q(w_1)\alpha n_2$, where $\theta \in \mathbb{N}$:

- $\alpha = \circ$ and $w = \rho$, then $t' = (t \setminus \{(pp', res) \mid p' \in \mathbb{N}^* \wedge res = t(pp')\}) \cup \{(p, \boxtimes(w_1) \circ n_2)\}$; if $p = \rho$ and $w = \rho$, then $t' = (t \setminus \{(pp', res) \mid p' \in \mathbb{N}^* \wedge res = t(pp')\}) \cup \{(p, \mu(w_1) \circ n_2)\}$;
- $n_1 \neq n_2$ and $w \neq \rho$, then $t' = (t \setminus \{(p\theta, \boxtimes(w) \circ n_1)\}) \cup \{(p\theta, \mu(w) \circ n_1)\}$;
- $\alpha = \boxplus$ and $w = \rho$, then if all nodes in $\{p' \mid p' \in child(p) \wedge p' \neq p\theta\}$ have no child, and the locations of all nodes are same to $\boxtimes$, then $t' = (t \setminus \{(pp', res) \mid p' \in \mathbb{N}^* \wedge res = t(pp')\}) \cup \{(p, \boxtimes(w_1) \circ n_2)\}$.

In all conditions, $g' = g$.

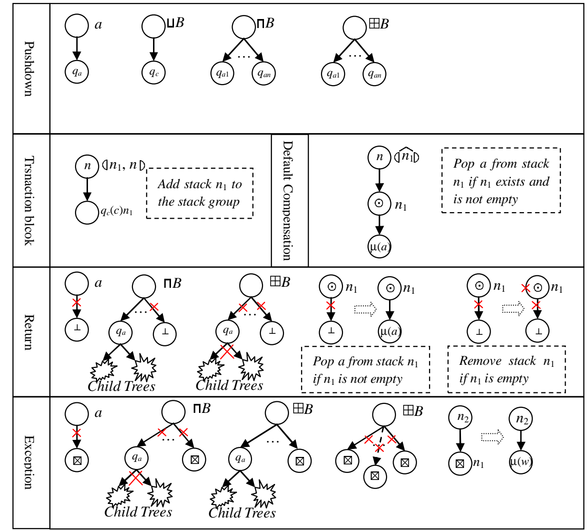


Figure 3: Operation diagrams of transition rules .

Figure 3 shows the operation diagrams of the transition rules, and the main ideas can be explained as follows.

- The operations in transition rules 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 G decides on the invoked terms.
- The terms a and $\sqcup\mathcal{B}$ lead to pushing down the leaf node. The terms $\sqcap\mathcal{B}$ and $\boxplus\mathcal{B}$ lead to branching the leaf node. These are reflected in rule (**Pushdown**).
- The rule (**TB-Begin**) depicts the beginning of a transaction block. It needs a new stack for recording the execution of the transaction block.
- The rule (**CP-Begin**) depicts the beginning of compensation for a transaction block. If the compensation stack exists and the block has some successful completed child blocks, the tree will be pushed down with a compensation node whose type is $\odot$.
- The reaching of $\bot$ or $\boxtimes$ leads to cutting nodes; The return of a *fork-choice* child action will cut all the other child branches. The return of transaction block

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 $\nu_0, \mathcal{A}_0, \nu_1, \mathcal{A}_1, ...$, where $\nu_i \xrightarrow{\mathcal{A}_i} \nu_{i+1}$ for all $i \in \{0, 1, ..., n, ...\}$. 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, Δ), 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 $\mathcal{T}$ and a provided action $a = \langle m, o\rangle \in \mathcal{D}$, the interface behaviour invoked by a can be transformed into a *labeled transition system*, which is denoted by $LTS(\mathcal{T}, a)$. The transformation procedure of $LTS(\mathcal{T}, a) = (S_a, I_a, L_a, \Delta_a)$ can be given as follows:

- $S_a = \mathbb{T}(Q_t) \times \mathbb{G}(\mathcal{B}_T, Term(\mathcal{A}, \mathcal{B}_T)) \times 2^{\delta}$;
- $I_a = (\{(\rho, (\mathcal{R}(a), \rho, \circ, m))\}, \{(m, s_m(0))\}, \emptyset)$;
- $L_a = 2^{\mathcal{A}\cup\{ret,exp,cpb\}}$;
- Δ_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 to the environment. The simulation relation of the underlying labeled transition systems can be extended to relax the conditions that substitutivity should satisfy. We denote $(s_1, a, s_1') \in \Delta$ as $s_1 \rightarrow_a s_1'$. If $t = a_1a_2...a_n \in L^*$, $s_1 \rightarrow_{a_1}\rightarrow_{a_2} ... \rightarrow_{a_n} s_1'$ is denoted as $s_1 \rightarrow_t s_1'$.

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 $\preccurlyeq_W \subseteq S_1 \times S_2$ such that:*

- *for every $s_1 \in M_1, s_2 \in M_2$, if $s_2 \preccurlyeq_W s_1$, then for every $(s_2, a, s_2') \in \Delta_2$, there exists $s_1 \rightsquigarrow_a s_1'$ in Δ_1, such that $s_2' \preccurlyeq_W s_1'$, where $s_1 \rightsquigarrow_a s_1'$ represents $s_1 \rightarrow_t s_1'$, in which $t = a_1a_2...a_n$, 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 $s_2 \preccurlyeq_W s_1$.*

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

Definition 13 (PI Substitutivity). *Given two protocol interfaces $\mathcal{T}_1 = (G_1, \mathcal{D}_1, \mathcal{R}_1, \mathcal{R}_{\mathcal{C}1}, \mathcal{R}_{\mathcal{F}1})$ and $\mathcal{T}_2 = (G_2, \mathcal{D}_2, \mathcal{R}_2, \mathcal{R}_{\mathcal{C}2}, \mathcal{R}_{\mathcal{F}2})$, $\mathcal{T}_2$* refines *$\mathcal{T}_1$ ($\mathcal{T}_2 \preccurlyeq \mathcal{T}_1$) if the following conditions are satisfied:*

- *$psi(\mathcal{T}_2) \preccurlyeq psi(\mathcal{T}_1)$;*
- *for every $a \in \mathcal{D}_1$, $LTS(\mathcal{T}_2, a)$ is weakly simulated by $LTS(\mathcal{T}_1, a)$ over $2^{\{ret,exp,cpb\}}$.*

Given three protocol interfaces $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$, if $comp(\mathcal{T}_1, \mathcal{T}_3)$, $comp(\mathcal{T}_2, \mathcal{T}_3)$, and $\mathcal{T}_2 \preccurlyeq \mathcal{T}_1$, then $\mathcal{T}_2 \parallel \mathcal{T}_3 \preccurlyeq \mathcal{T}_1 \parallel \mathcal{T}_3$.

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 *bpel* 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.

$$
\begin{aligned}
Process &::= process(n, Activity, Fault, Comp)\\
Activity &::= receive(n)\\
&\mid invoke(n)\\
&\mid reply(n)\\
&\mid assign(n)\\
&\mid throw(n)\\
&\mid compensate(n)
\end{aligned}
$$

$$
\begin{array}{lll}
 & & \mid empty \\
 & & \mid sequence(n, Activity_1, ..., Activity_m) \\
 & & \mid switch(n, Activity_1, ..., Activity_m) \\
 & & \mid flow(n, Activity_1, ..., Activity_m) \\
 & & \mid link(n, Activity_1, Activity_2) \\
 & & \mid while(n, Activity) \\
 & & \mid scope(n, Activity, Fault, Comp) \\
Comp & ::= & compensationHandler(Activity) \\
Fault & ::= & faultHandlers(catch_1(n_1, Activity_1), ..., \\
 & & \qquad catch_m(n_m, Activity_m)) \\
 & & \mid faultHandlers(catch_1(n_1, Activity_1), ..., \\
 & & \qquad catch_m(n_m, Activity_m), \\
 & & \qquad catchAll(Activity_a))
\end{array}
$$

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 *receive*, *invoke* and *reply* activities need environment support and *assign* is internal activity. Because data handing is omitted in this paper, *assign* is simply mapped the action whose target location of $\mathcal{R}$ is $\perp$. *throw* activity indicates that there is an exception occurred in the process, and it is mapped to a transition whose target location is $\boxtimes$. *compensate* activity indicates that the compensation for scope n is needed. According to the specification of BPEL, if n is the current enclosing scope, the *compensate* activity will invoke the default compensation behaviour of the current transaction block; if n is the enclosed scope, the corresponding action is mapped to the compensation start location for the successful scope action. *empty* activity is mapped to the transition whose term is τ.

