Verifying BPEL-like Programs with Hoare Logic *

Chenguang Luo[1] Shengchao Qin[1] Zongyan Qiu[2]
[1] Department of Computer Science, Durham University
[2] LMAM and Department of Informatics, School of Math. Sciences, Peking University
{chenguang.luo, shengchao.qin}@durham.ac.uk zyqiu@pku.edu.cn

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.*

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 [17], WSFL [11], StAC [4], and WS-BPEL [2, 3].

Among these orchestration languages, WS-BPEL has now become a *de facto* standard. One important feature of WS-BPEL, as well as 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. [15, 14, 16, 10, 20]. 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 as 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

*This work is supported in part by UK EPSRC project EP/E021948/1 and China NNSF project 60773161.

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 postconditions 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. Sec 2 introduces our language *BPEL** which is a core subset of WS-BPEL. A new operational semantics for *BPEL** is then presented in Sec 3. Sec 4 is devoted to the Hoare logic style verification system for *BPEL**. Sec 5 deals with the soundness of our verification system, while Sec 6 gives a nontrivial example proof using our verification system. Related work 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 Figure 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*.

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

Figure 1. The Syntax of *BPEL**

In Figure 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 $\curvearrowleft n$ can only appear in these two constructs. Note that in 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-formedness conditions. For instance, the compensation activity $\curvearrowleft n$ will only occur in the immediate enclosing scope of the scope n.

To focus more on the novel aspects of WS-BPEL, including the fault and compensation handling, we restrict the parallel composition (flow) 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, eg. [18, 19].

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 are hard to describe without language support. WS-BPEL deals with this necessity with its scope and compensation mechanism, which can be invoked to reverse some 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. [15] 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

state. The formal notations we use are as follows:

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

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 the 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 $\lnot n$ 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 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 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 passed on.

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:

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

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 $\alpha_1 = [\delta_1, .., \delta_m]$ and $\alpha_2 = [\delta'_1, .., \delta'_n]$):

$$\begin{array}{rcl} \delta_0 \cdot \alpha_1 & = & [\delta_0, \delta_1, .., \delta_m] \\ \mathsf{hd}(\alpha_1) & = & \delta_1 \\ \mathsf{tl}(\alpha_1) & = & [\delta_2, .., \delta_m] \\ \alpha_1 \frown \alpha_2 & = & [\delta_1, .., \delta_m, \delta'_1, .., \delta'_n] \end{array}$$

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 similar relation:

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

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':

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

When synchronized communication $\mathsf{inv}\ a\ x\ y$ succeeds, the received value is assigned to y; while failed communication also makes the process fail.

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

where ν is the value achieved through the communication.

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

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

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. 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 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:

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

where 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., $\mathit{Var}(A_1) \cap \mathit{Var}(A_2) = \emptyset$. The function $\mathit{interleave}(\alpha_1, \alpha_2)$ returns a merged sequence of α_1 and α_2 by arbitrarily interleaving elements of α_1 and 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 [2], 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:

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

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

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:

$$\begin{array}{rcl} P & \in & \mathit{Assn} \\ P & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{normal} \mid x \mathbin{\widetilde{\in}} e \mid \sim P \mid P_{\epsilon} \mid P \rfloor_V \mid \\ & & P_{+n} \mid P_{-n} \mid P_{\restriction n} \mid P_{*n} \mid P \| P \mid P \star P \mid P * P \mid \\ & & \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \end{array}$$

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

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

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

Figure 2. Semantics for Assertions

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. We also use three operations to extract information w.r.t. scope n from compensation context α: Operation $\mathit{firstof}(n, \sigma)$ extracts from $\alpha = \sigma.3$ the first state snapshot for n, and merges it with $\sigma.2$:

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

When $n \notin \sigma.3$, $\mathit{firstof}(n, \sigma)$ is undefined. $\mathit{before}(n, \alpha)$ returns the largest prefix of α which contains no closure for scope n, and $\mathit{after}(n, \alpha)$ returns the sub-sequence of α *after* the first closure for scope n, or the empty sequence when no such closure in α. We omit their formal definitions here.

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, $(x{>}0 \wedge y{\leq}0)\rfloor_{\{x\}} = x{>}0$. The second one, $P \| Q$, enumerates all possible interleaving cases of compensation contexts of states in $[\![P]\!]$ and $[\![Q]\!]$, respectively.

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.

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 $P, Q \in \mathit{Assn}$ and $A \in \mathbb{P}$ is an activity.

One thing notable is that a business process may communicate with external processes via activities inv, rec and rep. 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.

For a given business process, we assume that a set of specifications $\{P\}\ c\ \{Q\}$ are known, where each c is of the form $\mathsf{inv}\ a\ x\ y$, $\mathsf{rec}\ a\ y$, or $\mathsf{rep}\ a\ x$, representing a communication that might be executed by the process with the environment. We will use T to denote a set of such specifications and pass T as a context to the verification rules. For a specification $\{P\}\ c\ \{Q\} \in T$, the precondition P acts as an assertion imposed on the current process to ensure that information sent out via c satisfies the requirement of the environment, while Q acts as an assumption made on the environment: the result sent back by the environment would satisfy Q.

