Front. Comput. Sci. China 2008, 2(4): 344–356
DOI 10.1007/s11704-008-0039-2

RESEARCH ARTICLE

Verifying BPEL-like programs with Hoare logic

Chenguang LUO (✉)[1], Shengchao QIN[1], Zongyan QIU[2]

1 Department of Computer Science, Durham University, Durham DH1 3LE, UK
2 LMAM and Department of Informatics, School of Mathematical Sciences, Peking University, Beijing 100871, China

Abstract The WS-BPEL language has recently become a *de facto* standard for modeling Web-based business processes. One of its essential features is the fully programmable compensation mechanism. To understand it better, many recent works have mainly focused on formal semantic models for WS-BPEL. In this paper, we make one step forward by investigating the verification problem for business processes written in BPEL-like languages. We propose a set of proof rules in Hoare-logic style as an axiomatic verification system for a BPEL-like core language containing key features such as data states, fault and compensation handling. We also propose a big-step operational semantics which incorporates all these key features. Our verification rules are proven sound with respect to this underlying semantics. The application of the verification rules is illustrated via the proof search process for a nontrivial example.

Keywords WS-BPEL, compensation mechanism, operational semantics, axiomatic verification system, soundness

1 Introduction

The Internet is now developing at a high speed supported by the web technology. As a result, many web-based applications, such as Web services, begin to flourish and play a more and more significant role in various application areas. Web services boost a new approach to the construction of business processes where many basic functions are encapsulated and provided as individual services on the web, which later may be composed to form complex services according to diverse clients' demands. To cater for the description of Web service composition, researchers and industrial practitioners have proposed several Web service orchestration languages such as XLANG [1], WSFL [2], StAC [3], and WS-BPEL [4,5].

Received July 19, 2008; accepted September 26, 2008

E-mail: {chenguang.luo, shengchao.qin}@durham.ac.uk,
zyqiu@ pku.edu.cn

Among these orchestration languages, WS-BPEL has now become a *de facto* standard. One important feature of WS-BPEL, as well as of some other similar languages, is its mechanism for supporting long run transactions (LRTs). In any single step of an LRT, a fault may occur and appropriate compensation actions may be required. To address such demand, WS-BPEL provides a set of scope-based fault handling and compensation mechanisms to deal with faults and potential undoing of some already completed business activities. The compensation mechanisms are fully programmable, and thus allow users to define any application-specific compensation rules. Nevertheless, these mechanisms, despite very flexible and powerful, also bring intricacies into the WS-BPEL language specification. As a result, it becomes a challenging issue to formalize and reason about WS-BPEL processes.

Many recent works focused mostly on the formal semantics for WS-BPEL, e.g. [6–10]. These pioneering works are very important for reducing possible ambiguity in the language specification and also for better understanding of the language. In this paper we will target at an orthogonal but equally important problem, the partial correctness of WS-BPEL processes. To make the presentation simple, we shall focus on a subset of WS-BPEL. However, our core language will take into account most of the important language features of WS-BPEL, including data state, fault handling and compensation mechanism. We will design a concise yet novel operational semantics for our language, and propose a Hoare logic style verification system on top of it, which will be proven sound with respect to the underlying semantics. Due to the complexity of web-based business processes, the correctness of such programs remains a challenge. Our verification system for BPEL-like language makes one step forward towards tackling this challenging problem. To the best of our knowledge, this is the first axiomatic verification system for a language with data states, scope-based fault and compensation handling mechanisms. The main contributions of this paper can be summarized as follows:

- We propose a concise yet novel operational semantics for a BPEL-like core language. Although there are some semantic works with similar topics, our semantics is interesting in that it integrates features like scopes, data states, fault handling and compensation in a very simple way.
- We design an assertion language for specifying certain safety properties for BPEL-like processes, and also propose a set of axioms and inference rules in Hoare logic style to form an axiomatic verification system for the language. The pre- and post-conditions are formulas expressed in our assertion language.
- We state and prove the soundness of our axiomatic verification system with respect to the semantics. That is, provable specifications are all semantically valid. A nontrivial example is presented to illustrate the application of the verification rules.

The remainder of this paper is organized as follows. Section 2 introduces our language *BPEL** which is a core subset of WS-BPEL. A new operational semantics for *BPEL** is then presented in Section 3. Section 4 is devoted to the Hoare logic style verification system for *BPEL**. Section 5 deals with the soundness of our verification system, while Section 6 gives a nontrivial example proof using our verification system. Related works and concluding remarks follow afterwards.

2 The *BPEL** language

To concentrate on the main aim of this study, we take into account a core subset of the WS-BPEL language, called *BPEL**, which comprises not only the important fault and compensation handling mechanisms but also data states of WS-BPEL processes.

The abstract syntax of *BPEL** is given in Fig. 1. Note that a program written in *BPEL** is called a *business process* (denoted as *BP*) which may contain an activity A and a fault handler F. We may sometimes use the general term *process* to refer to an activity A, a compensation handler C, or a fault handler F. The set of all processes is denoted as $\mathbb{P}$.

In Fig. 1, x and y stand for variable names, e represents arithmetic expressions, b is for boolean expressions, and n for scope names. A denotes a general activity, while C and F are for compensation and fault handlers, respectively. It is worth noting that the compensation activity $\urcorner n$ can only appear in these two constructs.

Generally a business process has an activity to perform its normal work, and once an error occurs it uses the fault handler to deal with it. As for a general activity, the skip does nothing to a process, and an assignment simply overwrites a variable's value to user's intention.

The inv, rec and rep constitute our abstract model of communication with external Web services. The a here is

$$
\begin{array}{lll}
BP & ::= \{\!| A : F |\!\} & \text{(business process)} \\
A & ::= \ \mathsf{skip} & \text{(do nothing)} \\
 & \ \ | \ x := e & \text{(assignment)} \\
 & \ \ | \ \mathsf{inv}\ a\ x\ y & \text{(invoke)} \\
 & \ \ | \ \mathsf{rec}\ a\ y & \text{(receive)} \\
 & \ \ | \ \mathsf{rep}\ a\ x & \text{(reply)} \\
 & \ \ | \ \mathsf{throw} & \text{(throw a fault)} \\
 & \ \ | \ A; A & \text{(sequence)} \\
 & \ \ | \ \mathsf{if}\ b\ \mathsf{then}\ A\ \mathsf{else}\ A & \text{(conditional)} \\
 & \ \ | \ A \parallel A & \text{(flow)} \\
 & \ \ | \ n : \{A \ ?\ C : F\} & \text{(scope)} \\
C, F ::= & \ \urcorner n & \text{(compensation)} \\
 & \ \ | \ \ldots & \text{(similar as A)}
\end{array}
$$

Fig. 1 The syntax of *BPEL**

an abstraction of call port of some Web service beyond the current process we are interested in. In our work we have taken a most general web model, where "a" is assumed to have some arbitrary behavior as far as the current business process is concerned, as it either returns an arbitrary value or fails. It is also possible to take more specialized models, which is out of the consideration of this paper.

The throw throws a fault to prevent any other activity in the current scope from being processed, until the end of the business process, or it is captured somewhere by a fault handler.

The $A \parallel A$ is a simplification of activities' parallel composition (flow). To focus more on the novel aspects of WS-BPEL, including the fault and compensation handling, we put some restrictions over this construct so that links between its components (i.e. additional control-flow restrictions) are disallowed in *BPEL**. We can do so because this issue is almost orthogonal to our focus in this paper and it has already been well investigated by researchers, e.g. [11,12].

Inside a scope n: $\{A \ ?\ C : F\}$, A is the normal activity, C is the compensation handler, and F is the fault handler. In *BPEL**, we assume all names for variables defined in a business process are distinct, so are the scope names. This is just for simplicity and does not lose generality as we can easily achieve this by a pre-processing step. Under such assumptions, we can refer to a variable or a scope simply by its name, with no need of mentioning its enclosing context. We also assume that the processes under consideration have been statically checked to meet certain basic well-formed conditions. For instance, the compensation activity $\urcorner n$ will only occur in the immediate enclosing scope of the scope n.

3 Dynamic semantics

In this section, we propose a big-step operational semantics for *BPEL**. The semantics not only serves as a runtime model for the language, but also acts as a reference

semantics in the soundness proof for our axiomatic verification system. In what follows, we will define the runtime states used for the semantics and then depict the semantic rules.

3.1 Runtime states

The nontrivial business processes need often to support long-running transactions (LRTs), where the exceptional faults are unavoidable, and as a result the partially completed tasks may need to be revoked accordingly. This kind of processes is hard to describe without language support. WS-BPEL deals with this necessity with its scope and compensation mechanism, which can be invoked to reverse partially completed transactions. Since a fault may happen from time to time, the WS-BPEL specification advocates to keep records of state snapshots for the successfully completed scopes, as the associated compensation handlers may refer to such completion states when the compensation is invoked. Our semantics will record those successfully completed scope snapshots in the runtime state, similar to the way used in Qiu et al. [6] for recording compensation closures. To facilitate the handling of faults, we also instrument the runtime state with a boolean value to indicate whether the current state is a normal state or a faulty one. The formal notations we use are as follows:

$$
\begin{aligned}
f \in \mathit{Status} &=_{df} \{\mathsf{fail}, \mathsf{norm}\} \\
s \in \mathit{Val} &=_{df} \mathit{Var} \rightharpoonup \mathit{Value} \\
\alpha, [\delta, \ldots, \delta] \in \mathit{CPCtx} &=_{df} \mathsf{seq}\ \mathit{CPCl} \\
\delta, \langle n, s, \alpha \rangle \in \mathit{CPCl} &=_{df} \mathit{ScopeN} \times \mathit{Val} \times \mathit{CPCtx} \\
\sigma, (f, s, \alpha) \in \Sigma &=_{df} \mathit{Status} \times \mathit{Val} \times \mathit{CPCtx}
\end{aligned}
$$

In the semantic model, a runtime state $\sigma = (f,s,\alpha)$ is composed of three elements, where f indicates whether the current state is normal ($f = \mathsf{norm}$) or of a fault ($f = \mathsf{fail}$), and s records current snapshot for the values of all variables in the process. The third element α is the compensation context used to record the state snapshots and relative compensation information for successfully completed scopes.