BPEL	Translation Procedure
$receive(n)$	$\delta = \delta \cup \{(q, \langle receive_n, OK\rangle, q')\}$
$invoke(n)$	$\delta = \delta \cup \{(q, \langle invoke_n, OK\rangle, q')\}$
$reply(n)$	$\delta = \delta \cup \{(q, \langle reply_n, OK\rangle, q')\}$
$assign(n)$	$\delta = \delta \cup \{(q, \langle assign_n, OK\rangle, q')\}$ $\mathcal{R} = \mathcal{R} \cup \{(\langle assign_n, OK\rangle, \perp)\}$
$throw(n)$	$\delta = \delta \cup \{(q, \tau, \boxtimes)\}$
$compensate(n)$	$if\ n \neq s\ then$ $\delta = \delta \cup \{(q, \langle compensate_n, OK\rangle, q'\}$ $\mathcal{R} = \mathcal{R} \cup \{(\langle compensate_n, OK\rangle, \mathcal{R}_{\mathcal{C}}(\langle scope_n, OK\rangle))\}$ end $if\ n = s\ then$ $\delta = \delta \cup \{(q, \widehat{(\!|n|\!)}, q'\}$
$empty$	$\delta = \delta \cup \{(q, \tau, q')\}$

Table 1: Translating the basic BPEL activities.

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.

links in *flow* activities are used for coordination between parallel activities. The *joinCondition* describes different join policies. According to the *link* semantics in BPEL, we can use the *coordination* action and the 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, we can map the target activity to a transition whose start location type is $\oslash$. When the source activity ends, it should invoke a fork term specifying the start *links*, which will invoke a *coordination* action ℓ and reach the the target location. The *parent* used in the Table 2 denotes the parent activity of the current BPEL activity. Because we are only concerned with the control flow, the *joinCondition* of the target activity and the *DPE* mechanism are omitted. The translation procedure will check whether the translated activity has *link* information, and when *link* is translated is determined by its parent type.

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-

BPEL	Translation Procedure
sequence	$r' = GetResult(sequence, r)$ $\delta = \delta \cup \{(q, \langle sequence_n, r'\rangle, q')\}$ $\mathcal{R} = \mathcal{R} \cup \{(\langle sequence_n, r'\rangle, q_0)\}$ $q_1 = translate(Activity_1, q_0, s, r')$ $q_2 = translate(Activity_2, q_1, s, r')$ $\dots$ $q_k = translate(Activity_k, q_{k-1}, s, r')$
switch	$r' = GetResult(switch, r)$ $\delta = \delta \cup \{(q, \langle switch_n, r'\rangle, q')\}$ $\mathcal{R} = \mathcal{R} \cup \{(\langle switch_n, r'\rangle, q_0)\}$ $a_1 = \langle branch_1, GetResult(Activity_1, r')\rangle$ $translate(Activity_1, q_1, s, r')$ $\dots$ $a_k = \langle branch_k, GetResult(Activity_k, r')\rangle$ $translate(Activity_k, q_k, s, r')$ $\delta = \delta \cup \{(q_0, \sqcup\{a_1, ..., a_k\}, q_0')\}$
flow	$r' = GetResult(flow, r)$ $\delta = \delta \cup \{(q, \langle flow_n, r'\rangle, q')\}$ $\mathcal{R} = \mathcal{R} \cup \{(\langle flow_n, r'\rangle, q_0)\}$ $a_1 = \langle branch_1, GetResult(Activity_1, r')\rangle$ $translate(Activity_1, q_1, s, r')$ $\dots$ $a_m = \langle branch_m, GetResult(Activity_m, r')\rangle$ $translate(Activity_m, q_m, s, r')$ $\delta = \delta \cup \{(q_0, \sqcap\{a_1, ..., a_m\}, q_0')\}$
while	$r' = GetResult(while, r)$ $\delta = \delta \cup \{(q, \langle while_n, r'\rangle, q')\}$ $\mathcal{R} = \mathcal{R} \cup \{(\langle while_n, r'\rangle, q_0)\}$ $translate(Activity, q_0, s, r')$ $\delta = \delta \cup \{(q', \tau, q)\}$
link $(source)$ $\wedge$ *paernt* $\neq$ *sequence*	$if\ q' \neq \perp \wedge q' \neq \boxtimes\ then$ $\quad \delta = \delta \cup \{(q', \sqcap\{..., \langle link_i, OK\rangle, ...\}, \perp)\}$ $else$ $\quad \delta = (\delta \setminus \{(q, a, q')\}) \cup \{(q, a, q_{t_1})\}$ $\quad \delta = \delta \cup \{(q_{t_1}, \sqcap\{..., \langle link_i, OK\rangle, ...\}, q')\}$ end $\mathcal{R} = \mathcal{R} \cup \{(\langle link_i, OK\rangle, q_{l_i})\}$ $\delta = \delta \cup \{(q_{l_i}, \ell, q_t)\}$ $L = L \cup \{\langle Activity_1^i, (q_{l_i}, \ell, q_t)\rangle\}\ \dots$
link $(source)$ $\wedge$ *paernt* $=$ *sequence*	$r' = GetResult(Activity, r)$ $\delta = \delta \cup (q, \langle n_{link}, r'\rangle, q')$ $\mathcal{R} = \mathcal{R} \cup \{\langle n_{link}, r'\rangle, q_l\}$ $q_l' = translate(Activity_w, q_l, s, r)$ $Activity_w$ is the $Activity$ without source *links* $if\ q_l' \neq \perp \wedge q_l' \neq \boxtimes\ then$ $\quad \delta = \delta \cup \{(q_l', \sqcap\{..., \langle link_i, OK\rangle, ...\}, \perp)\}$ $else$ $\quad \delta = (\delta \setminus \{(q_l, a, q_l')\}) \cup \{(q_l, a, q_{t_1})\}$ $\quad \delta = \delta \cup \{(q_{t_1}, \sqcap\{..., \langle link_i, OK\rangle, ...\}, q_l')\}$ end $\mathcal{R} = \mathcal{R} \cup \{(\langle link_i, OK\rangle, q_{l_i})\}$ $\delta = \delta \cup \{(q_{l_i}, \ell, q_t)\}$ $L = L \cup \{\langle Activity_1^i, (q_{l_i}, \ell, q_t)\rangle\}\ \dots$
link $(target)$	$a = \langle name, GetResult(Activity_2, r')\rangle$ $M = \{t' \mid t' \in \delta \wedge \exists q \in L\ (q.t = t' \wedge q.target = Activity)\}$ $\mathcal{L} = (\mathcal{L} \setminus \{\langle \mathcal{R}(a), \ovee\rangle\}) \cup \{\langle \mathcal{R}(a), \owedge\rangle\}$ $\delta = (\delta \setminus M) \cup \{(t.q, \ell, \langle \mathcal{R}(a), \owedge\rangle) \mid t \in M\}$

Table 2: Translating the structured BPEL activities.

