Theoretical Foundations of Scope-Based Compensable Flow Language for Web Service*

Geguang Pu[1], Huibiao Zhu[1], Zongyan Qiu[2],
Shuling Wang[2], Xiangpeng Zhao[2], and Jifeng He[1]

[1] Software Engineering Institute
East China Normal University, Shanghai, China, 200062
[2] LMAM and Department of Informatics, School of Mathematical Sciences
Peking University, Beijing, China, 100871

Abstract. Web Services have become more and more important in these years, and BPEL4WS is a de facto standard for the web service composition and orchestration. In this paper, we propose a language *BPEL0* to capture the important features of BPEL4WS, with the scope-based compensation handling mechanism, which allow the users to specify the compensation behaviors of processes in application-specific manners. The operational semantics of *BPEL0* is formalized, with some key concepts related to compensation handling, i.e., the compensation closure and compensation context. Based on the achieved semantics, the concept of bisimulation in hierarchy structure is investigated, which is used to define the equivalence between *BPEL0* programs.

1 Introduction

Web services and other web-based applications have been becoming more and more important in practice. In this blooming field, various web-based business process languages are introduced, such as XLANG [19], WSFL [13], BPEL4WS (BPEL) [9], and StAC [6], which are designed for the description of services composed by a set of processes across the Internet. Their goal is to achieve the universal interoperability between applications by using web standards, as well as to specify the technical infrastructure for carrying out business transactions. However, BPEL has become the de facto standard for specifying and executing workflow specification for web service composition.

The important feature of BPEL is that it supports the stateful, long-running interactions involving two or more parties. Therefore, it provides the ability to define fault and compensation handing in application-specific manner, resulting in a feature called *Long-Running (Business) Transactions (LRTs)*. The concept *compensation* is due to the use of Sagas [11] and open nested transactions [15].

Aimed to be a language for web service composition and LRTs, BPEL provides a special form of compensation mechanism, with the *scope-based* fault and

* The authors at East China Normal University were supported by National Basic Research Program of China (No. 2002CB312001). The authors at Peking University were supported by National Natural Science Foundation of China (No. 60573081).

R. Gorrieri and H. Wehrheim (Eds.): FMOODS 2006, LNCS 4037, pp. 251–266, 2006.
© IFIP International Federation for Information Processing 2006

compensation handling. The mechanism adopted by BPEL is very flexible and powerful, and of course, it causes the complexity of the BPEL and increases the difficulty of the usage. As a result, not surprisingly, the formal semantics of scope-based workflow language, such as BPEL, is not very clear at present.

In this paper, we focus on the theoretical foundation of scope-based flow languages, and propose a language called *BPEL0* which can be regarded as the foundation of BPEL. The operational semantics of *BPEL0* is carefully studied, and with the help of the key concepts of *compensation closure* and *compensation context*, *BPEL0* clearly illustrates how the scope-based compensation mechanism works. For the discussion of the equivalence of *BPEL0* programs, which not only includes the normal programs, but also contains the compensation programs, we propose the concept of bisimulation in hierarchy structure, which reflects the scope-based compensation mechanism, as the scopes in *BPEL0* are allowed to be nested arbitrarily.

This paper is organized as follows. Section 2 introduces the *BPEL0* language with its informal illustrations. Section 3 presents the semantics of *BPEL0*. Section 4 studies the equivalence of *BPEL0* by means of bisimulation in hierarchy structure. Section 5 discusses the related work on compensational workflow language. The last section gives the conclusion and future work.

2 The *BPEL0* Language

The design of *BPEL0* is enlightened by BPEL, where the complicated XML syntactical style of BPEL is abandoned, but all the important features are included. *BPEL0* process is constructed by activities, as shown in BPEL. The syntax of *BPEL0* is as follows:

$$
\begin{array}{lll}
BA & ::= & \mathsf{skip} \mid \bar{x} := \bar{e} \mid \mathsf{wait}\ t \mid \mathsf{rec}\ a\ x \mid \mathsf{rep}\ a\ v \mid \mathsf{inv}\ a\ x\ y \mid \mathsf{throw} \mid \epsilon \\
A & ::= & BA \mid A;\ A \mid A \lhd b \rhd A \mid b * A \mid \\
 & & g \rightarrow A [\!] g \rightarrow A \mid LA \parallel_L LA \mid A \sqcap A \mid \{A\,?\,C : F\}_n \\
LA & ::= & b\ \{\check{l}_1, \check{l}_2\} \circ A \mid A \circ \{b_1 \rhd \hat{l}_1,\ b_2 \rhd \hat{l}_2\} \\
g & ::= & \mathsf{rec}\ a\ x \mid \mathsf{wait}\ t \\
C, F & ::= & \urcorner n \mid \ldots \quad \text{(similar to } A\text{)} \\
BP & ::= & \{\!| A : F |\!\}
\end{array}
$$

Basic Activities. The basic activity skip does nothing and terminates immediately. $\bar{x} := \bar{e}$ is a multiple assignment which modifies the global state of the business process. Activity wait t makes the process to wait for a given time period t. Activities rec $a\ x$ and rep $a\ v$ communicate with the environment of the business process, while inv $a\ x\ y$ calls a web service offered by its environment, with two kinds of functions: synchronous request/response or asynchronous one-way operation. Here we assume inv is a two-way operation. The behavior of one-way inv is similar to that of activity skip.