When a compensation activity $\urcorner n$ runs, the code to be executed (i.e. the compensation handler defined in scope n) is statically determined. However, the behavior of the compensation will depend on not only the scope snapshot of n, but also the dynamic execution of the normal activity in scope n that yields the state snapshot. This is due to the fact that (1) the current compensation may invoke compensation handlers from the immediate sub-scopes of n, so its behavior will depend on whether or not each of the sub-scopes has completed successfully (thus the associative compensation handler has been installed), and (2) such information is determined dynamically during the execution of the normal activity of scope n. To record such information along with the scope snapshot, we define the compensation context α as a (possibly empty) sequence of compensation closures $[\delta_1,\delta_2,\ldots,\delta_n]$, whereby a compensation closure $\delta_i = \langle n,s,\alpha_1 \rangle$ is a nested structure which records the state snapshot s for scope n (i.e., the data state at the end of the normal execution of scope n). The third element α_1 is the compensation context accumulated during the execution of the normal activity of scope n. It includes all the compensation closures for those normally completed immediately-enclosed sub-scopes. When the compensation handler of n is invoked, both the scope snapshot s and the enclosed context α_1 are restored for the compensation activity.

We do not record the handlers in the context as such information can be statically determined for a given business process. Instead, we assume the availability of a mapping to fetch the corresponding handlers:

$$\mathcal{C} : \mathit{ScopeN} \rightarrow \mathbb{P}$$

where ScopeN is the set of scope names. For a valid scope name $n \in \mathsf{dom}(\mathcal{C})$, $\mathcal{C}(n) \in \mathbb{P}$ is the compensation handler defined in scope n.

We will make use of standard sequence operators given below (where $\alpha_1 = [\delta_1,\ldots,\delta_m]$ and $\alpha_2 = [\delta'_1,\ldots,\delta'_n]$):

$$
\begin{aligned}
\delta_0 \cdot \alpha_1 &= [\delta_0, \delta_1, \ldots, \delta_m] \\
\mathsf{hd}(\alpha_1) &= \delta_1 \\
\mathsf{tl}(\alpha_1) &= [\delta_2, \ldots, \delta_m] \\
\alpha_1 \frown \alpha_2 &= [\delta_1, \ldots, \delta_m, \delta_1', \ldots, \delta_n']
\end{aligned}
$$

We define a membership relation as follows:

$$
\delta \in \alpha =_{df} \begin{cases} \mathsf{false} & \mathit{if}\ \alpha = [\,] \\ \mathsf{true} & \mathit{if}\ \mathsf{hd}(\alpha) = \delta \\ \delta \in \mathsf{tl}(\alpha) & \mathit{else} \end{cases}
$$

$$\delta \notin \alpha =_{df} \neg(\delta \in \alpha)$$

Based on it we can define the following analogous relation:

$$
\begin{aligned}
n \in \alpha &=_{df} \exists s, \alpha_1 \cdot \langle n, s, \alpha_1 \rangle \in \alpha \\
n \notin \alpha &=_{df} \neg(n \in \alpha)
\end{aligned}
$$

where n is a scope name and α is a compensation context. Informally, $n \in \alpha$ indicates that the compensation handler for the scope n has been installed (and hence n's scope snapshot appears in α).

3.2 Operational semantics

In this subsection, we present the semantic rules for the processes in *BPEL**. The big-step operational semantics for *BPEL** is defined by a set of rules of the form

$$\langle A, \sigma \rangle \rightsquigarrow \sigma'$$

where A is a process, while σ and σ' denote the initial and final states, respectively.

When a fault has occurred, the process to be executed will do nothing but propagate the fault. The rule below describes this scenario:

$$\frac{\sigma = (\mathsf{fail}, s, \alpha)}{\langle A, \sigma\rangle \rightsquigarrow \sigma}$$

The following rules define the behavior of skip, assignment, and throw activities from normal states:

$$\langle \mathsf{skip}, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (\mathsf{norm}, s, \alpha)$$
$$\langle x := e, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (\mathsf{norm}, s \oplus \{x \mapsto s(e)\}, \alpha)$$
$$\langle \mathsf{throw}, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (\mathsf{fail}, s, \alpha)$$

where $s \oplus s'$ is a state formed by s and s':

$$(s \oplus s')(x) =_{df} \begin{cases} s'(x) & \text{when } x \in \mathsf{dom}\ s' \\ s(x) & \text{otherwise} \end{cases}$$

With $s(e)$ to denote the value of expression e under state s, the skip and assignment are analogous to normal imperative language. The throw here changes the process faulty state to fail immediately, resulting in its propagation to all following activities until the end of the enclosing scope or the whole business process, where it will be dealt with by the fault handler.

When synchronized communication activity inv $a\ x\ y$ succeeds, the value received from a, the other end of communication, is assigned to y, while failed communication also makes the process fail.

$$\langle \mathsf{inv}\ a\ x\ y, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (\mathsf{norm}, s \oplus \{y \mapsto v\}, \alpha)$$
$$\langle \mathsf{inv}\ a\ x\ y, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (\mathsf{fail}, s, \alpha)$$

where v is the value achieved through the communication.

The rules for the one-way communications rec $a\ y$ and rep $a\ x$ are as follows:

$$\langle \mathsf{rec}\ a\ y, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (\mathsf{norm}, s \oplus \{y \mapsto v\}, \alpha)$$
$$\langle \mathsf{rec}\ a\ y, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (\mathsf{fail}, s, \alpha)$$
$$\langle \mathsf{rep}\ a\ x, (f, s, \alpha)\rangle \rightsquigarrow (f, s, \alpha)$$

Note that the one-way communications provide an invocation mechanism for external Web services. The rec $a\ y$ is used to retrieve parameters from other Web services (a). Its effect is to update variable y using the value received from the external Web service. On the contrary, the rep $a\ x$ replies to other external Web services (a) with the value of x. Thus its effect is just like a skip to the current process.

Rules for sequence and conditional activities are routine:

$$\frac{\langle A_1, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (f_1, s_1, \alpha_1) \quad \langle A_2, (f_1, s_1, \alpha_1)\rangle \rightsquigarrow (f_2, s_2, \alpha_2)}{\langle A_1; A_2, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (f_2, s_2, \alpha_2)}$$

$$\frac{s(b) = \mathsf{true} \quad \langle A_1, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (f_1, s_1, \alpha_1)}{\langle \mathsf{if}\ b\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (f_1, s_1, \alpha_1)}$$

$$\frac{s(b) = \mathsf{false} \quad \langle A_2, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (f_1, s_1, \alpha_1)}{\langle \mathsf{if}\ b\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (f_1, s_1, \alpha_1)}$$

The rule for the parallel composition is as follows:

$$\frac{\begin{array}{c}(s_1, s_2) = split(s, Var(A_1), Var(A_2)) \\ \langle A_1, (\mathsf{norm}, s_1, [\,])\rangle \rightsquigarrow (f_1, s'_1, \alpha_1) \\ \langle A_2, (\mathsf{norm}, s_2, [\,])\rangle \rightsquigarrow (f_2, s'_2, \alpha_2) \\ f' = f_1 \wedge f_2 \quad s' = s'_1 \cup s'_2 \quad \alpha' = interleave(\alpha_1, \alpha_2) \frown \alpha\end{array}}{\langle A_1 \parallel A_2, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (f', s', \alpha')}$$

where the splitting of variable mappings is based on the separation of variable names:

$$split(s, Var(A_1), Var(A_2)) =_{df} (\{x_1 \mapsto e_1 \mid x_1 \in Var(A_1) \wedge x_1 \mapsto e_1 \in s\} \quad \{x_2 \mapsto e_2 \mid x_2 \in Var(A_2) \wedge x_2 \mapsto e_2 \in s\})$$

in which $Var(A_1) \cap Var(A_2) = \emptyset$. And for f_1 and f_2, $f_1 \wedge f_2$ is defined as

$$f_1 \wedge f_2 =_{df} \begin{cases} \mathsf{norm}, \text{if } f_1 = \mathsf{norm} \text{ and } f_2 = \mathsf{norm}; \\ \mathsf{fail}, \text{otherwise.} \end{cases}$$

The initial sub-states s_1 and s_2 are obtained from the overall state s via a splitting operation whose definition is straightforward given that A_1 and A_2 do not share variables, i.e., $Var(A_1) \cap Var(A_2) = \emptyset$. The function *interleave* (α_1, α_2) returns a merged sequence of α_1 and α_2 by arbitrarily interleaving elements of α_1 and α_2:

$$interleave(\alpha_1, \alpha_2) =_{df} [\delta_1, \delta_2, \ldots, \delta_{m+n}]$$

where we denote $\alpha_1 = [\delta_{1,1}, \delta_{1,2}, \ldots, \delta_{1,m}]$ and $\alpha_2 = [\delta_{2,1}, \delta_{2,2}, \ldots \delta_{2,n}]$, and then the following holds: $\delta_i \in \alpha_1 \frown \alpha_2$, $i = 1,2,\ldots,m+n$; $\forall 1 \leqslant i < j \leqslant m+n$, if δ_i, $\delta_j \in \alpha_1$, $\delta_i = \delta_{1,s}$ and $\delta_j = \delta_{1,t}$, then $s < t$; and the same condition for α_2.