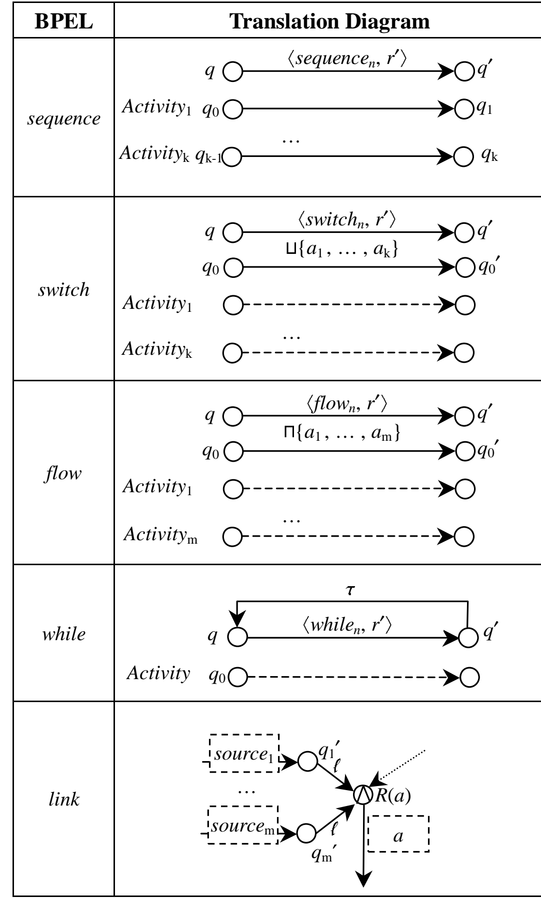


Figure 4: Translation diagrams of the structured BPEL activities.

dled by the *scope faultHandles* and the *scope* will not diffuse the exception to the outside *scope*. So the translation should incorporate some executions that will result in exceptions, which will be surely handled by the *faultHandles* of the *scope*; (3) the *scope* result is not *OK*, and this condition reflects that the *scope* must result in exception. According to the BPEL specification, if there is no explicit *faultHandlers* for the *scope*, the default *faultHandlers* 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 $\delta = \delta \cup \{(q_f, \widehat{(\!|n|\!)}, \boxtimes)\}$ 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 *faultHandles* is $translate_f(Fault, q, s, r_1, r_2)$. The meanings of q and s are same as those of $translate_f(Activity, q, s, r)$, r_1 is the result of the activity in the *scope*, and r_2 is the actual result of the *scope*. $r_1 \in N$ represents that the exception can be handled by a *catch* handle. $r_1 \notin N \wedge r_2 = OK$ represents that the exception will be handled by the *catchAll* handle. $r_1 \notin N \wedge r_2 \neq OK$ 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 $translate_p(Process)$ is similar to that of *scope*. The generated action of the *process* should be added not only to the domain of $\mathcal{R}$, but also to the provided action set $\mathcal{D}$.

BPEL	Translation Procedure
scope	$r' = GetResult(scope, r)$ $R = GetResult(Activity)$ $R_r = GetResult(scope)$ $E = GetException(scope)$
	$r' = OK \wedge \#E = 0$

	$\delta = \delta \cup \{(q, (\!|n, s|\!), q_t), (q_t, \langle scope_n, r' \rangle, q')\}$
	$\mathcal{R} = \mathcal{R} \cup \{(\langle scope_n, r' \rangle, q_0)\}$
	$translate(Activity, q_0, n, r')$
	$\mathcal{R}_\mathcal{C} = \mathcal{R}_\mathcal{C} \cup \{(\langle scope_n, r' \rangle, q_c)\}$
	if Comp exists then
	  $translate(Comp, q_c, n, r')$
	else
	  $\delta = \delta \cup \{(q_c, \widehat{(\!|n|\!)}, \perp)\}$

	$r' = OK \wedge \#E > 0, E = E \setminus (R_r \setminus \{OK\})$

	for each $r_i \in E$
	  $\mathcal{R} = \mathcal{R} \cup \{(\langle scope_n, r_i \rangle, q_i)\}$
	  $translate(Activity, q_i, n, r_i)$
	  $\mathcal{R}_\mathcal{F} = \mathcal{R}_\mathcal{F} \cup \{(\langle scope_n, r_i \rangle, q_{f_i})\}$
	  $translate_f(Fault, q_{f_i}, n, r_i, OK)$
	end
	$\mathcal{B} = \bigcup\{\langle scope_n, r_i \rangle\}$, *where* $r_i \in E$
	if $OK \in R$ *then*
	  $\mathcal{R} = \mathcal{R} \cup \{(\langle scope_n, r' \rangle, q_0)\}$
	  $translate(Activity, q_0, n, r')$
	  $\mathcal{R}_\mathcal{C} = \mathcal{R}_\mathcal{C} \cup \{(\langle scope_n, r' \rangle, q_c)\}$
	  if Comp exists then
	    $translate(Comp, q_c, n, r')$
	  else
	    $\delta = \delta \cup \{(q_c, \widehat{(\!|n|\!)}, \perp)\}$
	  end
	  $\mathcal{B} = \mathcal{B} \cup \{\langle scope_n, r' \rangle\}$
	end
	$\delta = \delta \cup \{(q, (\!|n, s|\!), q_t), (q_t, \sqcup\mathcal{B}, q')\}$

	$r' \neq OK$

	$\delta = \delta \cup \{(q, (\!|n, s|\!), q_t), (q_t, \langle scope_n, r' \rangle, q')\}$
	$\mathcal{R} = \mathcal{R} \cup \{(\langle scope_n, r' \rangle, q_0)\}$
	$translate(Activity, q_0, n, r')$
	$\mathcal{R}_\mathcal{F} = \mathcal{R}_\mathcal{F} \cup \{(\langle scope_n, r' \rangle, q_f)\}$
	if Fault exists then
	  $translate_f(Fault, q_f, n, r', r')$
	else
	  $\delta = \delta \cup \{(q_f, \widehat{(\!|n|\!)}, \boxtimes)\}$

Fault	$N = \{n_1, ..., n_m\}$
	if $r_1 \in N$ *then*
	  $translate(Activity_{r_1}, q, s, OK)$
	if $r_1 \notin N \wedge r_2 = OK$ *then*
	  $translate(Activity_a, q, s, OK)$
	if $r_1 \notin N \wedge r_2 \neq OK$ *then*
	  $\delta = \delta \cup \{(q, \tau, \boxtimes)\}$

Comp	$translate(Activity, q, s, r)$

Table 3: Translating the scope activity.

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 skeletons will be presented in algorithms 1, 2 and 3.

- L and $\mathcal{T}$ used in the preceding translation procedures are global variables. L is a set and $\mathcal{T}$ is a protocol interface.

BPEL	Translation Procedure
Process	$R_p = GetProcessResult(process)$ $R_{pt} = GetResult(Activity)$ $E = GetException(Activity)$ *For each* $r_i \in R_p$
	$r_i = OK \wedge \#E = 0$

	$\mathcal{R} = \mathcal{R} \cup \{(\langle n, r_i \rangle, q_0)\}, \mathcal{D} = \mathcal{D} \cup \{\langle n, r_i \rangle\}$
	$translate(Activity, q_0, n, r_i)$
	$\mathcal{R}_\mathcal{C} = \mathcal{R}_\mathcal{C} \cup \{(\langle n, r_i \rangle, q_c)\}$
	if Comp exists then
	  $translate(Comp, q_c, n, r_i)$
	else
	  $\delta = \delta \cup \{(q_c, \widehat{(\!|n|\!)}, \perp)\}$