Activity throw generates a fault from inside the business process explicitly. We assume any fault produced in an activity can be captured by its corresponding fault handler when the fault handler does exist. We use ϵ to denote the empty text.

Sequential, Conditional, and Iterative Activities. A; B is the sequential composition of activities A and B. The behavior of the conditional $A \triangleleft b \triangleright B$ is the same as that of A if boolean variable b is evaluated to true, otherwise, it is the same as B. Activity $b * A$ supports repeated performance of the specified activity A, until the given boolean condition b no longer holds.

Choice Activities. *BPEL0* provides two kinds of choice: the external choice $g_1 \rightarrow A \parallel g_2 \rightarrow B$ and the internal choice $A \sqcap B$. In BPEL, there is only the external choice, which awaits the occurrence of one of a set of events and then performs the activity associated with the event that occurred. We added the internal choice into *BPEL0* to facilitate the reasoning about programs.

Flow and Link Activities. Flow activity $A \parallel_L B$ executes activities A and B in parallel, where A and B are synchronized over the link set L.

The link construct is a mechanism in BPEL to provide additional synchronization in flow activities. Each link must have exactly one activity within the flow as its source and exactly one activity as its target. The source and target of a link may be nested in arbitrary depth within the flow activity, except for the boundary-crossing restrictions [9]. To model this, two link structures $A \circ \{b_1 \triangleright \hat{l}_1,\ b_2 \triangleright \hat{l}_2\}$ and $b\ \{\check{l}_1, \check{l}_2\} \circ A$ are introduced into *BPEL0*. In fact, an activity can be the source or target of an arbitrary number of links in BPEL. We make them two here to simplify the discussion, which can be generalized.

$A \circ \{b_1 \triangleright \hat{l}_1,\ b_2 \triangleright \hat{l}_2\}$ denotes that A is the source of l_1 and l_2 which are assigned boolean values b_1 and b_2 when A completes, while in $b\ \{\check{l}_1, \check{l}_2\} \circ B$, B is the target of l_1 and l_2 with condition b. We use $\hat{l}$ and $\check{l}$ to stand for the source and target of link l respectively. Consider the following example:

$$\check{l} \circ A \parallel_{\{l\}} B \circ \{true \triangleright \hat{l}\}$$

Though activities A and B can execute in parallel if there were no link l, but now, they cannot, because the target activities of links have to wait until the link make its condition becoming true. Thus, only when B finishes and stores *true* into link l, activity A can perform its execution because link l enables its condition. Therefore, the behavior of this program is like B; A. Essentially speaking , the flow activity in *BPEL0* provides a kind of synchronization similar to the shared variable.

Suppose $l \in L$, we make l a variable recording the status of the link l. The value of l is from the three-values set $\{true,\ false,\ \oslash\}$, where $\oslash$ denotes that the status of l is not determined. The following table shows the results of the conjunction operator for the values of a link variable. Other boolean operators are defined similarly.

$\wedge$	$true$	$false$	$\oslash$
$true$	$true$	$false$	$\oslash$
$false$	$false$	$false$	$\oslash$
$\oslash$	$\oslash$	$\oslash$	$\oslash$

Scope Activity with Compensation and Fault Handlers. The interesting feature in *BPEL0* (same as in BPEL) is its scope activity, which provides fault

and compensation handlers, and both of them are important to support the Long-Running Transactions. Similar to BPEL, the compensation mechanism in *BPEL0* is:

Scope-based (not activity-based). The compensation handlers can only be attached to the scopes.
Fault triggered. A compensation handler can only be invoked directly or indirectly by some fault handler, which is triggered by a fault in the execution.
Fully programmable. The compensation handlers are named. The installed handlers can be invoked in any order, interweaved with any other activities.

$\{A\,?\,C : F\}_n$ denotes a scope with the name n. A is its primary activity, while C and F are its compensation handler and fault handler respectively. The execution of a scope is the execution of its primary activity. The compensation handler is installed with the same name as its scope when the primary activity completes its execution (terminates successfully). An installed compensation handler n is invoked by activity $\Lsh n$, which can only appear in the fault handler or compensation handler of the scope immediately enclosing the scope named n. As mentioned earlier, we suppose that any fault can be caught by the fault handler of the immediately enclosing scope.

Business Process. A complete program in *BPEL0* is in the form of a business process $\{\!|A : F|\!\}$, which is actually an outmost scope without name and compensation handler. If A completes successfully, the whole business process completes as well. While fault handler F terminates successfully when it catches the fault occurring in A, the whole business is still regarded as completed. The last case in which F terminates with a fault denotes that the whole business process terminates abnormally.

The *BPEL0* language provides almost all the features offered by BPEL except the event handlers. We present the comparison for *BPEL0*and BPELin [18].

3 Semantics

This section formalizes the operational semantics of *BPEL0*. In the semantics, the configuration is defined as a tuple:

$$\langle A,\ \sigma,\ \alpha,\ \beta\rangle \in (\mathit{Activity} \cup \{\boxtimes\}) \times \mathit{State} \times \mathit{Compensation} \times \mathit{Compensation}$$

where *Activity* is the set of program texts consisted of *BPEL0* activities or a termination mark $\boxtimes$, *State* is the set of functions from variables to values. As variables are defined in scopes, we suppose each variable is qualified with the scope name it belongs to. This means all variables are distinct in the state no matter how the scopes are nested.