The execution of a scope n: $\{A\ ?\ C : F\}$ may result in two different situations: the execution of A may complete successfully or raise a fault. For the former, the compensation handler will be installed by adding the compensation closure into the compensation context. For the latter, the fault handler is invoked instead.

$$\frac{\langle A, (\mathsf{norm}, s, [\,])\rangle \rightsquigarrow (\mathsf{norm}, s_1, \alpha_1) \quad s' = s_1 \rfloor_{V(n)}}{\langle n : \{A\ ?\ C : F\}, (\mathsf{norm}, s, \alpha)\rangle \rightsquigarrow (\mathsf{norm}, s_1, \langle n, s', \alpha_1\rangle \cdot \alpha)}$$

$$\frac{\langle A, (\mathsf{norm}, s, [\,]) \rangle \rightsquigarrow (\mathsf{fail}, s_1, \alpha_1) \quad \langle F, (\mathsf{norm}, s_1, \alpha_1) \rangle \rightsquigarrow (f_2, s_2, \alpha_2)}{\langle n : \{A \ ? \ C : F\}, (\mathsf{norm}, s, \alpha) \rangle \rightsquigarrow (f_2, s_2, \alpha_2)}$$

Here $V(n)$ denotes the set of local variables of scope n, and $s_1\rfloor_{V(n)}$ takes the part of state local to n, which is the snapshot of scope n when it completes execution.

Note that the scope is the only part in the model to deal with faults. Once a fault is propagated from an activity A to its enclosing scope, it will be caught by the relevant fault handler F. If the fault handler of the immediately enclosing scope of A throws the fault again rather than completes the handling, the fault continues its propagation to the next fault handler, or meets the end of the process. This is elaborated in the rules defined above.

Next comes the definition of compensation. According to the WS-BPEL Specification [4], our compensation looks for the installed compensation closure of corresponding scope, removes it from the compensation context and runs its handler. If the closure is not installed, the invocation behaves like a skip. Since we have actually accumulated the compensation contexts, it turns out simple to execute the handler as below:

$$\frac{n \notin \alpha}{\langle n, (\mathsf{norm}, s, \alpha) \rangle \rightsquigarrow (\mathsf{norm}, s, \alpha)}$$

$$\frac{\sigma = (\mathsf{norm}, s, \alpha_1 \frown [\langle n, s', \beta \rangle] \frown \alpha_2) \quad \langle \mathcal{C}(n), (\mathsf{norm}, s \oplus s', \beta) \rangle \rightsquigarrow (f_1, s_1, \gamma)}{\langle n, \sigma \rangle \rightsquigarrow (f_1, s_1, \alpha_1 \frown \alpha_2)}$$

Note that $n \notin \alpha$, defined in last section, means that the compensation handler for n is not installed (hence the closure for n does not appear in α).

The rules for the whole business process are as follows:

$$\frac{\langle A, \sigma \rangle \rightsquigarrow (\mathsf{norm}, s_1, \alpha_1)}{\langle \{|A : F|\}, \sigma \rangle \rightsquigarrow (\mathsf{norm}, s_1, \alpha_1)}$$

$$\frac{\langle A, \sigma \rangle \rightsquigarrow (\mathsf{fail}, s_1, \alpha_1) \quad \langle F, (\mathsf{norm}, s_1, \alpha_1) \rangle \rightsquigarrow (f_2, s_2, \alpha_2)}{\langle \{|A : F|\}, \sigma \rangle \rightsquigarrow (f_2, s_2, \alpha_2)}$$

There is no top-level compensation handler in the business process because no one could invoke it if there were any.

4 An axiomatic system for *BPEL**

As a first step to support mechanized verification for *BPEL** processes, we propose in this section a set of inference rules in the style of a Floyd-Hoare logic.

4.1 Assertion language

To specify properties for *BPEL** processes, apart from the usual logical operations, we shall make use of some logical constructs that are specific for compensation related reasoning. The syntax for the assertion language *Assn* is:

$$P \in Assn$$

$$P ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{normal} \mid x \ominus e \mid \sim P \mid P_{\epsilon} \mid P\rfloor_V \mid P_{+n} \mid P_{-n} \mid P_{\restriction n} \mid P_{*n} \mid P \| P \mid P \star P \mid P * P \mid \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P$$

Note that x, e and n denote a variable name, an expression and a scope name, respectively. The $\ominus$ denotes a relational operator in $\{=, <, >, \leqslant, \geqslant\}$.

In the axiomatic system, each assertion is viewed as a set of states that satisfy the assertion. The semantics for all assertions is given in Fig. 2.

Among all assertion constructs, true and false are modeled as the whole and empty sets of states, respectively. Semantics for normal contains all states without fault. Assertion $x \ominus e$ can be in forms $x < e$, $x = e$, $x > e$ and so forth, to model the relationship between variable x and expression e.

To facilitate the description, we use here (and below) $\sigma.i$ to denote the i-th element of tuple σ. For instance, given $\sigma = (f, s, \alpha)$, we will have $\sigma.1 = f$, $\sigma.2 = s$ and $\sigma.3 = \alpha$. In the definition, $n \in \sigma.3$, defined in last section, denotes that the compensation handler for scope n is installed in σ. We also use three operations to extract information w.r.t. scope n from compensation context α:

$$\begin{aligned}
[\![\mathsf{true}]\!] &= \Sigma \\
[\![\mathsf{false}]\!] &= \emptyset \\
[\![x]\!]\sigma &= \sigma.2(x) \\
[\![\mathsf{normal}]\!] &= \{\sigma \mid \sigma.1 = \mathsf{norm}\} \\
[\![e]\!]\sigma &= \sigma.2(e) \text{ which is the evaluation result of } e \text{ under state } \sigma \\
[\![x \ominus e]\!] &= \{\sigma \mid [\![x]\!]\sigma \ominus [\![e]\!]\sigma\}, \text{where } \ominus \text{ has the semantics of the relational operator} \\
[\![\sim P]\!] &= \{(\neg\sigma.1, \sigma.2, \sigma.3) \mid \sigma \in [\![P]\!]\} \\
[\![P_{\epsilon}]\!] &= \{(\sigma.1, \sigma.2, [\,]) \mid \sigma \in P\} \\
[\![P\rfloor_V]\!] &= \{(\sigma.1, \sigma.2\rfloor_V, \sigma.3) \mid \sigma \in [\![P]\!]\} \\
[\![P_{+n}]\!] &= \{(\sigma.1, \sigma.2, \langle n, \sigma.2\rfloor_{V(n)}, \sigma.3\rangle) \mid \sigma \in [\![P]\!]\} \\
[\![P_{-n}]\!] &= \{(\sigma.1, \sigma.2, \alpha) \mid \sigma \in [\![P]\!] \wedge \alpha = \mathit{before}(n, \sigma.3) \frown \mathit{after}(n, \sigma.3)\} \\
[\![P_{\restriction n}]\!] &= \{\sigma \mid \sigma \in [\![P]\!] \wedge n \in \sigma.3\} \\
[\![P_{*n}]\!] &= \{\mathit{firstof}(n, \sigma) \mid \sigma \in [\![P]\!] \wedge n \in \sigma.3\} \\
[\![P \| Q]\!] &= \{(\sigma.1 \wedge \sigma'.1, \sigma.2 \cup \sigma'.2, \alpha) \mid \sigma \in [\![P]\!] \wedge \sigma' \in [\![Q]\!] \wedge \alpha = \mathit{interleave}(\sigma.3, \sigma'.3)\} \\
[\![P \star Q]\!] &= \{(\sigma_1.1, \sigma_1.2, \sigma_1.3 \frown \sigma_2.3) \mid \sigma_1 \in [\![P]\!] \wedge \sigma_2 \in [\![Q]\!]\} \\
[\![P * Q]\!] &= \{(\sigma_1.1, \sigma_1.2, \sigma_2.3) \mid \sigma_1 \in [\![P]\!] \wedge \sigma_2 \in [\![Q]\!]\} \\
[\![\neg P]\!] &= \Sigma \setminus [\![P]\!] \\
[\![P \wedge Q]\!] &= [\![P]\!] \cap [\![Q]\!] \\
[\![P \vee Q]\!] &= [\![P]\!] \cup [\![Q]\!] \\
[\![P \Rightarrow Q]\!] &= [\![\neg P \vee Q]\!]
\end{aligned}$$

Fig. 2 Semantics for assertions

$$firstof(n,\sigma) =_{df} (\mathsf{norm},\sigma.2\oplus s,\beta) \quad \text{if } \sigma.3=\alpha_1\frown[\langle n,s,\beta\rangle]\frown\alpha_2\wedge n\notin\alpha_1$$

$$before(n,\alpha) =_{df} \begin{cases}\alpha, & \text{if } n\notin\alpha\\ \alpha_1, & \text{if } \alpha=\alpha_1\frown[\langle n,s,\beta\rangle]\frown\alpha_2\wedge n\notin\alpha_1\end{cases}$$

$$after(n,\alpha) =_{df} \begin{cases}[\,], & \text{if } n\notin\alpha\\ \alpha_2, & \text{if } \alpha=\alpha_1\frown[\langle n,s,\beta\rangle]\frown\alpha_2\wedge n\notin\alpha_1\end{cases}$$

Operation $firstof(n,\sigma)$ extracts from $\alpha=\sigma.3$ the first state snapshot for n, and merges it with $\sigma.2$. In the case $n\notin\sigma.3$, $firstof(n,\sigma)$ is undefined. $before(n,\alpha)$ returns the largest prefix of α which contains no closure for scope n, and $after(n,\alpha)$ returns the sub-sequence of α *after* the first closure for scope n, or the empty sequence when there is no such closure in α.