	$r_i = OK \wedge \#E > 0, E = E \setminus (R_p \setminus \{OK\})$

	for each $r_j \in E$
	  $\mathcal{R} = \mathcal{R} \cup \{(\langle n, r_j \rangle, q_j)\}, \mathcal{D} = \mathcal{D} \cup \{\langle n, r_j \rangle\}$
	  $translate(Activity, q_j, n, r_j)$
	  $\mathcal{R}_\mathcal{F} = \mathcal{R}_\mathcal{F} \cup \{(\langle n, r_j \rangle, q_{f_j})\}$
	  $translate_f(Fault, q_{f_j}, n, r_j, OK)$
	end
	if $OK \in R_{pt}$ *then*
	  $\mathcal{R} = \mathcal{R} \cup \{(\langle n, OK \rangle, q_0)\}$
	  $\mathcal{D} = \mathcal{D} \cup \{\langle n, OK \rangle\}$
	  $translate(Activity, q_0, n, OK)$
	  $\mathcal{R}_\mathcal{C} = \mathcal{R}_\mathcal{C} \cup \{(\langle n, OK \rangle, q_c)\}$
	  if Comp exists then
	    $translate(Comp, q_c, n, OK)$
	  else
	    $\delta = \delta \cup \{(q_c, \widehat{(\!|n|\!)}, \perp)\}$
	  end
	end

	$r_i \neq OK$

	$\mathcal{R} = \mathcal{R} \cup \{(\langle n, r_i \rangle, q_0)\}, \mathcal{D} = \mathcal{D} \cup \{\langle n, r_i \rangle\}$
	$translate(Activity, q_0, n, r_i)$
	$\mathcal{R}_\mathcal{F} = \mathcal{R}_\mathcal{F} \cup \{(\langle n, r_i \rangle, q_f)\}$
	if Fault exists then
	  $translate_f(Fault, q_f, n, r_i, r_i)$
	else
	  $\delta = \delta \cup \{(q_f, \widehat{(\!|n|\!)}, \boxtimes)\}$

Table 4: Translating the process.

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_r, E, Result r', r_i, **boolean** b_s,
location $q', q_t, q_0, q_c, q_f, q_i, q_{f_i}, q_l, q_{l_i}, q_l', q_{t_1}$;

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: $b_s = false$;
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: $b_s = true$;
15: **end**
16: **if** $Activity$ is the target of a *link* **then**
17: generate *middle location* q_n for the q;
18: do the translation procedures in Table 2;
19: **end**
20: **if** the $Activity$ parent is not *sequence* **and** $b_s = false$ **and** $q' \neq \boxtimes$ **then**
21: $\delta = \delta \cup \{(q', \tau, \bot)\}$;
22: **return** q';

Algorithm 2 translate$_\mathbf{f}$($Fault, q, s, r_1, r_2$)

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: $translate(Activity_{r_1}, q, s, OK)$;
4: **if** $r_1 \notin N$ **and** $r_2 = OK$ **then**
5: $translate(Activity_a, q, s, OK)$;
6: **if** $r_1 \notin N$ **and** $r_2 \neq OK$ **then**
7: $\delta = \delta \cup \{(q, \tau, \boxtimes)\}$;

Algorithm 3 translate$_\mathbf{p}$($Process$)

Input : The input $Process$;
Output : none;
Variables : Set E, R_p, R_{pt}, Output r_i, r_j,
location $q_0, q_c, q_j, q_f, q_{f_j}$;

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_i, F;**boolean** r_f;

1: $R = \emptyset$;
2: **if** $Activity$ is the basic activity **and** $Activity \neq throw$ **then**
3: $R = \{OK\}$;
4: **if** $Activity = throw(n)$ **then**
5: $R = \{n\}$;
6: **if** $Activity$ is the structured activity **and** $Activity \neq while$ **then**
7: $r_f = false$;
8: **for each** $i = 1$ **to** m **do**
9: $R_i = GetResult(Activity_i)$;
10: $R = R \cup R_i$;
11: **if** $OK \notin R_i \wedge Activity = sequence$ **then**
12: $R = R \setminus \{OK\}$;
13: **break**;
14: **end**
15: **if** $OK \notin R_i \wedge Activity = flow$ **then**
16: $r_f = true$;
17: **end**
18: **if** $r_f = true \wedge Activity = flow$ **then**
19: $R = R \setminus \{OK\}$;
20: **end**
21: **if** $Activity = while(n, Activity_1)$ **then**
22: $R = GetResult(Activity_1)$;
23: **if** $Activity = scope(n, Activity_1, ...)$ **then**
24: **if** $Fault$ has *catchAll* **then**
25: $R = \{OK\}$;
26: **else**
27: $R_i = GetResult(Activity_1)$;
28: $F = \{n_1, ..., n_m\}$;
29: $R = R_i \setminus F$;
30: **if** $\#R < \#R_i$ **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 $O(k * n * n)$, 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.

Algorithm 5 GetException(*Activity*)

Input : The input *Activity*;
Output : The set of exceptions that can be resulted from the *Activity*;
Variables : Set E, R_i;

```
1: E = ∅;
2: if Activity is the basic activity and
3:    Activity ≠ throw then
4:    E = ∅;
5: if Activity = throw(n) then
6:    E = {n};
7: if Activity is the structured activity and
      Activity ≠ while then
8:    for each i = 1 to m do
9:       R_i = GetResult(Activity_i);
10:      E = E ∪ (R_i \ {OK});
11:      if OK ∉ R_i ∧ Activity = sequence then
12:         break;
13:   end
14: end
15: if Activity = while(n, Activity_1) then
16:    E = GetException(Activity_1);
17: if Activity = scope(n, Activity_1, ...) then
18:    E = GetException(Activity_1);
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*.

Algorithm 6 GetProcessResult(*Process*)

Input : The input *Process*;
Output : The set of possible results of *Process*;;
Variables : Set R, R_i;

```
1: if Fault has catchAll then
2:    R = {OK};
3: else
4:    R_i = GetResult(Activity);
5:    F = {n_1, ..., n_m};
6:    R = R_i \ F;
7:    if #R < #R_i then
8:       R = R ∪ {OK}
9: end
10: return R;
```

- 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 $k = max\{1, n * j\}$, n is the number of XML elements in the corresponding XML of the input *Activity*, and if r is OK, then $j = 0$, else $j = 1$.

Algorithm 7 GetResult(*Activity*, r)

Input : The input *Activity* and the outside result r;
Output : The actual result of *Activity*;
Variables : Set R, r_i;