The compensation context set *Compensation* is the key to deal with the scope-based compensational flow language. Contexts $\alpha,\ \beta \in \mathit{Compensation}$ are sequences of compensation closures of the form $(C_n : \alpha_1)$, where n is the same name as the scope where the handler C is defined, and α_1 is still a compensation context. When handler C is invoked, it runs in company with the context α_1.

There are two compensation contexts α and β in the configuration. As mentioned earlier, the compensation handler C in scope $\{P\,?\,C\!:\!F\}_n$ is installed only when P completes. We use α to record the accumulated compensation handlers installed in the immediately enclosing scope before the current scope starts. We call α *static compensation context*. On the other hand, β records the accumulated compensation closures during the execution of P, which can be changed with the execution of P. We denote β as the *active compensation text*. The following example illustrates the difference between α and β.

$$\{\underbrace{\{P_1\,?\,C_1\!:\!F_1\}_{n_1};A;}_{\alpha}\{\overbrace{P_2}^{\beta}\,?\,C_2\!:\!F_2\}_{n_2};\{P_3\,?\,C_3\!:\!F_3\}_{n_3}\,?\,C\!:\!F\}_n$$

In this example, when P_2 is executing, β records the compensation closures installed in the execution of P_2, while α records the context for the scope n. When the control enters scope n_3, β will be reset empty and start to record the context accumulated in the execution of P_3 in scope n_3.

$\langle \epsilon,\ \sigma,\ \alpha,\ \langle\rangle\rangle$ is a terminated configuration. As a process might complete (terminate successfully) or fail (terminate with a fault), we use $\langle \boxtimes,\ \sigma,\ \alpha,\ \langle\rangle\rangle$ to denote the failure configuration.

We distinguish three kinds of events: visible event a, time elapsing event $\surd$, and silent event τ. The visible event set mainly contains the events communicating with the external environment. The time elapsing event denotes the time elapses one time-unit in the real world. The silent event stands for a silent action of the corresponding activity. We assume that when a fault occurs in program P, the event η with fault transition belongs to $\{\tau, a\}$, which leads to make the control flow enter the fault handler from the primary activity. For simplicity, we use symbol δ to stand for an activity in $\{\tau, a, \surd\}$.

Because compensation text α is a sequence, we list some operators to deal with sequences, which will be used later.

$$a_0 \cdot \langle a_1, \ldots, a_n\rangle = \langle a_0, a_1, \ldots, a_n\rangle \qquad \langle\rangle \parallel \langle a_0, a_1, \ldots, a_n\rangle = \langle a_0, a_1, \ldots, a_n\rangle$$
$$hd(\langle a_1, a_2, \ldots, a_n\rangle) = a_1 \qquad tl(\langle a_1, a_2, \ldots, a_n\rangle) = \langle a_2, \ldots, a_n\rangle$$
$$\langle a_1, , \ldots, a_n\rangle^\frown\langle b_1, \ldots, b_m\rangle = \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$$

The operator $\parallel$ on sequence denotes two sequences are composed in parallel. If one part of parallel sequence is empty, then this part can be omitted.

3.1 Basic Activities

The semantics of basic activities are listed as follows:

$$\langle \mathsf{skip},\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{\tau} \langle \epsilon,\ \sigma,\ \alpha,\ \langle\rangle\rangle$$
$$\langle \mathsf{inv}\ a\ x\ y,\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{\surd} \langle \mathsf{inv}\ a\ x\ y,\ \sigma,\ \alpha,\ \langle\rangle\rangle$$
$$\langle \mathsf{inv}\ a\ x\ y,\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{a.v} \langle \epsilon,\ \sigma[y \mapsto v],\ \alpha,\ \langle\rangle\rangle$$
$$\langle \mathsf{rec}\ a\ x,\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{\surd} \langle \mathsf{rec}\ a\ x,\ \sigma,\ \alpha,\ \langle\rangle\rangle$$
$$\langle \mathsf{rec}\ a\ x,\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{a.v} \langle \epsilon,\ \sigma[x \mapsto v],\ \alpha,\ \langle\rangle\rangle$$
$$\langle \mathsf{rep}\ a\ x,\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{\surd} \langle \mathsf{rep}\ a\ x,\ \sigma,\ \alpha,\ \langle\rangle\rangle$$
$$\langle \mathsf{rep}\ a\ x,\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{\tau} \langle \epsilon,\ \sigma,\ \alpha,\ \langle\rangle\rangle$$