Among the semantics for the assertions, some relating to flow, scope, and compensation are worth illustration.

The assertions $P\rfloor_V$ and $P\|Q$ are used in verification of flow constructs. In the first one, V is a set of variables and $P\rfloor_V$ restricts the domain of variable mapping $\sigma.2$ (where $\sigma\in\|[P]\|$) to V. For example, $(x>0\wedge y\leqslant 0)\rfloor_{\{x\}}=x>0$. The second one, $P\|Q$, enumerates all possible interleaving cases of compensation contexts of states in $\|[P]\|$ and $\|[Q]\|$, respectively.

The following assertions mainly concern scope and compensation. $\sim P$ reverses all the faulty states in each $\sigma\in\|[P]\|$ (from norm to fail and vice versa). This corresponds to the verification of throw activity and fault handler which change the process faulty state. $P_{\in}$ reserves the first and second components of states but empties their compensation contexts. This is useful for verifying scopes whose inner activity A begins with empty compensation context.

Assertion P_{+n} extracts each state σ from set $\|[P]\|$, sets its compensation context to the closure $\langle n,\sigma.2\rfloor_{V(n)},\sigma.3\rangle$, and forms a new set with all of these states.

As its form suggests, P_{-n} performs an "elimination" of scope name n "from" the elements in $\|[P]\|$. It extracts first the compensation context α from each state of $\|[P]\|$, then finds the first compensation closure with name n, and removes it to form a new context α. If there is no such closure found, then α will be the original context. The semantics of P_{-n} is the set of states with these newly formed α.

What $P_{\restriction n}$ does is, informally, to "restrict" $\|[P]\|$ to the set of states in which the compensation context contains a closure with name n, P_{*n} "locates" the first occurrence of the closure with name n in each state in $\|[P]\|$, and forms a set of states from these closures.

$P\star Q$ and $P*Q$ are for compensation contexts concatenation and replacement between assertions, respectively. The first appends the compensation contexts within Q's model to those of P's, to accumulate new compensation closure based on old ones, according to scope's behavior. The second discards directly the compensation contexts of the states in P's semantics, because of the manner of compensation handlers.

An assertion is modeled as a set containing all the states which satisfy it. Thus we define

$$\sigma\models P =_{df} \sigma\in\|[P]\|.$$

A specification in our system takes the ordinary form $\{P\}A\{Q\}$, where $P,Q\in Assn$ and $A\in\mathbb{P}$ is an activity.

One thing notable is that a business process may communicate via activities inv, rec and rep with external processes, which are essentially other Web services within the same application or from third party. As a result, whether a business process behaves in a desired way might depend on the external processes being interacted with. Hence, a business process is more like an open system which makes the verification problem rather challenging. Our proposal is to verify each business process separately according to certain dependency order in the first step. We assume that specifications for communication activities are available in the verification of one business process. When all relevant business processes have been verified separately, we can then check the consistency of all the assumptions made on communication activities. In this paper, we focus only on the verification of individual business processes.

Remembering that in the operational semantics for communications with external Web services, we have addressed that their behaviors can be arbitrary, either to deliver a value or to fail. However, to verify a business process involving communications more precisely, we need to put more restrictions over semantics of the communications. These restrictions take the form of a set of specifications $\{P\}c\{Q\}$, where each c is any one of inv $a\ x$, rec $a\ y$, or rep $a\ x$, representing a communication that might be executed by the process with the environment. We use T to denote a set of such specifications and use it as a context of the verification rules. For example, for a specification $\{P\}\mathsf{inv}\ a\ x\ y\ \{Q\}\in T$, the precondition P acts as an assertion imposed on the current process to ensure that information sent out (the value of x) satisfies the requirement of the environment, while Q acts as an assumption made on the environment: the result sent back by the environment (final value of y) satisfies the constraint described by Q, with possible substitutions of the communication channel and variable names.

The proof rules in our verification system are of the form $\mathcal{C},T\vdash\{P\}\ A\ \{Q\}$, where $\mathcal{C}$, defined earlier, is the mapping from scope names to associated compensation handlers, and T is the set of specifications defined above. We shall now present the syntax-directed proof rules in our logic.

4.2 Consequence rules

The only structural rule in our axiomatic system is the consequence rule for precondition weakening and

post-condition strengthening:

$$\frac{P \Rightarrow P' \quad \mathcal{C},T \vdash \{P'\}\ A\ \{Q'\} \quad Q' \Rightarrow Q}{\mathcal{C},T \vdash \{P\}\ A\ \{Q\}} \quad (conseq)$$

4.3 *BPEL**-specific rules

The rules for skip and assignment are simple:

$$\mathcal{C},T \vdash \{P\}\ \mathsf{skip}\ \{P\} \quad (skip)$$

$$\mathcal{C},T \vdash \{\mathsf{nomal} \wedge P\ [e/x]\}\ x := e\ \{P\} \quad (assign)$$

The rule for throw is clear too:

$$\mathcal{C},T \vdash \{P\}\ \mathsf{throw}\ \{\neg\mathsf{normal} \wedge (P \vee \sim P)\} \quad (throw)$$

Here we do not need to care whether the pre-condition is normal, because the type of fault is not in the range of our current consideration.

For the basic communication activities, the rules need to use their assumed specifications in T. For the convenience of description, we assume the variable names in the pre- and post-conditions are correspondent with those used in the invocations. Meanwhile, as is stated in former section, in the verification of the process, a triple $\{P\}A\{Q\}$ in T can also be used to verify a triple whose pre- and post-condition have the same denotation of compensation contexts, such as $\{P \star R\}A\{Q \star R\}$. And in this situation it must be guaranteed that the denotations of compensation contexts in both pre- and post-condition are the same.

If the environment can be modeled as a subset of normal, then rec sets the variable's value to what the specification denotes. Or it just propagates the fault.

$$\frac{\{\mathsf{normal}\}\ \mathsf{rec}\ a\ v\ \{Q\} \in T \quad \neg\mathsf{normal} \Rightarrow Q\ [v/y]}{\mathcal{C},T \vdash \{\mathsf{true}\}\ \mathsf{rec}\ a\ y\ \{Q\ [v/y]\}} \quad (rec)$$

where $Q\ [v/y]$ is an assertion formed by substituting each occurrence of v in Q by y, for filling the gap between the specification in T and the current process. Because of rep's analogous behavior to skip, its rule is also the same.

$$\mathcal{C},T \vdash \{P\}\ \mathsf{rep}\ a\ x\ \{P\} \quad (rep)$$

The semantics of two-way invocation is simple:

$$\frac{\{P\}\ \mathsf{inv}\ a\ u\ v\ \{Q\} \in T}{\mathcal{C},T \vdash \{P\}\ \mathsf{inv}\ a\ x\ y\ \{Q\ [u,v/x,y]\}} \quad (inv)$$

Note that these rules depend on T, the set of specifications assumed on communication activities.

The rules for control structures are as follows:

$$\frac{\neg\mathsf{normal} \wedge P \Rightarrow Q \quad \mathcal{C},T \vdash \{\mathsf{normal} \wedge P\}\ A\ \{R\} \quad \mathcal{C},T \vdash \{R\}\ B\ \{Q\}}{\mathcal{C},T \vdash \{P\}\ A;\ B\ \{Q\}} \quad (seq)$$

$$\frac{\neg\mathsf{normal} \wedge P \Rightarrow Q \quad \mathcal{C},T \vdash \{\mathsf{normal} \wedge P \wedge b\}\ A\ \{Q\} \quad \mathcal{C},T \vdash \{\mathsf{normal} \wedge P \wedge \neg b\}\ B\ \{Q\}}{\mathcal{C},T \vdash \{P\}\ \mathsf{if}\ b\ \mathsf{then}\ A\ \mathsf{else}\ B\ \{Q\}} \quad (if)$$

where b is a boolean expression of the form $x \ominus e$.

Since we assume that the different parallel flows share no variables, the rule for the parallel structures is given as

$$\frac{\neg\mathsf{normal} \wedge P \Rightarrow (Q_1 \| Q_2) \star P \quad \mathcal{C},T \vdash \{P_{\in} \rfloor_{V_1}\}\ A\ \{Q_1\} \quad \mathcal{C},T \vdash \{P_{\in} \rfloor_{V_2}\}\ B\ \{Q_2\}}{\mathcal{C},T \vdash \{P\}\ A \| B\ \{(Q_1 \| Q_2) \star P\}} \quad (flow)$$

where V_1 and V_2 are disjoint variable sets and A and B only modify variables in V_1 and V_2, respectively.

Now we present the two most significant rules, which reveal the essential features of our language. The rule for scopes is as follows:

$$\frac{\neg\mathsf{normal} \wedge P \Rightarrow Q \quad \mathcal{C},T \vdash \{\mathsf{normal} \wedge P_{\in}\}\ A\ \{R\} \quad (\mathsf{normal} \wedge R)_{+n} \star P \Rightarrow Q \quad \mathcal{C},T \vdash \{\sim(\neg\mathsf{normal} \wedge R)\}\ F\ \{Q\}}{\mathcal{C},T \vdash \{P\}\ n : \{A\ ?\ C : F\}\ \{Q\}} \quad (scope)$$

Note that the rule (*scope*) captures two cases. One stands for the scenario where a fault occurs in A. In that case the control transfers to the fault handler, and the compensation handler for scope n is not installed. The other is for the normal completion of A and the concatenation of this scope's compensation context to the process state.