```
1: if r = OK then
2:    r_i = OK
3: else
4:    R = GetResult(Activity);
5:    if r ∈ R then
6:       r_i = r;
7:    else
8:       r_i = OK;
9:    end
10: return r_i;
```

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 $a \to \varphi$, where φ 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 φ over an action set $\mathcal{A}$ must meet the following syntactic rules, where $D \subseteq \mathcal{A}$.*

$$\chi ::= true \mid false \mid D \mid \neg\chi \mid \chi \wedge \chi'$$
$$\varphi ::= true \mid false \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \boldsymbol{E}\gamma \mid \boldsymbol{A}\gamma$$
$$\gamma ::= [\varphi\{\chi\}\ \boldsymbol{U}\ \{\chi'\}\varphi'] \mid [\varphi\{\chi\}\ \underline{\boldsymbol{U}}\ \{\chi'\}\varphi']$$

Compared to *Action Computation Tree Logic* (ACTL) in Meolic 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 $M = \langle S, I, L, \delta \rangle$ and a path $p = s_1 A_1 s_2 A_2 s_3 ...$ of M, where $s_i \in S, A_i \in L, i$ is the natural number and $i \geq 1$, the semantics of path formula can be given as follows.

$$p \models [\varphi\{\chi\} \ \mathbf{U} \ \{\chi'\}\varphi'] \ \textit{iff} \ \exists i \geq 1, \forall j \in [1, i) \bullet (A_i \models \chi' \wedge s_{i+1} \models \varphi' \wedge s_i \models \varphi \wedge s_j \models \varphi \wedge A_j \models \chi)$$

$$p \models [\varphi\{\chi\} \ \underline{\mathbf{U}} \ \{\chi'\}\varphi'] \ \textit{iff} \ p \models [\varphi\{\chi\} \ \mathbf{U} \ \{\chi'\}\varphi'] \vee (\forall i \geq 1 \bullet (A_i \models \chi \wedge s_i \models \varphi))$$

- 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).

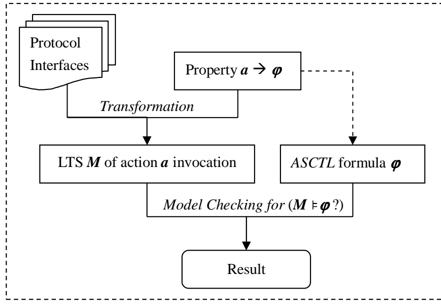


Figure 5: The verification process of protocol property.

The complete verification process of protocol property is shown in Figure 5. First, the corresponding LTS of the property should be generated, and the LTS only contains the transitions whose labels are external action sets. Next, Model checking technique is used for protocol property verification. The method for model checking is same as Meolic et al. (2000), which use symbolic model checking method based on fixed point calculation for ACTL verification. It is necessary to point out that the computation complexity of ASCTL verification is same as that of ACTL verification in Meolic et al. (2000) with the assumption that the time for judging $A \models D$ is the unit time. Currently, we use the first version of Efficient Symbolic Tools (EST) (EST, 2006) as our underlying experimental tool for verifying properties and LTS models whose containing action sets are single action sets.

4.2 Methodology

The verification methodology is shown in Figure 6. The Web service development process can be divided into many steps, in some of which verification can be taken to ensure the correctness of the system under development.

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

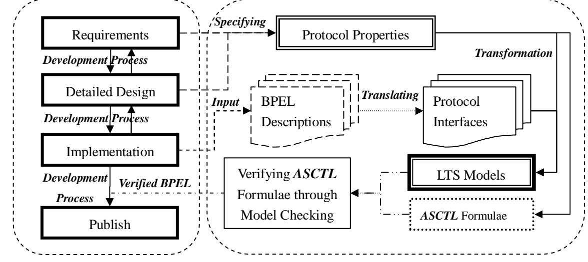


Figure 6: The verification methodology.

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 the preceding all steps, the verified BPEL description can be deployed to some BPEL engines such as BPWS4J. The actual Web services can be established and provide services to the clients.

5 Case Study

This section demonstrates the formalization and verification by two examples. The first example is used to exemplify the multi-scope, 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.

5.1 Example 1

The BPEL script skeleton is shown in Figure 7. The *process* $p1$ has a *scope* activity $s1$ in which *sequence* activity se contains five sequential activities. There is no explicit *faultHandlers* for $s1$, 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.

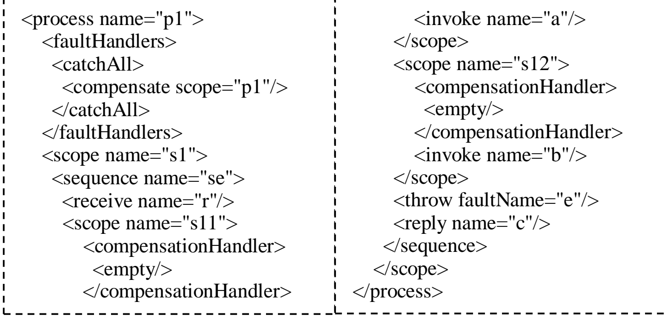
{

$\langle \mathbf{p1}, \mathbf{e} \rangle \longrightarrow_{\mathcal{R}} (q_0, (\!| s1, p1 |\!), q_1), (q_1, \langle \text{s1,e} \rangle, \boxtimes),$

$\langle \mathbf{s1}, \mathbf{e} \rangle \longrightarrow_{\mathcal{R}} (q_3, \langle \text{se,e} \rangle, \boxtimes),$

$\langle \mathbf{se}, \mathbf{e} \rangle \longrightarrow_{\mathcal{R}} (q_5, \langle recevie_r, \text{OK} \rangle, q_6)$
$(q_6, (\!| s11, s1 |\!), q_7), (q_7, \langle scope_{s11}, \text{OK} \rangle, q_8),$
$(q_8, (\!| s12, s1 |\!), q_{13}), (q_{13}, \langle scope_{s12}, \text{OK} \rangle, q_{14}),$
$(q_{14}, \tau, \boxtimes),$

$\langle \mathbf{scope_{s11}}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}} (q_9, \langle invoke_a, \text{OK} \rangle, q_{10}), (q_{10}, \tau, \perp),$



```
<process name="p1">
  <faultHandlers>
    <catchAll>
      <compensate scope="p1"/>
    </catchAll>
  </faultHandlers>
  <scope name="s1">
    <sequence name="se">
      <receive name="r"/>
      <scope name="s11">
        <compensationHandler>
          <empty/>
        </compensationHandler>
        <invoke name="a"/>
      </scope>
      <scope name="s12">
        <compensationHandler>
          <empty/>
        </compensationHandler>
        <invoke name="b"/>
      </scope>
      <throw faultName="e"/>
      <reply name="c"/>
    </sequence>
  </scope>
</process>
```

Figure 7: BPEL script skeleton of Example 1.

$\langle \mathbf{scope_{s12}}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}} (q_{15}, \langle invoke_b, \text{OK} \rangle, q_{16}), (q_{16}, \tau, \perp),$
$\langle \mathbf{scope_{s11}}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{11}, \tau, q_{12}), (q_{12}, \tau, \perp),$
$\langle \mathbf{scope_{s12}}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{17}, \tau, q_{18}), (q_{18}, \tau, \perp),$
$\langle \mathbf{s1}, \mathbf{e} \rangle \longrightarrow_{\mathcal{R}_{\mathcal{F}}} (q_{19}, \widehat{(\!|s1|\!)}, \boxtimes),$
$\langle \mathbf{p1}, \mathbf{e} \rangle \longrightarrow_{\mathcal{R}_{\mathcal{F}}} (q_{20}, \widehat{(\!|p1|\!)}, \perp)$
}

The provided action set is $\{\langle p1, e \rangle\}$. 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* handle 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 *compensationHandlers* for $s12$ and $s11$ in sequence.

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.

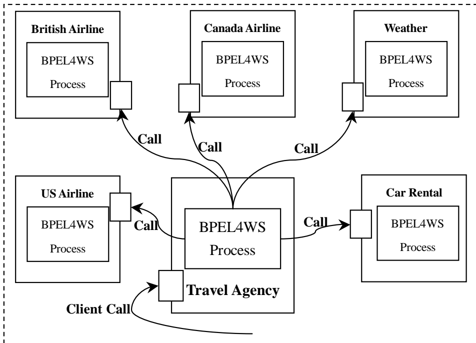


Figure 9: Orchestration architecture of the travel agency Web service.

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*.

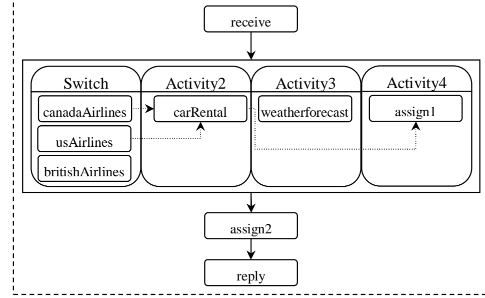


Figure 10: The structure of the travel agency BPEL process.

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 $\mathcal{D}$ is $\{\langle BookTravel, OK \rangle\}$.
{

$\langle \mathbf{BookTravel}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}} (q_0, \langle \text{travelSeq,OK} \rangle, q_1), (q_1, \tau, \perp),$
$\langle \mathbf{TravelSeq}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}} (q_2, \langle \text{bookReceive,OK} \rangle, q_3),$
$(q_3, \langle flow_1, \text{OK} \rangle, q_4),$
$(q_{22}, \langle assign_2, \text{OK} \rangle, q_{23}),$
$(q_{23}, \langle \text{BookReply,OK} \rangle, \perp),$
$\langle \mathbf{flow_1}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}} (q_5, \sqcap\{\langle branch_1, OK \rangle, \langle branch_2, OK \rangle,$
$\langle branch_3, OK \rangle, \langle branch_4, OK \rangle\}, \perp),$
$\langle \mathbf{branch_1}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}} (q_6, \langle switch_1, OK \rangle, q_7), (q_7, \tau, \perp),$
$\langle \mathbf{switch_1}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}} (q_8, \sqcup\{\langle branch_5, OK \rangle, \langle branch_6, OK \rangle,$
$\langle branch_7, OK \rangle\}, \perp),$
$\langle \mathbf{branch_5}, \mathbf{OK} \rangle \longrightarrow_{\mathcal{R}} (q_9, \langle invokeca, OK \rangle, q_{10}),$