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, T is the set of specifications defined above. We shall now present the syntax-directed proof rules in our logic.

4.2. Consequence Rule

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

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

4.3. *BPEL**-specific Rules

The rules for skip and assignment are simple:

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

The rule for throw is clear too:

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

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 postconditions 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 postcondition 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 postcondition 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.

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

Because of its similar behavior as skip, rep's rule is also the same.

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

The semantics of two-way invocation is simple:

$$\frac{\{P\}\ \mathsf{inv}\ a\ x\ y\ \{Q\} \in T}{\mathcal{C}, T \vdash \{P\}\ \mathsf{inv}\ a\ x\ y\ \{Q\}}(inv)$$

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

The rules for control structures are as follows.

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

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

where b is an boolean expression of the form $x \widetilde{\Theta} e$.

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

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

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:

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

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:

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

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.

At last is the rule for the whole business process:

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

5. Soundness

This section confirms the soundness of our verification system. We will first give two definitions, then formalize the soundness theorem.

Definition 1 (Validity). We denote that a specification $\{P\}\ A\ \{Q\}$ 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 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 1. *The verification system for BPEL***, given in the last section, is sound.*

As is indicated by Definition 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. Due to space limitation, we omit the proof here. A detailed proof can be found in our companion technical report [12].

6. Example

In this section a purchase example, which is a modified version of that in the book [6], is exhibited to illustrate the verification of a real business process.

The general flow of the example is as follows. First the business process receives the price for each single item (stored in variable p) and the 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 number 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 fail and hence call for compensation.

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

$\{\!|\ n_1 : \{\mathsf{rec}\ a\ p;\ q := p\ ?\ p := -p : \mathsf{skip}\};$
$\mathsf{rec}\ b\ y;$
$\mathsf{if}\ y = 1\ \mathsf{then}$
$\quad n_2 : \{p := p \times 0.5\ ?\ p := p \times 2 : \mathsf{skip}\}$
else
$\quad n_3 : \{p := p \times 0.8\ ?\ p := p \times 1.25 : \mathsf{skip}\};$
$n_4 : \{\mathsf{rec}\ c\ t;\ p := p \times t\ ?\ p := p/t : \mathsf{skip}\};$
$\mathsf{if}\ t > 500\ \mathsf{then}$
$\quad n_5 : \{p := p + 500\ ?\ p := p - 500 : \mathsf{skip}\}$
else
$\quad n_6 : \{p := p + t\ ?\ p := p - t : \mathsf{skip}\ \};$
$\mathsf{if}\ t > 0\ \mathsf{then}\ p := p/t; \mathsf{rep}\ d\ p\ \mathsf{else}\ \mathsf{throw}$
$\ :\ \ \urcorner n_6; \urcorner n_5; \urcorner n_4; \urcorner n_3; \urcorner n_2; \urcorner n_1\ \ |\!\}$

For this business process we propose a rather complicated specification to verify:

$$\{\ \mathsf{normal}\ \}\ BP\ \ \{\ p{=}q/2{+}500/t \vee p{=}0.8q{+}500/t \vee p{=}q/2{+}1 \vee p{=}0.8q{+}1 \vee p = -q\ \}$$

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, and G_3 for further verification:

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

where A stands for the codes before the last ':' in the process, Q is the postcondition we want to verify, F represents the six compensations ($\urcorner n_6; \urcorner n_5; \ldots; \urcorner n_1$), and the generated assertion R should be no stronger than the strongest postcondition for A given the precondition normal, and no weaker than (part of) the weakest precondition for F given the postcondition Q, and should establish Q given normal. Due to space limitation, the concrete representation of R is omitted here.

To discharge G_1, by the (*seq*) rule, it is sufficient to discharge six smaller subgoals (given in the report), which are then proved by several rules including (*scope*), (*rec*), (*assign*) and (*if*). the implication in G_2 is quite straightforward to prove by logic, resulting in part of the postcondition ($p{=}q/2{+}500/t \vee p{=}0.8q{+}500/t \vee p{=}q/2{+}1 \vee p{=}0.8q{+}1$). The verification of G_3 mainly utilizes the (*compensate*) rule to "consume" the compensation segments within R to achieve the last disjunctive part of postcondition ($p = -q$).

A detailed proof for this example can be found in the technical report [12] in a strict backward manner of verification.

7. Related Work

The concept of compensation dates back to Sagas [8] and nested transactions [13]. There are many attempts to formalize workflow languages [1, 9, 4], and still many of them are about compensation.