Then the most intricate rule in our system, the named compensation, comes as follows:

$$\frac{\neg\mathsf{normal} \wedge P \Rightarrow Q \quad \neg P_{\restriction n} \wedge P \Rightarrow Q \quad \mathcal{C},T \vdash \left\{\left(P_{\restriction n}\right)_{*n}\right\}\ \mathcal{C}(n)\ \{R\} \quad R * P_{-n} \Rightarrow Q}{\mathcal{C},T \vdash \{P\}\ \curvearrowleft n\ \{Q\}} \quad (compensate)$$

In this rule, the behavior of a named compensation is depicted with the relevant compensation handler. If the pre-condition does not entail a scope name n, the post-condition must be automatically satisfied. Otherwise, the snapshots' set (as the pre-condition for the compensation

handler) is extracted and the post-condition is a combination of the fault and variable mapping states after the handler's execution, and the compensation context with the elimination of the first compensation closure named n.

Last is the rule for the whole business process:

$$\frac{\begin{array}{c}\mathcal{C},T \vdash \{P\}\ A\ \{R\} \\ (\mathsf{normal} \wedge R) \Rightarrow Q \\ \mathcal{C},T \vdash \{\sim(\neg\mathsf{normal} \wedge R)\}\ F\ \{Q\}\end{array}}{\mathcal{C},T \vdash \{P\}\{|A : F|\}\ \{Q\}} \quad (bp)$$

5 Soundness

This section is devoted to the soundness of our verification system. We will first give two definitions and then formalize the soundness theorem and its proof.

Definition 5.1 (Validity) We denote that a triple $\{P\}\ A\ \{Q\}$ is valid under $\mathcal{C},T$, i.e. $\mathcal{C},T \models \{P\}A\{Q\}$, if for all $\sigma \in \Sigma$, if $\sigma \models P$ and $\langle A, \sigma\rangle \rightsquigarrow \sigma'$ for some σ', then $\sigma' \models Q$.

Definition 5.2 (Soundness) Our verification system for *BPEL** is sound if all provable specifications are indeed valid, that is, if $\mathcal{C},T \vdash \{P\}A\{Q\}$, then $\mathcal{C},T \models \{P\}A\{Q\}$.

The theorem for soundness can be stated as below:

Theorem 5.1 The Hoare logic for *BPEL** presented in this paper is sound.

As is indicated by Definition 5.2 above, we need to show that, for any P, A, Q, if $\mathcal{C},T \vdash \{P\}A\{Q\}$, then $\mathcal{C},T \models \{P\}A\{Q\}$. The proof can be accomplished by structural induction over A.

Proof The verification of $\mathcal{C},T \vdash \{P\}A\{Q\}$ (denoted as t) can be a process such as

$$\frac{\begin{array}{ccc}\underline{\text{(some premises)}} & & \underline{\text{(some other premises)}} \\ \vdots & \cdots & \vdots \\ \cdots & & \cdots\end{array}}{\mathcal{C},T \vdash \{P\}\ A\ \{Q\}}$$

From the perspective of backwards reasoning, a rule r should be utilized on t according to A's structure, and from this rule some other premises need to be verified with similar backward verifications until all the premises are axioms or known facts. As an illustration, if A is $\mathcal{C},T \vdash \{P\}\{|A_1{:}F_1|\}\ \{Q\}$, then we must verify $\mathcal{C},T \vdash \{P\}A\{R\}$,$(\mathsf{normal} \wedge R) \Rightarrow Q$ and $\mathcal{C},T \vdash \{\sim(\neg\mathsf{normal} \wedge R)\}F\{Q\}$, according to the ($bp$) rule. Hence the last rule r used to verify t depends on the structure of the activity A. Therefore, the following cases are organized according to the structure of A, which is equivalent to r to some extent.

- Case (skip). The last rule r for this is (*skip*):

$$\mathcal{C},T \vdash \{P\}\ \mathsf{skip}\ \{P\}$$

Since $\langle \mathsf{skip}, \sigma\rangle \rightsquigarrow \sigma$, it is easy to see that rule (*skip*) is sound in our system.

- Case ($x := e$). The corresponding rule is (*assign*):

$$\mathcal{C},T \vdash \{\mathsf{normal} \wedge P[e/x]\}x := e\ \{P\}$$

The proof for rule (*assign*) simply follows the canonical Hoare logic's proof using the Substitution Theorem and thus is omitted here.

- Case (throw). The last rule to apply is (*throw*):

$$\mathcal{C},T \vdash \{P\}\ \mathsf{throw}\ \{\neg\mathsf{normal} \wedge (P \vee \sim P)\}$$

Take any σ such that $\sigma \models P$. If $\sigma.1 = \mathsf{fail}$, then we have $\langle \mathsf{throw}, \sigma\rangle \rightsquigarrow \sigma'$ and $\sigma \models \neg\mathsf{normal} \wedge P$. Otherwise, if $\sigma.1 = \mathsf{norm}$, then we have $\langle \mathsf{throw}, \sigma\rangle \rightsquigarrow \sigma'$ where $\sigma' = (\mathsf{fail}, \sigma.2, \sigma.3)$, and $\sigma' \models \neg\mathsf{normal} \wedge \sim P$. So we get $\mathcal{C},T \models \{P\}\mathsf{throw}\{\neg\mathsf{normal} \wedge (P \vee \sim P)\}$.

- Case (rec a y).

$$\frac{\begin{array}{c}\{\mathsf{normal}\}\ \mathsf{rec}\ a\ v\ \{Q\} \in T \\ \neg\mathsf{normal} \Rightarrow Q\ [v/y]\end{array}}{\mathcal{C},T \vdash \{\mathsf{true}\}\ \mathsf{rec}\ a\ y\ \{Q\ [v/y]\}}$$

For the proof of the rule (*rec*), if $\sigma \models \mathsf{normal}$, then since $\{\mathsf{normal}\}\ \mathsf{rec}\ a\ v\ \{Q\}$ is already known for the communication, the model of post-condition $Q\ [v/y]$ should contain the final state transited from σ (either $(\mathsf{norm}, \sigma.2 \oplus \{y \mapsto v\}, \sigma.3)$ or $(\mathsf{fail}, \sigma.2, \sigma.3)$, according to the communication's behavior). Otherwise if $\sigma \models \neg\mathsf{normal}$, then from the semantics for $\neg\mathsf{normal} \Rightarrow Q\ [v/y]$ we know $\sigma \models Q\ [v/y]$. Therefore we conclude in this case.

- Case (rep a x).

$$\mathcal{C},T \vdash \{P\}\ \mathsf{rep}\ a\ x\ \{P\}$$

Since the communication of reply does not change the process status, rule (*rep*) shares the same proof of *skip*'s.

- Case (inv a x y).

$$\frac{\{P\}\ \mathsf{inv}\ a\ u\ v\ \{Q\} \in T}{\mathcal{C},T \vdash \{P\}\ \mathsf{inv}\ a\ x\ y\ \{Q\ [u,v/x,y]\}}$$

The proof can be completed in the similar way as that of rec a y.

- Case (A; B). The rule applied in this case is (*seq*):

$$\frac{\begin{array}{c}\neg\mathsf{normal} \wedge P \Rightarrow Q \\ \mathcal{C},T \vdash \{\mathsf{normal} \wedge P\}\ A\ \{R\} \\ \mathcal{C},T \vdash \{R\}\ B\ \{Q\}\end{array}}{\mathcal{C},T \vdash \{P\}\ A;\ B\ \{Q\}}$$

The proof for rule (*seq*) is classical, except that the faulty state is taken into consideration first. That is, for any state $\sigma \models P$, if $\sigma.1 = \mathsf{fail}$, then $\sigma \models \neg\mathsf{normal} \wedge P$ and thus $\sigma \models Q$. If not, then take σ^* as $\langle A, \sigma\rangle \rightsquigarrow \sigma^*$, we have $\sigma^* \models R$. And from $\langle B, \sigma^*\rangle \rightsquigarrow \sigma'$ and the inductive assumption, it holds that $\sigma' \models Q$.

- Case (if b then A else B). In this case the condition rule (*if*) is applied:

$$\frac{\begin{array}{c}\neg\mathsf{normal} \wedge P \Rightarrow Q \\ \mathcal{C},T \vdash \{\mathsf{normal} \wedge P \wedge b\}\ A\ \{Q\} \\ \mathcal{C},T \vdash \{\mathsf{normal} \wedge P \wedge \neg b\}\ B\ \{Q\}\end{array}}{\mathcal{C},T \vdash \{P\}\ \mathsf{if}\ b\ \mathsf{then}\ A\ \mathsf{else}\ B\ \{Q\}}$$

The proof of the condition rule is also similar as the classical one. Except for the abnormal state, consider any σ where $\sigma.1 = \mathsf{norm}$. Then no matter whether $\sigma \models \mathsf{normal} \wedge P \wedge b$ or $\sigma \models \mathsf{normal} \wedge P \wedge \neg b$, for some σ' and σ^* that $\langle A, \sigma\rangle \rightsquigarrow \sigma'$ in the first case and $\langle B, \sigma\rangle \rightsquigarrow \sigma^*$ in the second, we always get $\sigma' \models Q$ and $\sigma^* \models Q$ from inductive assumption.

- Case ($A \| B$). The last rule used is (*flow*):

$$\frac{\begin{array}{c}\neg\mathsf{normal} \wedge P \Rightarrow (Q_1 \| Q_2) \star P \\ \mathcal{C},T \vdash \{P_{\in} \lfloor_{V_1}\}\ A\ \{Q_1\} \\ \mathcal{C},T \vdash \{P_{\in} \lfloor_{V_2}\}\ B\ \{Q_2\}\end{array}}{\mathcal{C},T \vdash \{P\}\ A \| B\ \{(Q_1 \| Q_2) \star P\}}$$

Take any $\sigma \models \mathsf{normal} \wedge P$ (the case for $\sigma.1 = \mathsf{fail}$ is like other rules), from the premises and the inductive assumption we know that $\langle A, (\sigma.1, \sigma.2\lfloor_{s_1}, [\,])\rangle \rightsquigarrow \sigma'_1$ and $\langle B, (\sigma.1, \sigma.2\lfloor_{s_2}, [\,])\rangle \rightsquigarrow \sigma'_2$, for some $\sigma'_1 \models Q_1$, $\sigma'_2 \models Q_2$. Hence $(\sigma'_1.1 \wedge \sigma'_2.1, \sigma'_1.2 \cup \sigma'_2.2, \mathit{interleave}(\sigma'_1.3, \sigma'_2.3) \frown \sigma.3) \models (Q_1 \| Q_2) \star P$, and thus we conclude in this case.