```
<process name="BookTravel"> …
<sequence name="travelSeq">
  <receive operation="bookTravel" name="bookReceive"/>
  <flow>
    <links>
      <link name="travel-canada"/>
      <link name="travel-us"/>
      <link name="rent-to-assign"/>
    </links>
    <switch>
      <case condition = "…">
        <invoke name="invokeca" operation="makeReservation" …>
          <source linkName="travel-canada" />
        </invoke>
      </case>
      <case condition = "…">
        <invoke name="invokeam" operation="makeReservation" …>
          <source linkName="travel-us" transitionCondition="…"/>
        </invoke>
      </case>
      <otherwise>
        <invoke name="invokebr" operation="makeReservation" />
      </otherwise>
    </switch>
    <invoke name="invokeweather" operation="queryWeather"/>
    <invoke name="invokerental" operation="makeCarRental">
      <target linkName="travel-canada"/>
      <target linkName="travel-us"/>
      <source linkName="rent-to-assign"/>
    </invoke>
    <assign>
      <target linkName="rent-to-assign" />
      <copy>...</copy>
    </assign>
  </flow>
<assign> … </assign>
<reply operation="bookTravel" name="BookReply"/>
</sequence>
</process>
```

Figure 8: BPEL specification of the travel agency service.

```
<process name="BookTravel"> …
<faultHandlers>
  <catch name="NOCAR">
      <invoke name="apologize" …/>
  </catch>
</faultHandlers>
<sequence name="travelSeq">
  <receive operation="bookTravel" name="bookReceive"/>
  <scope name="s1">
  <flow>
    <links>
      <link name="travel-canada"/>
      <link name="travel-us"/>
      <link name="rent-to-assign"/>
    </links>
    <switch>
      <case condition = "…">
        <invoke name="invokeca" operation="makeReservation" …>
          <source linkName="travel-canada" />
          <compensationHandler>
            <invoke name="returnca" />
          </compensationHandler>
        </invoke>
      </case>
      <case condition = "…">
        <invoke name="invokeam" operation="makeReservation" …>
          <source linkName="travel-us" transitionCondition="…"/>
          <compensationHandler>
            <invoke name="returnam" />
          </compensationHandler>
        </invoke>
      </case>
      <otherwise>
        <invoke name="invokebr" operation="makeReservation">
          <compensationHandler>
            <invoke name="returnam" />
          </compensationHandler>
        </invoke>
      </otherwise>
    </switch>
    <invoke name="invokeweather" operation="queryWeather"/>
    <scope name="s2">
      <sequence>
      <invoke name="invokerental" operation="makeCarRental">
        <target linkName="travel-canada"/>
        <target linkName="travel-us"/>
        <source linkName="rent-to-assign"/>
      </invoke>
      <switch>
        <case …><throw faultName="NOCAR"/></case>
        <otherwise><empty /></otherwise>
      </switch>
      </sequence>
    </scope>
    <assign>
      <target linkName="rent-to-assign" />
      <copy>...</copy>
    </assign>
  </flow>
</scope>
<assign> … </assign>
<reply operation="bookTravel" name="BookReply"/>
</sequence> …
</process>
```

Figure 11: BPEL specification of the travel agency service with transaction.

$(q_{10}, \sqcap\{\langle link_1, OK\rangle\}, \bot)$,
$\langle \mathbf{link_1}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{25}, \ell, \langle q_{17}, \circledtriangle\rangle)$,
$\langle \mathbf{branch_6}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{11}, \langle invokeam, OK\rangle, q_{12})$,
$(q_{12}, \sqcap\{\langle link_2, OK\rangle\}, \bot)$,
$\langle \mathbf{link_2}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{26}, \ell, \langle q_{17}, \circledtriangle\rangle)$,
$\langle \mathbf{branch_7}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{13}, \langle invokebr, OK\rangle, q_{14}), (q_{14}, \tau, \bot)$,
$\langle \mathbf{branch_2}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{15}, \langle invokeweather, OK\rangle, q_{16}), (q_{16}, \tau, \bot)$,
$\langle \mathbf{branch_3}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (\langle q_{17}, \circledtriangle\rangle, \langle invokerental, OK\rangle, q_{18})$,
$(q_{18}, \sqcap\{\langle link_3, OK\rangle\}, \bot)$,
$\langle \mathbf{link_3}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{27}, \ell, \langle q_{19}, \circledtriangle\rangle)$,
$\langle \mathbf{branch_4}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (\langle q_{19}, \circledtriangle\rangle, \langle assign_1, OK\rangle, q_{20}), (q_{20}, \tau, \bot)$,
$\langle \mathbf{assign_1}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} \bot$,
$\langle \mathbf{assign_2}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} \bot$,
$\langle \mathbf{BookTravel}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{24}, (\!|\widehat{BookTravel}|\!), \bot)$
}

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 $\{\langle BookTravel, OK\rangle, \langle BookTravel, NOCAR\rangle\}$. It is necessary to point out that the translated orchestration process needs a new Web service for sending letters.

{
$\langle \mathbf{BookTravel}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_0, \langle \text{travelSeq,OK}\rangle, q_1), (q_1, \tau, \bot)$,
$\langle \mathbf{TravelSeq}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_2, \langle \text{bookReceive,OK}\rangle, q_3)$,
$(q_3, (\!|s1, BookTravel|\!), q_4)$,
$(q_4, \langle scope_{s1}, \text{OK}\rangle, q_5)$,
$(q_5, \langle assign_2, \text{OK}\rangle, q_{53})$,
$(q_{53}, \langle \text{BookReply,OK}\rangle, \bot)$,
$\langle \mathbf{scope_{s1}}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_6, \langle flow_1, OK\rangle, q_7), (q_7, \tau, \bot)$,
$\langle \mathbf{flow_1}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_8, \sqcap\{\langle branch_1, OK\rangle, \langle branch_2, OK\rangle,$
$\langle branch_3, OK\rangle, \langle branch_4, OK\rangle\}, \bot)$,
$\langle \mathbf{branch_1}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_9, \langle switch_1, OK\rangle, q_{10}), (q_{10}, \tau, \bot)$,
$\langle \mathbf{switch_1}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{11}, \sqcup\{\langle branch_5, OK\rangle, \langle branch_6, OK\rangle,$
$\langle branch_7, OK\rangle\}, \bot)$,
$\langle \mathbf{branch_5}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{12}, (\!|s11, s1|\!), q_{13}), (q_{13}, \langle s_{11}, OK\rangle, q_{14})$,
$(q_{14}, \tau, \bot)$,
$\langle \mathbf{s11}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{15}, \langle invokeca, OK\rangle, q_{16})$,
$(q_{16}, \sqcap\{\langle link_1, OK\rangle\}, \bot)$,
$\langle \mathbf{link_1}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{45}, \ell, \langle q_{43}, \circledtriangle\rangle)$,
$\langle \mathbf{branch_6}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{19}, (\!|s12, s1|\!), q_{20}), (q_{20}, \langle s_{12}, OK\rangle, q_{21})$,
$(q_{21}, \tau, \bot)$,
$\langle \mathbf{s12}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{22}, \langle invokeam, OK\rangle, q_{23})$,