$$\langle \bar{x} := \bar{e},\ \sigma,\ \alpha,\ \langle\rangle \rangle \xrightarrow{\tau} \langle \epsilon,\ \sigma[\bar{x} \mapsto \sigma(\bar{e})],\ \alpha,\ \langle\rangle \rangle$$
$$\langle \mathsf{wait}\ t,\ \sigma,\ \alpha,\ \langle\rangle \rangle \xrightarrow{\surd} \langle \mathsf{wait}\ t-1,\ \sigma,\ \alpha,\ \langle\rangle \rangle \quad t > 1$$
$$\langle \mathsf{wait}\ 1,\ \sigma,\ \alpha,\ \langle\rangle \rangle \xrightarrow{\surd} \langle \epsilon,\ \sigma,\ \alpha,\ \langle\rangle \rangle$$
$$\langle \mathsf{inv}\ a\ x\ y,\ \sigma,\ \alpha,\ \langle\rangle \rangle \xrightarrow{a} \langle \boxtimes,\ \sigma,\ \alpha,\ \langle\rangle \rangle$$
$$\langle \mathsf{rec}\ a\ x\ y,\ \sigma,\ \alpha,\ \langle\rangle \rangle \xrightarrow{a} \langle \boxtimes,\ \sigma,\ \alpha,\ \langle\rangle \rangle$$
$$\langle \mathsf{rep}\ a\ x,\ \sigma,\ \alpha,\ \langle\rangle \rangle \xrightarrow{a} \langle \boxtimes,\ \sigma,\ \alpha,\ \langle\rangle \rangle$$

The last three rules for the basic activities show that, a fault might take place when *BPEL0* process communicates with the environment. Note that the basic activities do not embed any scope. Therefore, the active compensation context is empty for each of them.

3.2 Composition Activities

Sequence. Compared with the sequential composition in traditional programming languages, one interesting rule here is that when a fault takes place in activity A, the whole structure $A; B$ goes into the fault state immediately, where the active compensation text β is reset empty.

$$\langle \epsilon; A,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\tau} \langle A,\ \sigma,\ \alpha,\ \beta \rangle \qquad \frac{\langle A,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\delta} \langle A',\ \sigma', \alpha',\ \beta' \rangle}{\langle A; B,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\delta} \langle A'; B,\ \sigma',\ \alpha',\ \beta' \rangle}$$

$$\frac{\langle A,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\eta} \langle \boxtimes,\ \sigma,\ \alpha,\ \langle\rangle \rangle}{\langle A; B,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\eta} \langle \boxtimes,\ \sigma,\ \alpha,\ \langle\rangle \rangle}$$

For switch, iteration, external and internal choice, their transition rules can be found in [18].

Link. Link structure provides the synchronization mechanism in parallel composition of *BPEL0*.

$$\frac{\langle A,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\delta} \langle A',\ \sigma',\ \alpha',\ \beta' \rangle}{\langle A \circ \{b_1 \rhd \hat{l}_1,\ b_2 \rhd \hat{l}_2\},\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\delta} \langle A' \circ \{b_1 \rhd \hat{l}_1,\ b_2 \rhd \hat{l}_2\},\ \sigma',\ \alpha',\ \beta' \rangle}$$

$$\langle \epsilon \circ \{b_1 \rhd \hat{l}_1,\ b_2 \rhd \hat{l}_2\},\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\tau} \langle \epsilon,\ \sigma[l_1 \mapsto \sigma(b_1),\ l_2 \mapsto \sigma(b_2)],\ \alpha,\ \langle\rangle \rangle$$

$$\frac{\langle A,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\eta} \langle \boxtimes,\ \sigma', \alpha',\ \langle\rangle \rangle}{\langle A \circ \{b_1 \rhd \hat{l}_1,\ b_2 \rhd \hat{l}_2\},\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\eta} \langle \boxtimes,\ \sigma[l_1 \mapsto \mathit{false},\ l_2 \mapsto \ \mathit{false}],\ \alpha,\ \langle\rangle \rangle}$$

$$\frac{\sigma(b\{\check{l}_1,\ \check{l}_2\}) =\ \mathit{true}}{\langle b\{\check{l}_1,\ \check{l}_2\} \circ A,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\tau} \langle A,\ \sigma,\ \alpha,\ \beta \rangle}$$

$$\frac{\sigma(b\{\check{l}_1,\ \check{l}_2\}) =\ \mathit{false}}{\langle b\{\check{l}_1,\ \check{l}_2\} \circ A,\ \sigma,\ \alpha,\ \beta \rangle \xrightarrow{\tau} \langle \boxtimes,\ \sigma,\ \alpha,\ \langle\rangle \rangle}$$

Note that when a fault occurs in source link structure $A \circ \{b_1 \rhd \hat{l}_1,\ b_2 \rhd \hat{l}_2\}$, it enters the fault state with assigning *false* to all its link variables. This mechanism ensures that the target link structure can work well, even though its corresponding source link structure has a fault. If the valuation of the boolean variable in the target link is *false*, a standard fault will be thrown immediately. This rule reflects the *non dead-path-elimination semantics* in the flow structure of the BPEL.

Scope. Scope activity is one of the most important features in *BPEL0*. By means of the compensation and fault handlers with the scope activity, *BPEL0* can deal with very complicated long running transactions in business process.

$$\frac{\langle A,\ \sigma,\ \beta,\ \gamma\rangle \xrightarrow{\delta} \langle A',\ \sigma',\ \beta',\ \gamma'\rangle}{\langle \{A\,?\,C\!:\!F\}_n,\ \sigma,\ \alpha,\ \beta\rangle \xrightarrow{\delta} \langle \{A'\,?\,C\!:\!F\}_n,\ \sigma',\ \alpha,\ \beta'\rangle}$$

$$\langle \{\epsilon\,?\,C\!:\!F\}_n, \sigma,\ \alpha,\ \beta\rangle \xrightarrow{\tau} \langle \epsilon,\ \sigma,\ (C_n\!:\!\beta)\cdot\alpha, \langle\rangle\rangle$$

$$\frac{\langle A,\ \sigma,\ \beta,\ \gamma\rangle \xrightarrow{\eta} \langle \boxtimes,\ \sigma,\ \beta,\ \gamma'\rangle}{\langle \{A\,?\,C\!:\!F\}_n,\ \sigma,\ \alpha,\ \beta\rangle \xrightarrow{\eta} \langle F,\ \sigma,\ \beta,\ \langle\rangle\rangle}$$

The relation between static and active compensation contexts embodied in scope activity is that the active compensation context of $\{A\,?\,C:F\}_n$ is exactly the static compensation context of activity A.