- Case (n: $\{A\ ?\ C : F\}$). Rule (*scope*) is the last rule applied in the proof for $\mathcal{C}, T \vdash \{P\}\ n{:}\ \{A\ ?\ C : F\}\ \{Q\}$:

$$\frac{\begin{array}{c}\neg\mathsf{normal} \wedge P \Rightarrow Q \\ \mathcal{C},T \vdash \{\mathsf{normal} \wedge P_{\in}\}\ A\ \{R\} \\ (\mathsf{normal} \wedge R)_{+n} \star P \Rightarrow Q \\ \mathcal{C},T \vdash \{\sim(\neg\mathsf{normal} \wedge R)\}\ F\ \{Q\}\end{array}}{\mathcal{C},T \vdash \{P\}\ n : \{A\ ?\ C : F\}\ \{Q\}}$$

The following cases are discussed for all $\sigma \models P$.

- If $\sigma.1 = \mathsf{fail}$, from inductive assumption and the premise $\neg\mathsf{normal} \wedge P \Rightarrow Q$, we have $\sigma \models \neg\mathsf{normal} \wedge P$, and thus $\sigma \models Q$.
- If $\sigma.1 = \mathsf{norm}$, then take $\sigma_{\in} = (\sigma.1, \sigma.2, [\,])$, and hence we have $\sigma \models \neg\mathsf{normal} \wedge P_{\in}$. With inductive assumption and the premise, denoting $\sigma'_{\in}$ as $\langle A, \sigma_{\in}\rangle \rightsquigarrow \sigma'_{\in}$, then $\sigma'_{\in} \models R$ is achieved.
  - If $\sigma'_{\in}.1 = \mathsf{norm}$, then $\sigma'_{\in} \models \mathsf{normal} \wedge R$, and $\sigma'_{+n} = (\sigma'_{\in}.1, \sigma'_{\in}.2, \langle n, \sigma'_{\in}.2\rfloor_{V(n)}, \sigma'_{\in}.3\rangle) \models (\mathsf{normal} \wedge R)_{+n}$, and still $\sigma' = (\sigma'_{+n}.1, \sigma'_{+n}.2, \sigma'_{+\in}.3 \cdot \sigma.3) \models (\mathsf{normal} \wedge R)_{+n} \star P$. We get $\sigma' \models Q$ from the last implication.
  - If $\sigma'_{\in}.1 = \mathsf{fail}$, then $\sigma'_F \models \sim(\neg\mathsf{normal} \wedge R)$ where $\sigma'_F = (\mathsf{norm}, \sigma'_{\in}.2, \sigma'_{\in}.3)$. From the inductive assumption and the semantics $\langle F, \sigma'_F\rangle \rightsquigarrow \sigma'$ for some σ', we have $\sigma' \models Q$. This completes our proof for *scope*.

- Case ($\lceil n$):

$$\frac{\begin{array}{c}\neg\mathsf{normal} \wedge P \Rightarrow Q \\ \neg P_{\restriction n} \wedge P \Rightarrow Q \\ \mathcal{C},T \vdash \left\{(P_{\restriction n})_{*n}\right\}\ C(n)\ \{R\} \\ R * P_{-n} \Rightarrow Q\end{array}}{\mathcal{C},T \vdash \{P\}\ \lceil n\ \{Q\}}$$

For the rule of compensation, consider any $\sigma \models P$ in the following cases.

- If $\sigma.1 = \mathsf{fail}$, then directly we have $\sigma \models Q$.
- If $\sigma.1 \neq \mathsf{fail}$ and there are no compensation closures named n in σ's compensation context, then $\sigma \models \neg P_{\restriction n} \wedge P$ by definition, and thus $\sigma \models Q$ which conforms to the operational semantics.
- Otherwise, we need to run the compensation handler named n. Denote $\sigma_n = \mathit{firstof}(n, \sigma)$, and we have $\sigma_n \models (P_{\restriction n})_{*n}$ and hence $\langle \mathcal{C}(n), \sigma_n\rangle \rightsquigarrow \sigma'_n$ for some σ'_n, while $\sigma'_n \models R$. Then take $\sigma = (f, \sigma, \alpha_1 \frown [\langle n, \sigma^*, \beta\rangle] \frown \alpha_2)$, and we have $\sigma' = (\sigma'_n.1, \sigma'_n.2, \alpha_1 \frown \alpha_2) \models R * P_{-n}$, and thus $\sigma' \models Q$. From all discussion above, we conclude this case.

- Case ($\{|A{:}F|\}$). The last rule applied in the proof for the whole business process will be the rule (*bp*):

$$\frac{\begin{array}{c}\mathcal{C},T \vdash \{P\}\ A\ \{R\} \\ (\mathsf{normal} \wedge R) \Rightarrow Q \\ \mathcal{C},T \vdash \{\sim(\neg\mathsf{normal} \wedge R)\}\ F\ \{Q\}\end{array}}{\mathcal{C},T \vdash \{P\}\ \{|A : F|\}\ \{Q\}}$$

- It is similar as the scope rule with compensation handler eliminated. For any $\sigma \models P$ and $\langle A, \sigma\rangle \rightsquigarrow \sigma'$ for some σ' there are the following two cases:
  - $\sigma'.1 = \mathsf{norm}$. From $(\mathsf{normal} \wedge R) \Rightarrow Q$, we know that $\sigma' \models Q$.
  - $\sigma'.1 = \mathsf{fail}$. If $\langle F, (\mathsf{norm}, \sigma'.2, \sigma'.3)\rangle \rightsquigarrow \sigma^*$ for some σ^*, then we have $\sigma^* \models Q$ from the premise $\mathcal{C},T \vdash \{\sim(\neg\mathsf{normal} \wedge R)\}\ F\ \{Q\}$.

- Besides the aforesaid A's possible structures directly related to rules, sometimes we may be not able to verify $\mathcal{C},T \vdash \{P\}\ A\ \{Q\}$ with an existing rule but can verify $\mathcal{C}, T \vdash \{P'\}A\{Q'\}$ where P' is weaker than P and/or Q' is stronger than Q. Thus the structural rule (*conseq*) is

employed in such cases:

$$\frac{P \Rightarrow P' \qquad \mathcal{C},T \vdash \{P'\}\ A\ \{Q'\} \qquad Q' \Rightarrow Q}{\mathcal{C},T \vdash \{P\}\ A\ \{Q\}}$$

For all $\sigma \models P$ and $\langle A,\sigma\rangle \rightsquigarrow \sigma'$ for some σ', we have $\sigma \models P'$ from $P \Rightarrow P'$ and also $\langle A,\sigma\rangle \rightsquigarrow \sigma^*$ for some $\sigma^* \models Q'$. Then from $Q' \Rightarrow Q$ we get $\sigma^* \models Q$. Hence σ^* is the σ' we need and the proof for this rule is completed.

Above are all the cases of our structural induction, and each of them is proven to be sound. Hence this completes our proof for the soundness.

6 Example

In this section a purchase example is exhibited to illustrate the verification of a real business process, which is a modified version of that in [13].

The general flow of the example is as follows. First the process receives price for each single item (stored in variable p) and class of the customer from other service with communication (into variable y). Then it decides the discount ratio according to the customer class, and receives the amount of items to store in t. After having all the items purchased, it computes the shipping fare according to the value of t. At last the real average price (including shipping cost) for each item is calculated and replied, which may incur fault and hence call for compensation.

This business process, denoted as BP, is written in $BPEL^*$ below.

$\{|$

$n_1 : \{\mathsf{rec}\ a\ p;\ q := p\ ?\ p := -p : \mathsf{skip}\};$

$\mathsf{rec}\ b\ y;$

$\mathsf{if}\ y = 1\ \mathsf{then}$

$n_2 : \{p := p \times 0.5\ ?\ p := p \times 2 : \mathsf{skip}\}$

else

$n_3 : \{p := p \times 0.8\ ?\ p := p \times 1.25 : \mathsf{skip}\};$

$n_4 : \{\mathsf{rec}\ c\ t;\ p := p \times t\ ?\ p := p/t : \mathsf{skip}\};$

$\mathsf{if}\ t > 500\ \mathsf{then}$

$n_5 : \{p := p + 500\ ?\ p := p - 500 : \mathsf{skip}\}$

else

$n_6 : \{p := p + t\ ?\ p := p - t : \mathsf{skip}\};$

$\mathsf{if}\ t > 0\ \mathsf{then}\ p := p/t; \mathsf{rep}\ d\ p\ \mathsf{else}\ \mathsf{throw}$

$: \overleftarrow{\neg} n_6; \overleftarrow{\neg} n_5; \overleftarrow{\neg} n_4; \overleftarrow{\neg} n_3; \overleftarrow{\neg} n_2; \overleftarrow{\neg} n_1$