$(q_{23}, \sqcap\{\langle link_2, OK\rangle\}, \bot)$,
$\langle \mathbf{link_2}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{52}, \ell, \langle q_{43}, \circledtriangle\rangle)$,
$\langle \mathbf{branch_7}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{27}, (\!|s13, s1|\!), q_{28}), (q_{28}, \langle s_{13}, OK\rangle, q_{29})$,
$(q_{29}, \tau, \bot)$,
$\langle \mathbf{s13}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{30}, \langle invokebr, OK\rangle, q_{31}), (q_{31}, \tau, q_{19})$,
$\langle \mathbf{branch_2}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{34}, \langle invokeweather, OK\rangle, q_{35}), (q_{35}, \tau, \bot)$,
$\langle \mathbf{branch_3}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{36}, (\!|s2, s1|\!), q_{37}), (q_{37}, \langle scope_{s2}, OK\rangle, q_{38})$,
$(q_{38}, \tau, \bot)$,
$\langle \mathbf{scope_{s2}}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{39}, \langle sequence_1, OK\rangle, q_{40}), (q_{40}, \tau, \bot)$,
$\langle \mathbf{sequence_1}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{41}, \langle invoke_{link}, OK\rangle, q_{42})$,
$(q_{42}, \langle switch_2, OK\rangle, \bot)$,
$\langle \mathbf{invoke_{link}}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (\langle q_{43}, \circledtriangle\rangle, \langle invokerental, OK\rangle, q_{44})$,
$(q_{44}, \sqcap\{\langle link_3, OK\rangle\}, \bot)$,
$\langle \mathbf{link_3}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{60}, \ell, \langle q_{50}, \circledtriangle\rangle)$,
$\langle \mathbf{switch_2}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{46}, \sqcup\{\langle branch_8, OK\rangle\}, \bot)$,
$\langle \mathbf{branch_8}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (q_{47}, \tau, q_{48}), (q_{48}, \tau, \bot)$,
$\langle \mathbf{branch_4}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} (\langle q_{50}, \circledtriangle\rangle, \langle assign_1, OK\rangle, q_{51}), (q_{51}, \tau, \bot)$,
$\langle \mathbf{assign_1}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} \bot$,
$\langle \mathbf{assign_2}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}} \bot$,
$\langle \mathbf{BookTravel}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{54}, \langle \text{travelSeq,NOCAR}\rangle, q_{55})$,
$(q_{55}, \tau, \bot)$,
$\langle \mathbf{TravelSeq}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{56}, \langle \text{bookReceive,OK}\rangle, q_{57})$,
$(q_{57}, (\!|s1, BookTravel|\!), q_{58})$,
$(q_{58}, \langle scope_{s1}, \text{NOCAR}\rangle, \boxtimes)$,
$\langle \mathbf{scope_{s1}}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{59}, \langle flow_1, NOCAR\rangle, \boxtimes)$,
$\langle \mathbf{flow_1}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{61}, \sqcap\{\langle branch_1, OK\rangle, \langle branch_2, OK\rangle,$
$\langle branch_3, NOCAR\rangle, \langle branch_4, OK\rangle\}, \boxtimes)$,
$\langle \mathbf{branch_3}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{62}, (\!|s2, s1|\!), q_{63})$,
$(q_{63}, \langle scope_{s2}, NOCAR\rangle, \boxtimes)$,
$\langle \mathbf{scope_{s2}}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{65}, \langle sequence_1, NOCAR\rangle, \boxtimes)$,
$\langle \mathbf{sequence_1}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{67}, \langle invoke_{link}, OK\rangle, q_{68})$,
$(q_{68}, \langle switch_2, NOCAR\rangle, \boxtimes)$,
$\langle \mathbf{switch_2}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{69}, \sqcup\{\langle branch_8, NOCAR\rangle\}, \boxtimes)$,
$\langle \mathbf{branch_8}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}} (q_{70}, \tau, \boxtimes)$,
$\langle \mathbf{s11}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{17}, \langle returnca, OK\rangle, q_{18}), (q_{18}, \tau, \bot)$,
$\langle \mathbf{s12}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{24}, \langle returnam, OK\rangle, q_{25}), (q_{26}, \tau, \bot)$,
$\langle \mathbf{s13}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{32}, \langle returnbr, OK\rangle, q_{33}), (q_{33}, \tau, \bot)$,
$\langle \mathbf{scope_{s2}}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{49}, (\!|\widehat{s2}|\!), \bot)$,
$\langle \mathbf{scope_{s1}}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{71}, (\!|\widehat{s1}|\!), \bot)$,
$\langle \mathbf{BookTravel}, \mathbf{OK}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{C}}} (q_{72}, (\!|\widehat{BookTravel}|\!), \bot)$,
$\langle \mathbf{scope_{s1}}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{F}}} (q_{73}, (\!|\widehat{s1}|\!), \boxtimes)$,
$\langle \mathbf{scope_{s2}}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{F}}} (q_{74}, (\!|\widehat{s2}|\!), \boxtimes)$,
$\langle \mathbf{BookTravel}, \mathbf{NOCAR}\rangle \longrightarrow_{\mathcal{R}_{\mathcal{F}}} (q_{75}, \langle apologize, OK\rangle, \bot)$
}

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.

Protocol Property	Result
$\langle$BookTravel,OK$\rangle \rightarrow$ **AF**$\{\{\langle$switch,OK$\rangle\}\}$	True
$\langle$BookTravel,OK$\rangle \rightarrow$ **AF**$\{\{\langle$invokeam,OK$\rangle\}\}$	False
$\langle$BookTravel,NOCAR$\rangle \rightarrow$ **AF**$\{\{\langle$apologize,OK$\rangle\}\}$	True
$\langle$BookTravel,NOCAR$\rangle \rightarrow$ $\neg$ **E** $[\{\neg\{\langle$invokeam,OK$\rangle\}\}$ **U** $\{\{\langle$returnam,OK$\rangle\}\}]$	True
$\langle$BookTravel,NOCAR$\rangle \rightarrow$ $\neg$ **E** $[\{\neg\{\langle$returnam,OK$\rangle\}\}$ **U** $\{\{\langle$apologize,OK$\rangle\}\}]$	False
$\langle$BookTravel,NOCAR$\rangle \rightarrow \neg$ **E** $[\{\neg$ $\{\langle$returnam,OK$\rangle\}\}$ **U** $\{\{\langle$apologize,OK$\rangle\}\}$ **EF** $\{\{\langle$returnam,OK$\rangle\}\}]$	True

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

6 Related Work

There are several work on formalizing business transactions (Bocchi et al., 2003; Butler et al., 2005; Butler and Ripon, 2005; Bruni et al., 2005). Bocchi et al. (2003) propose πt-calculus that extends asynchronous π-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 Concurrency Work-Bench (CWB) by a syntax compiler plug-in for BPEL-Calculus;

- **Automata**: Fu et al. (2004) use guarded finite state automata (GFSA) to describe service composition and formalize BPEL, and the GFSA 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 taken 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 work 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 ($StAC_i$) to formalize the named compensation behaviour. Qiu et al. (2005) propose a formal operational semantics for a subset of constructs in BPEL by a 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-

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

Arkin, A. *et al.* (2002) 'Web Service Choreography Interface Version 1.0', *http://www.w3.org/TR/wsci/.*

Beyer, D., Chakrabarti, A. and Henzinger, T.A. (2005a) 'Web Service Interfaces', *14th International World Wide Web Conference (WWW'05)*, pp.148–159, Chiba, Japan.