The primary activity A is executed with an empty context initially. When it completes, a compensation closure is created, and put in the front of α. A sequence of compensation closures will accumulate in this way. When primary activity A fails, the execution switches to the fault handler, and the termination status of the fault handler F is the termination status of the scope. The fault handler can do anything to the state and the environment. Basically, it has the responsibility to recover the process back to a normal state. Note that the fault handler resets its active compensation context empty again before it starts its computing task.

Business Process. A business process is just like the scope activity except lacking of compensation handler. As business process can be regarded as the outmost scope activity, its static compensation text always keeps empty. The following rules are similar to those of scope activity.

$$\frac{\langle P,\ \sigma,\ \beta,\ \gamma\ \rangle \xrightarrow{\delta} \langle\ P',\ \sigma',\ \beta',\ \gamma'\ \rangle}{\langle\ \{\!|P:F|\!\},\ \sigma,\ \langle\rangle,\ \beta\ \rangle \xrightarrow{\delta} \langle\ \{\!|P':F|\!\},\ \sigma',\ \langle\rangle,\ \beta'\ \rangle}$$

$$\langle\ \{\!|\epsilon:F|\!\},\ \sigma,\ \langle\rangle,\ \beta\ \rangle \xrightarrow{\tau} \langle\ \epsilon,\ \sigma',\ \langle\rangle,\ \langle\rangle\ \rangle$$

$$\frac{\langle\ P,\ \sigma,\ \beta,\ \gamma\ \rangle \xrightarrow{\eta} \langle\ \boxtimes,\ \sigma,\ \beta,\ \gamma'\ \rangle}{\langle\ \{\!|P:F|\!\},\ \sigma,\ \langle\rangle,\ \beta\ \rangle \xrightarrow{\eta} \langle\ F,\ \sigma,\ \beta,\ \langle\rangle\ \rangle}$$

Flow (Parallel). The activities in flow structure are synchronized by the link set defined within parallel activity. The flow activity obeys the following rules:

$$\frac{\langle\ A,\ \sigma,\ \alpha_A,\ \beta_A\ \rangle \xrightarrow{\delta} \langle\ A',\ \sigma',\ \alpha'_A,\ \beta'_A\ \rangle \ \text{ and }\ \delta \neq \surd}{\langle\ A \parallel_L B,\ \sigma,\ (\alpha_A \parallel \alpha_B)\cdot\alpha,\ \beta_A \parallel \beta_B\ \rangle \xrightarrow{\delta} \langle\ A' \parallel_L B,\ \sigma',\ (\alpha'_A \parallel \alpha_B)\cdot\alpha,\ \beta'_A \parallel \beta_B\ \rangle}$$

$$\frac{\langle\ B,\ \sigma,\ \alpha_B,\ \beta_B\ \rangle \xrightarrow{\delta} \langle\ B',\ \sigma',\ \alpha'_B,\ \beta'_B\ \rangle \ \text{ and }\ \delta \neq \surd}{\langle\ A \parallel_L B,\ \sigma,\ (\alpha_A \parallel \alpha_B)\cdot\alpha,\ \beta_A \parallel \beta_B\ \rangle \xrightarrow{\delta} \langle\ A \parallel_L B',\ \sigma',\ (\alpha_A \parallel \alpha'_B)\cdot\alpha,\ \beta_A \parallel \beta'_B\ \rangle}$$

$$\frac{\langle\ A,\ \sigma,\ \alpha_A,\ \beta_A\ \rangle \xrightarrow{\surd} \langle\ A',\ \sigma',\ \alpha'_A,\ \beta'_A\ \rangle \text{ and } \langle\ B,\ \sigma,\ \alpha_B,\ \beta_B\ \rangle \xrightarrow{\surd} \langle\ B',\ \sigma',\ \alpha'_B,\ \beta'_B\ \rangle}{\langle\ A \parallel_L B,\ \sigma,\ (\alpha_A \parallel \alpha_B)\cdot\alpha,\ \beta_A \parallel \beta_B\ \rangle \xrightarrow{\surd} \langle\ A' \parallel_L B',\ \sigma',\ (\alpha'_A \parallel \alpha'_B)\cdot\alpha,\ \beta'_A \parallel \beta'_B\ \rangle}$$

The operator $\|$ on ϵ and $\boxtimes$ is defined in the following table:

$\|$	ϵ	$\boxtimes$
ϵ	ϵ	$\boxtimes$
$\boxtimes$	$\boxtimes$	$\boxtimes$

Only when all of activities in the flow complete, the flow activity completes. Note that there is an interesting thing about a fault occurring in one branch of the flow activity. If one branch in a flow fails, the other branches can still run until they complete or fail. This seems a little unreasonable in real system, because all branches are supposed to be terminated when one of the branch in flow fails. In the next section, the concept of *forced termination* is introduced to modify the semantics provided here in order to conform to the behavior of fault in the real system.