$|\}$

The specification for us to verify is $\{\mathsf{normal}\}BP\{Q\}$ where Q is $p = q/2 + 500/t \vee p = 0.8q + 500/t \vee p = q/2 + 1 \vee p = 0.8q + 1 \vee p = -q$. The first four parts of the disjunctions in Q present the different situations of discount ratio and shipping fare, while the last $p = -q$ is the case where a fault is compensated. This specification states that, if BP starts in a normal state and terminates at last, it should establish the post-condition Q, provided that the specifications of the communication activities are as follows:

$$\{\mathsf{normal}\}\ \mathsf{rec}\ a\ y\ \{\mathsf{normal} \wedge y > 0\}$$
$$\{\mathsf{normal}\}\ \mathsf{rec}\ b\ y\ \{\mathsf{normal} \wedge (y = 1 \vee y = 2)\}$$
$$\{\mathsf{normal}\}\ \mathsf{rec}\ c\ y\ \{\mathsf{normal} \wedge y \neq 0\}$$

Here we give an outline of the verification for BP with the backwards searching strategy. First, for the whole business process, we use the rule of bp to get three subgoals G_1, G_2, G_3 for further verification:

$$G_1 : \quad \mathcal{C},T \vdash \{\mathsf{normal}\}\ A\ \{R\}$$
$$G_2 : \quad (\mathsf{normal} \wedge R) \Rightarrow Q$$
$$G_3 : \quad \mathcal{C},T \vdash \{\sim(\neg\mathsf{normal} \wedge R)\}\ F\ \{Q\}$$

where A stands for the codes before the last : in the process, Q is the post-condition we want to verify, F represents the six compensations ($\overleftarrow{\neg} n_6$; $\overleftarrow{\neg} n_5$; $\overleftarrow{\neg} n_4$; $\overleftarrow{\neg} n_3$; $\overleftarrow{\neg} n_2$; $\overleftarrow{\neg} n_1$), and R should be both strong enough to derive Q and still sufficiently weak as F's precondition to get Q. A possible R can be the assertion below:

$$(\mathsf{normal} \wedge (p = q/2 + 500/t \vee p = 0.8q + 500/t \vee p = q/2 + 1 \vee p = 0.8q + 1)) \vee (\neg\mathsf{normal} \wedge \sim P_6)$$

where P_6 is

$$\begin{aligned} &\mathsf{normal} \wedge \\ &((t \leq 500 \Rightarrow ((y = 1 \Rightarrow p = 0.5qt + t) \wedge \\ &\qquad (y \neq 1 \Rightarrow p = 0.8qt + t))_{+n_5} \wedge \\ &\quad t > 500 \Rightarrow ((y = 1 \Rightarrow p = 0.5qt + 500) \wedge \\ &\qquad (y \neq 1 \Rightarrow p = 0.8qt + 500))_{+n_6}) \star\ P_5) \end{aligned}$$

where P_5 stands for the set of states for compensation accumulated in the previous execution of the process, from a semantics perspective (and so are the other assertions to be depicted below); its definition is

$$\mathsf{normal} \wedge (((y = 1 \Rightarrow p = 0.5qt) \wedge (y \neq 1 \Rightarrow p = 0.8qt))_{+n_4} \star\ P_4)$$

where P_4 is

$$\begin{aligned} &\mathsf{normal} \wedge (((y = 1 \Rightarrow (p = 0.5q)_{+n_2} \wedge \\ &\qquad (y \neq 1 \Rightarrow (p = 0.8q)_{+n_3})) \star\ P_2) \end{aligned}$$

where P_2 is

$$(\mathsf{normal} \wedge p = q)_{+n_1} \star\ \mathsf{normal}$$

and the semantics and the derivations of these assertions will be introduced in the following descriptions. With them as bridges we will try to verify the three sub-goals separately.

For the first sub-goal G_1, it can still be divided into six sub-goals, since A is a sequence made up of six other activities, including two scopes, one basic communication activity and three conditional judgments. We will denote these activities as A_1, A_2, …, A_6 according to their original orders in BP, and call these six sub-goals $G_{1,i}$, $i = 1, 2, \ldots, 6$, defined as below:

$$G_{1,i} : \mathcal{C}, T \vdash \{P_i\}\ A_i\ \{P_{i+1}\}$$

where $i = 1, 2, \ldots, 6$, $P_1 = \mathsf{normal}$ and $P_7 = R$. (Note that the P_2, …, P_5 are what have been described above.) We will demonstrate the verification of each sub-goal.

For $G_{1,1}$, since A_1 is the scope n_1 and the precondition is normal, we will use the *scope* rule to divide it further into three sub-goals:

$$\begin{aligned} G_{1,1,1} :&\quad \{\mathsf{normal}_{\epsilon}\}\ \mathsf{rec}\ a\ p;\ q := p\ \{P_{1,1}\} \\ G_{1,1,2} :&\quad (\mathsf{normal} \wedge P_{1,1})_{+n_1} \star \mathsf{normal} \Rightarrow P_2 \\ G_{1,1,3} :&\quad \{\sim(\neg\mathsf{normal} \wedge P_{1,1})\}\ \mathsf{skip}\ \{P_2\} \end{aligned}$$

For the first sub-goal, the rules *seq*, *rec* and *assign* are used, to get that $P_{1,1}$ is $\mathsf{normal} \wedge p = q$. With this result and the second sub-goal the strongest P_2 is derived as $(\mathsf{normal} \wedge p = q)_{+n_1} \star \mathsf{normal}$. For the third sub-goal, since $\neg\mathsf{normal} \wedge P_{1,1} = \mathsf{false}$, it holds automatically. Then $G_{1,1}$ is verified with the post-condition P_2, that is, $(\mathsf{normal} \wedge p = q)_{+n_1} \star \mathsf{normal}$.

Next we will examine $G_{1,2}$. Here the rule of *rec* is applied to P_2 and $\mathsf{rec}\ b\ y$, with the result of post-condition P_3 which is $(y = 1 \vee y = 2) \wedge P_2$.

Sub-goal $G_{1,3}$ concerns the first if construct of the process, and its verification is an application of rule *if* with the result of two other sub-goals

$$\begin{aligned} G_{1,3,1} :&\quad \mathcal{C}, T \vdash \{\mathsf{normal} \wedge P_3 \wedge y = 1\}\ A_{1,3}\ \{P_4\} \\ G_{1,3,2} :&\quad \mathcal{C}, T \vdash \{\mathsf{normal} \wedge P_3 \wedge \neg y = 1\}\ B_{1,3}\ \{P_4\} \end{aligned}$$

where $A_{1,3}$ and $B_{1,3}$ are scopes n_2 and n_3, respectively. They can be verified similarly as n_1 (using rules *scope* and *assign*), and we get the post-condition P_4:

$$\begin{aligned} &\mathsf{normal} \wedge q = X \wedge \\ &((y = 1 \Rightarrow (p = 0.5q)_{+n_2} \wedge \\ &y \neq 1 \Rightarrow (p = 0.8q)_{+n_3}) \star P_2) \end{aligned}$$

$G_{1,4}$ is to verify the Hoare triple for scope n_4. Following similar way of $G_{1,1}$ it can be verified with P_5 as $\mathsf{normal} \wedge (((y = 1 \Rightarrow p = 0.5qt) \wedge (y \neq 1 \Rightarrow p = 0.8qt))_{+n_4} \star P)$.

$G_{1,5}$ again seeks the verification of the second if construct. With the approach like that of $G_{1,3}$ (splitting it into two subgoals) we can achieve P_6:

$$\begin{aligned} &\mathsf{normal} \wedge \\ &((t \leqslant 500 \Rightarrow ((y = 1 \Rightarrow p = 0.5qt + t) \wedge \\ &\qquad (y \neq 1 \Rightarrow p = 0.8qt + t))_{+n_5} \wedge \\ &t > 500 \Rightarrow ((y = 1 \Rightarrow p = 0.5qt + 500) \wedge \\ &\qquad (y \neq 1 \Rightarrow p = 0.8qt + 500))_{+n_6}) \star P_5) \end{aligned}$$

The last subgoal, $G_{1,6}$, is slightly different from the former two if's. It is also first divided into two subgoals:

$$\begin{aligned} &G_{1,6,1} : \mathcal{C}, T \vdash \{\mathsf{normal} \wedge P_6 \wedge t > 0\}\ p := p/t; \mathsf{rep}\ d\ p\ \{R\} \\ &G_{1,6,2} : \mathcal{C}, T \vdash \{\mathsf{normal} \wedge P_6 \wedge \neg t > 0\}\ \mathsf{throw}\ \{R\} \end{aligned}$$

in which the first subgoal's verification is as the former ones, with the post-condition $p = q/2 + 500/t \vee p = 0.8q + 500/t \vee p = q/2 + 1 \vee p = 0.8q + 1$. (Note that we omit the part for compensation and present a weaker assertion here.) However, the second one uses the *throw* rule to force a conjunction of $\neg\mathsf{normal}$ with the precondition. Therefore the whole post-condition for A, R, is as follows:

$$\begin{aligned} &(\mathsf{normal} \wedge (p = q/2 + 500/t \vee p = 0.8q + 500/t \vee \\ &p = q/2 + 1 \vee p = 0.8q + 1)) \vee (\neg\mathsf{normal} \wedge \sim P_6) \end{aligned}$$

It is clear that a conjunction of normal and this R automatically implies Q, which is demanded in the subgoal G_2. So the remaining work is to verify the subgoal G_3.

G_3 equals to $\mathcal{C}$, $T \vdash \{t < 0 \wedge P_6\}\ F\ \{Q\}$, where F is the sequence of six compensations. Similarly, this can be divided into six subgoals using the *seq* rule, and each sub-goal is solved equally with rule *compensate*. We will illustrate its usage with the first two compensations for scopes n_6 and n_5, and the others are the same as these two.