On the semantics of such languages there are many investigations. Qiu et al. [15] provided a formal operational semantics to a simplified version of WS-BPEL to specify the execution path of a process with possible compensation behavior. Pu et al. [14] also presented an abridged edition of WS-BPEL, discussed its operational semantics, and defined the equivalence between two processes with its proposed n-bi-simulation. He et al. [10] also focused on the process equivalence from the perspective of an observation-oriented model and its algebraic laws. Zhu et al. [20] made a link among different semantics (operational, denotational and algebraic) of the WS-BPEL language with the approach of the unifying theories of programming. These works can also be reference semantics for our verification system.

Apart from the work on semantic models, researchers have also tried to model and verify the WS-BPEL processes.

Duan et al. [5] 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 pre-condition for workflow. Fu et al. [7] showed some techniques to analyze and verify the WS-BPEL specified interactions among Web services with SPIN. Hamadi and Benatallah [9] 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. None of these works have attempted in verifying WS-BPEL-like fault handling and compensation as we have done in this paper.

8. Conclusion

In this paper we proposed an axiomatic system to verify the correctness of $BPEL^*$ processes. We have concentrated on a core subset of WS-BPEL, namely, $BPEL^*$, presented a complete state model including the fault state and variables for it, and created its operational semantics with state transition rules. Based on this, the assertions and Hoare triples and their semantics are set up, and the verification rules for $BPEL^*$ are formalized as well. We have also proven the soundness of this system by structural induction, and provided an example as an illustration. Possible future works include (1) extending the logic to cover more language features of WS-BPEL, and (2) mechanizing the verification system for practical use.

References

[1] W. Aalst, M. Dumas, A. Hofstede, and P. Wohed. Analysis of web services composition languages: The case of bpel4ws. In *LNCS*, volume 2813, pages 200–215. 22nd International Conference on Conceptual Modeling, Springer, 2003.

[2] A. Alves, A. Arkin, S. Askary, and et al. *Web Services Business Process Execution Language Version 2.0*. OASIS Standard, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, April 2007.

[3] C. Barreto, V. Bullard, T. Erl, and et al. *Web Services Business Process Execution Language Version 2.0 Primer*. OASIS, http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.html, May 2007.

[4] M. Butler and C. Ferreira. An operational semantics for stac, a language for modelling long-running business transactions. In *LNCS*, volume 2949, pages 87–104, Pisa, Italy, February 2004. Proceedings of Sixth International Conference on Coordination Models and Languages, Springer.

[5] Z. Duan, A. Bernstein, P. Lewis, and S. Lu. Semantics based verification and synthesis of bpel4ws abstract processes. pages 734–737. Proceedings of IEEE International Conference on Web Services, 2004, July 2004.

[6] M. Fowler and K. Scott. *UML distilled : a brief guide to the standard object modeling language*. Addison-Wesley, 2000.

[7] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. pages 621–630. Thirteenth International World Wide Web Conference (WWW 2004), ACM Press, 2004.

[8] H. Garcia-Molina and K. Salem. Sagas. pages 249–259, San Francisco, USA, May 1987. Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, ACM Press.

[9] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. volume 47, pages 191–200, Adelaide, Australia, 2003. Proceedings of the 14th Australasian database conference.

[10] J. He, H. Zhu, and G. Pu. A model for bpel-like languages. *Frontiers of Computer Science in China*, 1(1):9–19, 2007.

[11] F. Leymann. *WSFL: Web Services Flow Language*. IBM, http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001.

[12] C. Luo, S. Qin, and Z. Qiu. Verifying BPEL-like Programs with Hoare Logic. Technical report, Durham University, http://www.dur.ac.uk/shengchao.qin/papers/tase08_TR.pdf. Available also at http://www.math.pku.edu.cn:8000/var/preprint/7200.pdf, March 2008.

[13] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[14] G. Pu, H. Zhu, Z. Qiu, S. Wang, X. Zhao, and J. He. Theoretical foundation of scope-based compensable flow language for web service. In *LNCS*, volume 4037, pages 251–266. Int'l Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06), Springer, June 2006.

[15] Z. Qiu, S. Wang, G. Pu, and X. Zhao. Semantics of bpel4ws-like fault and compensation handling. In *LNCS*, volume 3582, pages 350–365. Formal Methods: Int'l Symposium of Formal Methods Europe, Springer, July 2005.

[16] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. pages 973–982. Proceedings of Sixteenth International World Wide Web Conference (WWW 2007), ACM Press, May 2007.

[17] S. Thatte. *XLANG: Web Service for Business Process Design*. Microsoft, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[18] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[19] H. Zhu. *Linking the Semantics of a Multithreaded Discrete Event Simulation Language*. PhD thesis, London South Bank University, Feburary 2005.

[20] H. Zhu, J. He, J. Li, and J. Bowen. Algebraic approach to linking the semantics of web services. pages 315–328. Fifth IEEE International Conference on Software Engineering and Formal Methods, September 2007.