Operation $\Lsh n$ looks up the compensation closure with the name n in the current compensation context. If no closure with the name is found, it acts like skip, otherwise, the handler in the closure is executed in company with its context:

$$\langle \Lsh n,\ \sigma,\ \alpha,\ \langle\rangle \rangle \xrightarrow{\tau} \langle\ gp(n,\alpha),\ \sigma,\ ge(n,\alpha),\ \langle\rangle\ \rangle$$

The lookup rules for parallel operator are as follows:

$$gp(n,(\alpha' \parallel \alpha'')\cdot\alpha) = gp(n,\alpha'^\frown\alpha''^\frown\alpha) \quad \text{and} \quad ge(n,(\alpha' \parallel \alpha'')\cdot\alpha) = ge(n,\alpha'^\frown\alpha''^\frown\alpha)$$

where $gp(n,\alpha)$ and $ge(n,\alpha)$ extract the process and the context of the compensation closure with name n from α, respectively (where $n \neq m$):

$$\begin{array}{r|r} gp(n,\langle\rangle) = \mathsf{skip} & ge(n,\langle\rangle) = \langle\rangle \\ gp(n,(C_n\!:\!\beta)\cdot\alpha') = C & ge(n,(C_n\!:\!\beta)\cdot\alpha') = \beta \\ gp(n,(C_m\!:\!\beta)\cdot\alpha') = gp(n,\alpha') & ge(n,(C_m\!:\!\beta)\cdot\alpha') = ge(n,\alpha') \end{array}$$

3.3 Forced Termination

In a BPEL flow activity, when one branch fails, the fault handler of the innermost enclosing scope begins its behavior by implicitly terminating all other (concurrent) activities in the scope, and then starts the execution of its body. This is called the *forced termination*. To deal with this mechanism, a new termination mark $\overline{\boxtimes}$ is introduced to describe this new kind of termination.

We have to add some rules to handle the forced termination. First of all, all basic actives will be allowed to complete their work as before and their completion can be regarded as a forced termination as well. We use P_{ba} to denote any basic activity, such as rec, inv etc.

$$\frac{\langle P_{ba},\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{\delta} \langle\epsilon,\ \sigma',\ \alpha,\ \langle\rangle\rangle}{\langle P_{ba},\ \sigma,\ \alpha,\ \langle\rangle\rangle \xrightarrow{\delta} \langle\overline{\boxtimes},\ \sigma',\ \alpha,\ \langle\rangle\rangle}$$

A rule for sequential composition is added:

$$\frac{\langle A,\ \sigma,\ \alpha,\ \beta\rangle \xrightarrow{\delta} \langle\overline{\boxtimes},\ \sigma',\alpha,\ \langle\rangle\rangle}{\langle A;B,\ \sigma,\ \alpha,\ \beta\ \rangle\ \xrightarrow{\delta} \langle\overline{\boxtimes},\ \sigma',\ \alpha,\ \langle\rangle\rangle}$$

A rule for link construct is added:

$$\frac{\langle A,\ \sigma,\ \alpha,\ \beta\rangle \xrightarrow{\delta} \langle\overline{\boxtimes},\ \sigma', \alpha,\ \langle\rangle\rangle}{\langle A\circ\{b_1\triangleright\hat{l}_1,\ b_2\triangleright\hat{l}_2\},\ \sigma,\ \alpha,\ \beta\rangle \xrightarrow{\delta} \langle\overline{\boxtimes},\ \sigma'[l_1\mapsto false,\ l_2\mapsto\ false],\ \alpha,\ \langle\rangle\rangle}$$

A new rule for scope is added as well.

$$\frac{\langle P,\ \sigma,\ \beta,\ \gamma\rangle \xrightarrow{\delta} \langle\overline{\boxtimes},\ \sigma',\ \beta,\ \langle\rangle\rangle}{\langle\{P\,?\,C\!:\!F\}_n,\ \sigma,\ \alpha,\ \beta\rangle \xrightarrow{\delta} \langle\overline{\boxtimes},\ \sigma',\ \alpha,\ \langle\rangle\rangle}$$

At last, we should modify the results of operator $\|$ while $\overline{\boxtimes}$ is added

$\|$	ϵ	$\overline{\boxtimes}$	$\boxtimes$
ϵ	ϵ	$\overline{\boxtimes}$	$\boxtimes$
$\overline{\boxtimes}$	$\overline{\boxtimes}$	$\overline{\boxtimes}$	$\boxtimes$
$\boxtimes$	$\boxtimes$	$\boxtimes$	$\boxtimes$

Example 1. (*Forced Termination*) Consider a *BPEL0* program $P = \{\{A_1\,?\,C_1 : F_1\}_{n_1}; A_2; A_3\ \|_{\{\}}\ A_4; A_5; A_6\,?\ \mathsf{skip} : \,\Lsh n\}_n$, where A_i $(i = 1...6)$ are all basic activities. Suppose a fault occurs in the execution of A_2, and all the other basic activities can complete. Using the semantic rules above, we can reason about the execution of P. For simplicity, we use some abbreviations $P_1 = \{A_1\,?\,C_1 : F_1\}_{n_1}$, $P_{11} = \{A_1\,?\,C_1 : F_1\}_{n_1}; A_2; A_3$ and $P_{12} = A_4; A_5; A_6$. In the following , we use $\longrightarrow^*$ to denote zero or multiple transitions.