Since $t < 0$ implies that $t \leqslant 500$, it can be deducted that the compensation context for n_6 must be installed, and thus we have only two possible cases to consider ($y = 1$; $y \neq 1$). We now take the first case as an example, in which the precondition of these compensations can be reduced as

$$\begin{aligned} &\mathsf{normal} \wedge y = 1 \wedge t \leqslant 500 \wedge \\ &(p = 0.5qt + t)_{+n_6} \star (p = 0.5qt)_{+n_4} \star \\ &(p = 0.5q)_{+n_2} \star (p = q)_{+n_1} \star \mathsf{normal} \end{aligned}$$

where we denote the $\star$ as right-associative to prevent excess parentheses. Using once the rule compensate we get

$$\begin{aligned} &\mathsf{normal} \wedge y = 1 \wedge t \leqslant 500 \wedge \\ &(p = 0.5qt)_{+n_4} \star (p = 0.5q)_{+n_2} \star \\ &(p = q)_{+n_1} \star \mathsf{normal} \end{aligned}$$

to remove the compensation context of n_6 from the states (on the level of semantics).

Then for the sub-goal concerning n_5, since in this case it is not installed in the compensation context (which can be seen from the structure of the assertion), its effect, due to rule *compensate*, is like a skip.

Therefore we use the rule four times more on $\ulcorner n_i$, $i = 4 \ldots 1$ respectively, to verify each remaining subgoal and gain the final assertion

$$\mathsf{normal} \wedge y = 1 \wedge t \leqslant 500 \wedge p = -q$$

which implies that $p = -q$, and hence Q. This completes the whole process' verification.

7 Related works

The concept of compensation dates back to Sagas [14] and nested transactions [15]. There are a few attempts to formalize workflow languages [3,16–18], and still many of them are about compensation.

On the semantics of such languages there are many investigations. Qiu et al. [6] provided a formal operational semantics for a simplified version of BPEL4WS to specify the execution path of a process with possible compensation behavior. In that work, they adopted an abridgment of BPEL4WS which was followed by a series of works with similar objective. Then they provided a formal semantics for fault handling, compensations, and parallel processes with respect to the original informal description by the BPEL4WS Specification [19], with a clarification to some of its elusive parts. Meanwhile they proved the completeness of the deduction of their semantics. Besides their work, there are still many others to formalize the semantics of these languages from different perspectives. Bruni et al. [20] presented an operational semantics for a series of languages, which included the compensation concept. Pu et al. [7] also presented an abridged edition of WS-BPEL, discussed its operational semantics, and defined the equivalence between two processes with its proposed n-bisimulation. He et al. [9] focused on the process equivalence from the perspective of an observation-oriented model and its algebraic laws. Zhu et al. [10] made a link among different semantics (operational, denotational and algebraic) of the WS-BPEL language with the approach of the unifying theories of programming. Some of the semantics proposed in these works may also be enhanced as suitable underlying semantics for our verification system, while our semantics is mainly distinguished from theirs in that it incorporates variable mappings in both program runtime states and compensation contexts, enabling concrete variable valuations to be stored in the semantics.

Apart from the work on semantic models, researchers have also tried to model and verify the WS-BPEL processes. Duan et al. [21] introduced a logic model to formally specify the semantics of workflow and its composite tasks described as WS-BPEL abstract processes, and made a deduction of the weakest precondition for workflow. Another work by Duan et al. [22] put some restrictions to such model and found an algorithm to proof the abstract processes' correctness. Fu et al. [23] proposed some techniques and related tools to analyze interactions of composite Web services written in BPEL4WS. The BPEL4WS specifications [19] are first translated into an intermediate representation, and then verified use SPIN [24]. Hamadi and Benatallah [17] transformed the formal semantics of the WS-BPEL composition operators to an expression of Petri nets, and hence allowed the verification of properties and the detection of inconsistencies both within and between services. Pu et al. [25] adopted a similar method by using model checker UPPAAL [26] to verify the correctness of BPEL4WS program including temporal properties. However, none of these works have attempted in verifying WS-BPEL-like fault handling and compensation as we have done in this paper.

8 Conclusions and future works

In this paper we proposed an axiomatic system to verify the correctness of *BPEL** processes. Here we have concentrated on an important core subset of WS-BPEL, namely *BPEL**. This subset reflects the key features of the language appealing to us, say, fault states, variable mappings and compensation contexts; meanwhile it keeps our mechanism both precise and concise, rather than builds up a huge and complicated system trying to cover all aspects of WS-BPEL. We have formalized these features into program runtime states, and created *BPEL**'s operational semantics with state transition rules, according to the organization for the advancement of structured information standards (OASIS) [4]. Based on this, we have set up the assertions to abstract and express the novel language features, leaving the Hoare triples for verification and their semantics as a natural result. The verification rules for *BPEL** are also formalized after the underlying operational semantics. With respect to such semantics, we have proven the soundness of this system by structural induction on *BPEL**'s constructs, and provided an example as an illustration to the verification process of our system.

Our possible future works following this mainly include two aspects. The first is to extend the logic to cover more language features of WS-BPEL. As our original intention is to propose a concise yet novel system to verify WS-BPEL's fault handling and compensation mechanism, we omitted some language constructs which may cause the verification rule to be uncontrollable under current model, say, partner links, all compensation activities and while loops. These language features may require further invention of verification techniques such as invariants on compensation contexts, and are worth being a

natural subsequence of our further work. The second aspect aims at mechanizing the verification system for practical use, which involves some kind of verification condition generators to create the verification conditions, some reasoners to discharge the produced sub-goals, and some verification algorithm to integrate these together.

Acknowledgements We appreciate the precious comments from the anonymous reviewers. This work was supported by the UK EPSRC Project (Grant No. EP/E021948/1) and China National Natural Science Fundation (Grant No. 60773161).

References

1. Thatte S. XLANG: web service for business process design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, Microsoft, 2001
2. Leymann F. WSFL: web service flow language. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, IBM, 2001
3. Butler M, Ferreira C. An operational semantics for StAC, a language for modelling long-running business transactions. In: Proceedings of the 6th International Conference on Coordination Models and Languages, Lecture Notes in Computer Science, Vol 2949, Springer, 2004, 87–104
4. Alves A, Arkin A, Askary S, et al. web service business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, OASIS Standard, 2007
5. Barreto C, Bullard V, Erl T, et al. web service business process execution language version 2.0 primer. http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.html, OASIS Standard, 2007
6. Qiu Z, Wang S, Pu G et al. Semantics of bpel4ws-like fault and compensation handing. In: Proceedings of the 1st International Symposium of Formal Methods Europe, Lecture Notes in Computer Science, Vol 3582, Springer, 2005, 350–365
7. Pu G, Zhu H, Qiu Z et al. Theoretical foundation of scope-based compensable flow language for web service. In: Proceedings of the 1st International Conference on Formal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science, Vol 4037, Springer, 2006, 251–266
8. Qiu Z, Zhao X, Cai C et al. Towards the theoretical foundation of choreography. In: Proceedings of the 6th International World Wide Web Conference, ACM Press, 2007, 973–982
9. He J, Zhu H, Pu G. A model for bpel-like languages. Frontiers of Computer Science in China. 2007, 1(1):9–19
10. Zhu H, He J, Li J et al. Algebraic approach to linking the semantics of web services. In: Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Method, 2007, 315–328
11. Xu Q, de Roever W P, He J. The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects of Computing, 1997, 9(2): 149–174
12. Zhu H. Linking the semantics of a multithreaded discrete event simulation language. Dissertation for the Doctoral Degree. London South Bank University, 2005
13. Fowler M, Scott K. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley, 2000
14. Garcia-Molina H, Salem K. Sagas. In: Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data Conference, ACM Press, 1987, 249–259
15. Moss J. Nested transactions: an approach to reliable distributed computing. Dissertation for the Doctoral Degree. Massachusetts Institute of Technology, 1981
16. Analst W, Dumas M, Hofstede A, et al. Analysis of web services composition languages: the case of bpel4ws. In: Proceedings of the 22nd International Conference on Conceptual Modeling, Lecture Notes in Computer Science, Vol 2813. Springer, 2003, 200–215
17. Hamadi R, Benatallah B. A petri net-based model for web service composition. In: Proceedings of the 14th Australasian Database Conference, Vol 47, Adelaide, Australia, 2003, 191–200
18. Brogi A, Canal C, Pimentel E et al. Formalizing web service choreographies. Electronic Notes in Theoretical Computer Science, 2004, 105: 73–94
19. Andrews T, Curbera F, Dholakia H, et al. Business process execution language for web services 1.1. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/WS-bpel.pdf, 2003
20. Bruni R, Melgratti H, Montanari U. Theoretical foundations for compensations in flow composition languages. In: Proceedings of the 32nd SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), New York, USA, 2005, 209–220
21. Duan Z, Bernstein A, Lewis P et al. Semantics based verification and synthesis of bpel4ws abstract processes. In: Proceedings of the IEEE International Conference on Web Services, 2004, 734–737
22. Duan Z, Bernstein A, Lewis P et al. A model for abstract process specification, verification and composition. In: Proceedings of the 2nd International Conference on Service Oriented Computing, New York, USA, 2004, 232–241
23. Fu X, Bultan T, Su J. Analysis of interacting bpel web services. In: Proceedings of the 13th International World Wide Web Conference, ACM Press, 2004, 621–630
24. Holzmann G. The spin model checker:primer and reference manual. Addison-Wesley, 2003
25. Pu G, Zhao S, Wang S. Towards the semantics and verification of bpel4ws. In: Proceedings of the International Workshop on Web Languages and Formal Methods (WLFM), Electronic Notes in Theoretical Computer Science, Vol 151, Elsevier, 2005, 33–52
26. Bengtsson J, Larsen K, Larsson F, et al. Uppaal-a tool suitable for automatic verification of real-time systems. In: Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control, Secaucus, New Jersey, USA, New York: Springer, 1996, 232–243