Beyer, D., Chakrabarti, A. and Henzinger, T.A. (2005b) 'An Interface Formalism for Web Services', *1st International Workshop on Foundations of Interface Technologies*, San Francisco, CA.

Bocchi, L., Laneve, C. and Zavattaro, G. (2003) 'A Calculus for Long Running Transactions', *6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'03)*, LNCS 2884, pp.124–138, Paris, France.

Bruni, R., Melgratti, H. and Montanari, U. (2005) 'Theoretical Foundations for Compensations in Flow Composition Languages', *32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pp.209–220, California, USA.

Butler, M., Ferreira, C. and Ng, M.Y. (2005) 'Precise Modelling of Compensating Business Transactions and its Application to BPEL', *Journal of Universal Computer Science*, Vol. 11, No. 5, pp.712–743.

Butler, M. and Ripon S. (2005) 'Executable Semantics for Compensating CSP', *2nd International Workshop on Web Services and Formal Methods (WS-FM'05)*, LNCS 3670, pp.243–256, Versailles, France.

Chen, Z.B., Wang, J., Dong, W. and Qi, Z.C. (2006a) 'An interface model for service-oriented software architecture', *Journal of Software*, Vol. 17, No. 7, pp. 1459-1469.

Chen, Z.B., Wang, J., Dong, W. and Qi, Z.C. (2006b) 'Towards Formal Interfaces for Web Services with Transactions', *2nd International Conference On Signal-Image Technology & Internet-based Systems (SITIS'06)*, pp.218-229, Hammamet, Tunisia.

Curbera, F. *et al.* (2003) 'Business Process Execution Language For Web Services Version 1.1', *ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.*

de Alfaro, L. and Henzinger, T.A. (2001) 'Interface automata', *8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'01)*, pp.109–120, Vienna, Austria.

EST (2006) 'Homepage of EST', *http://lms.uni-mb.si/EST/.*

Farahbod, R. (2004) 'Extending and Refining an Abstract Operational Semantics of the Web Services Architecture for the Business Process Execution Language', [*MS. Thesis*], Simon Fraser University, Burnaby B.C., Canada.

Fahland, D. (2005) 'Complete Abstract Operational Semantics for the Web Service Business Process Execution Language', *Technical Report*, Humboldt University at Berlin.

Foster, H., Uchitel, S., Magee, J. and Kramer, J. (2004) 'Compatibility for Web Service Choreography', *3rd IEEE International Conference on Web Services (ICWS'04)*, pp.738–741, San Diego, CA.

Fu, X., Bultan, T. and Su, J. (2004) 'Analysis of Interacting BPEL Web Services', *13th International World Wide Web Conference (WWW'04)*, pp.621–630, New York, NY, USA.

IBM (2004) 'Homepage of BPWS4J', *http://www.alphaworks.ibm.com/tech/bpws4j.*

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M. and Irwin, J. (1997) 'Aspect-Oriented Programming', *11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, pp.220–242, Finland.

Koshkina, M. (2003) 'Verification of Business Processes for Web Services', [*MS. Thesis*], York University.

Leymann, F. (2001) 'Web Services Flow Language Version 1.0', *http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.*

Little, M. (2003) 'Transactions and Web Services', *Communication of the ACM*, Vol. 46, No. 10, pp.49–54.

Meolic, R., Kapus, T. and Brezocnik, Z. (2000) 'Verification of Concurrent Systems using ACTL', *The IASTED International Conference of Artificial Intelligence (AI'2000)*, pp.663–669, Innsbruck, Austria.

Ouyang, C., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M. and Verbeek, H.M.W. (2005) 'Formal Semantics and Analysis of Control Flow in WS-BPEL', *BPM Center Report BPM-05-15, BPM-center.org.*

Peltz, C. (2003) 'Web Service orchestration and choreography: a look at WSCI and BPEL4WS - Feature', *Web Services Journal*, Vol. 03, No. 7.

Pistore, M., Traverso, P., Bertoli, P. and Marconi, A. (2005) 'Automated Synthesis of Composite BPEL4WS Web Services', *4rd IEEE International Conference on Web Services (ICWS'05)*, pp.293–301, Orlando, FL, USA.

Pu, G.G., Zhu, H.B., Qiu, Z.Y., Wang, S.L, Zhao, X.P. and He, J.F. (2006) 'Theoretical Foundations of Scope-based Compensable flow Languange for Web Service', *8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, LNCS 4037, pp.251–266, Bologna, Italy.

Qiu, Z.Y., Wang, S.L, Pu, G.G. and Zhao, X.P. (2005) 'Semantics of BPEL4WS-Like Fault and Compensation Handling', *International Symposium of Formal Methods Europe*, LNCS 3528, pp.350–365, Newcastle, UK.

Reynolds, M. (2003) 'The complexity of the temporal logic with "until" over general linear time', *Journal of Computer and System Sciences* , Vol. 66, No. 2, pp. 393-426.

Thatte, S. (2001) 'XLANG Web Services for Business Process Design', *http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/.*

Verbeek, H.M.W. and van der Aalst, W.M.P. (2005) 'Analyzing BPEL Processes using Petri Nets', *2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management at the Petri Nets'05*, pp.59-78.

Yan, Y.H., Cordier, M.O., Pencolé, Y. and Grastien, A. (2005) 'Monitoring Web Service Networks in a Model-based Approach', *3rd European Conference on Web Services (ECOWS'05)*, pp.192–203, Växjö, Sweden.

Appendix: ASCTL Semantics

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

$$\chi ::= true \mid false \mid D \mid \neg\chi \mid \chi \wedge \chi'$$
$$\varphi ::= true \mid false \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \mathbf{E}\gamma \mid \mathbf{A}\gamma$$
$$\gamma ::= [\varphi\{\chi\}\ \mathbf{U}\ \{\chi'\}\varphi'] \mid [\varphi\{\chi\}\ \underline{\mathbf{U}}\ \{\chi'\}\varphi']$$

The formula χ is *action set formula*, φ is *state formula* and γ is *path formula*. Given an LTS $M = \langle S, I, L, \delta\rangle$, 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 χ by A (denoted by $A \models \chi$), state formula φ by s (denoted by $s \models \varphi$) and path formula γ by p (denoted by $p \models \gamma$) is defined by the following semantic rules.

$A \models true$	*always*
$A \models false$	*never*
$A \models D$	*iff* $A \cap D \neq \emptyset$
$A \models \neg\chi$	*iff* $A \not\models \chi$
$A \models \chi \wedge \chi'$	*iff* $A \models \chi \wedge A \models \chi'$
$s \models true$	*always*
$s \models false$	*never*
$s \models \neg\varphi$	*iff* $s \not\models \varphi$
$s \models \varphi \wedge \varphi'$	*iff* $s \models \varphi \wedge s \models \varphi'$
$s \models \mathbf{E}\gamma$	*iff there exists a path* $p = s_1A_1s_2A_2s_3...$ *where* $s_1 = s \wedge p \models \gamma$
$s \models \mathbf{A}\gamma$	*iff for each path* $p = s_1A_1s_2A_2s_3...$ *where* $s_1 = s,\ p \models \gamma$
$p \models [\varphi\{\chi\}\ \mathbf{U}\ \{\chi'\}\varphi']$	*iff* $\exists i \geq 1, \forall j \in [1, i)\bullet$ $(A_i \models \chi' \wedge s_{i+1} \models \varphi' \wedge$ $s_i \models \varphi \wedge s_j \models \varphi \wedge A_j \models \chi)$
$p \models [\varphi\{\chi\}\ \underline{\mathbf{U}}\ \{\chi'\}\varphi']$	*iff* $p \models [\varphi\{\chi\}\ \mathbf{U}\ \{\chi'\}\varphi'] \vee$ $(\forall i \geq 1 \bullet (A_i \models \chi \wedge s_i \models \varphi))$

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.