When there is no forced termination:

(1) $\langle P_1, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\epsilon, \sigma', (C_n : \langle\rangle), \langle\rangle\rangle$
(2) $\langle P_{11}, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\boxtimes, \sigma'', (C_n : \langle\rangle), \langle\rangle\rangle$
(3) $\langle P_{12}, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\epsilon, \sigma'', \langle\rangle, \langle\rangle\rangle$
(4) $\langle P_{11}\ \|_{\{\}}\ P_{12}, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\boxtimes, \sigma'', (C_n : \langle\rangle), \langle\rangle\rangle$
(5) $\langle P, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\Lsh n, \sigma'', (C_n : \langle\rangle), \langle\rangle\rangle$
(6) $\langle P, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle C_n, \sigma'', \langle\rangle, \langle\rangle\rangle$

Now we take the forced termination into account:

(1) $\langle P_1, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\epsilon, \sigma', (C_n : \langle\rangle), \langle\rangle\rangle$
(2) $\langle P_{11}, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\boxtimes, \sigma'', (C_n : \langle\rangle), \langle\rangle\rangle$
(3) $\langle P_{12}, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow \langle\epsilon; A_2; A_3, \sigma_1, \langle\rangle, \langle\rangle\rangle$
(4) $\langle P_{12}, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\overline{\boxtimes}, \sigma_1, \langle\rangle, \langle\rangle\rangle$
(5) $\langle P_{11}\ \|_{\{\}}\ P_{12}, \sigma, \langle\rangle, \langle\rangle\rangle \longrightarrow^* \langle\boxtimes, \sigma'', (C_n : \langle\rangle), \langle\rangle\rangle$

From the second deduction, we can see that when a branch in a parallel process fails, the other activities that are currently active are forced to terminate by means of force termination rules.

4 Bisimulation

The behavior of a program can be represented in terms of execution steps. Two syntactically different programs may have the same observable behavior. Thus, a reasonable abstraction is desirable in defining program equivalence via operational semantics. Bisimulation is a useful approach in defining program equivalence. Algebraic laws can be explored using the formalized bisimulation.

Here are some auxiliary definitions for the definition of bisimulation.

Definition 1. *The transition relation* $\stackrel{id}{\Longrightarrow}$ *is defined as:*

$$\langle P,\sigma,\alpha,\beta\rangle \stackrel{id}{\Longrightarrow} \langle P',\sigma,\alpha,\beta\rangle$$
$$=_{df} \exists n, P_1,\ldots,P_n \bullet \langle P,\sigma,\alpha,\beta\rangle \stackrel{\eta_1}{\longrightarrow} \langle P_1,\sigma,\alpha,\beta\rangle \;\ldots \stackrel{\eta_n}{\longrightarrow} \langle P_n,\sigma,\alpha,\beta\rangle \quad \textit{and} \quad P_n = P'$$

where $\stackrel{\eta_i}{\longrightarrow}$ *can be of the form* $\stackrel{\tau}{\longrightarrow}$ *or* $\stackrel{a}{\longrightarrow}$. □

Definition 2. *The transition relation* $\stackrel{\delta}{\Longrightarrow}$ *(*$\delta \in \{\tau, a, \surd\}$*) is defined as:*

$$\langle P,\sigma,\alpha,\beta\rangle \stackrel{\delta}{\Longrightarrow} \langle P',\sigma',\alpha',\beta'\rangle$$
$$=_{df} \begin{cases} \langle P,\sigma,\alpha,\beta\rangle \stackrel{\delta}{\longrightarrow} \langle P',\sigma',\alpha',\beta'\rangle & \text{or} \\ \exists P_1 \bullet\ \langle P,\sigma,\alpha,\beta\rangle \stackrel{id}{\Longrightarrow} \langle P_1,\sigma,\alpha,\beta\rangle \stackrel{\delta}{\longrightarrow} \langle P',\sigma',\alpha',\beta'\rangle \end{cases}$$

In a *BPEL0* program configuration, the third element stores a sequence of programs. This gives the complexity of defining bisimulation for the programs. In order to deal with the definition, we firstly introduce the concept of 0-*Bisimulation*, which forms the basis for defining program equivalence.

Definition 3. *(*0*-Bisimulation) A symmetric relation* R *is a* 0*-Bisimulation* if and only if $\forall\ \langle P,\sigma,\alpha,\beta\rangle\, R\, \langle Q,\sigma,\alpha_1,\beta_1\rangle$

(1) if $\langle P,\sigma,\alpha,\beta\rangle \stackrel{\surd}{\Longrightarrow} \langle P',\sigma',\alpha',\beta'\rangle$,
then $\exists Q',\alpha_1',\beta_1' \bullet \langle Q,\sigma,\alpha_1,\beta_1\rangle \stackrel{\surd}{\Longrightarrow} \langle Q',\sigma',\alpha_1',\beta_1'\rangle$ *and* $\langle P',\sigma',\alpha',\beta'\rangle\, R\, \langle Q',\sigma',\alpha_1',\beta_1'\rangle$

(2) if $\langle P,\sigma,\alpha,\beta\rangle \stackrel{\eta}{\Longrightarrow} \langle P',\sigma',\alpha',\beta'\rangle \quad (\eta \in \{\tau, a\})$,

(2-1) if $\sigma \neq \sigma'$, *then*
$\exists Q',\alpha_1',\beta_1' \bullet \langle Q,\sigma,\alpha_1,\beta_1\rangle \stackrel{\eta}{\Longrightarrow} \langle Q',\sigma',\alpha_1',\beta_1'\rangle$ *and* $\langle P',\sigma',\alpha',\beta'\rangle\, R\, \langle Q',\sigma',\alpha_1',\beta_1'\rangle$

(2-2) if $\sigma = \sigma'$, *then*
either $\langle P',\sigma',\alpha',\beta'\rangle\, R\, \langle Q,\sigma,\alpha_1,\beta_1\rangle$
or $\exists Q',\alpha_1',\beta_1' \bullet \langle Q,\sigma,\alpha_1,\beta_1\rangle \stackrel{\eta}{\Longrightarrow} \langle Q',\sigma',\alpha_1',\beta_1'\rangle$ *and* $\langle P',\sigma',\alpha',\beta'\rangle\, R\, \langle Q',\sigma',\alpha_1',\beta_1'\rangle$

(3) if $\langle P,\sigma,\alpha,\beta\rangle \stackrel{\eta}{\longrightarrow} \langle \boxtimes,\sigma',\alpha',\langle\rangle\rangle \quad (\eta \in \{\tau, a\})$,
then $\exists \alpha_1' \bullet \langle Q,\sigma,\alpha,\beta\rangle \stackrel{\eta}{\longrightarrow} \langle \boxtimes,\sigma',\alpha_1',\langle\rangle\rangle$ □

Item (1) indicates that if process P makes time transition, so does the process Q and the two result configurations are also 0-bisimilar.

Item (2) stands for the case of atomic-like transitions. It can be divided into two types. If the two states before and after the transition are different, the bisimilarity analysis is similar to item (1). The second type models the case that the two states are the same. For this sub case, although process P has made transitions, process Q may not make further transitions and the result

configuration of P is directly bisimilar to the configuration of process Q. On the other hand, process Q may also need to do atomic-like transition and the result configurations of process P and Q after transitions are bisimilar.

Item (3) represents the failure case. If a process makes a failure transition, the corresponding process must also make a failure transition.

Definition 4. *(1) Configurations* $\langle P_1, \sigma, \alpha_1, \beta_1\rangle$ *and* $\langle P_2, \sigma, \alpha_2, \beta_2\rangle$ *are* 0*-bisimilar, written as* $\langle P_1, \sigma, \alpha_1, \beta_1\rangle \approx_0 \langle P_2, \sigma, \alpha_2, \beta_2\rangle$*, if there exists a* 0*-bisimulation relation* R *such that* $\langle P_1, \sigma, \alpha_1, \beta_1\rangle \; R \; \langle P_2, \sigma, \alpha_2, \beta_2\rangle$.

(2) Programs P *and* Q *are* 0*-bisimilar, written as* $P \approx_0 Q$*, if* $\forall \sigma, \alpha_1, \alpha_2, \beta_1, \beta_2 \bullet \langle P, \sigma, \alpha_1, \beta_1\rangle \approx_0 \langle Q, \sigma, \alpha_2, \beta_2\rangle$. □

This definition indicates that $\approx_0$ is the largest relation for 0-bisimulation over configurations. Further, the concept of 0-bisimulation has also been extended to the domain of processes.

Now we give the definition of the *simple compensation sequence*:

(1) $\langle\ \rangle$ is a simple compensation sequence;
(2) $(C_1 : \alpha_1)\hat{}\ldots\hat{}(C_n : \alpha_n)$ is a simple compensation sequence if $\alpha_1, \ldots, \alpha_n$ are also simple compensation sequences.

Example 2. Let $\alpha = (C_1 : \alpha_1)\hat{}(C_2 : \alpha_2)\hat{}(C_3 : \langle\ \rangle)$,

$\alpha_1 = (C_4 : \alpha_4)\hat{}(C_5 : \langle\ \rangle)$ and $\alpha_2 = (C_6 : \alpha_6)\hat{}(C_7 : \langle\ \rangle)\hat{}(C_8 : \alpha_8)$,

$\alpha_4 = (C_9 : \langle\ \rangle)\hat{}(C_{10} : \langle\ \rangle)$,

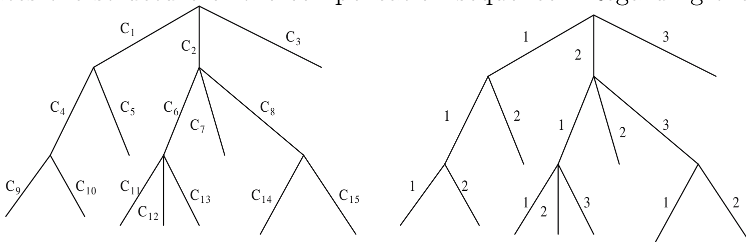
$\alpha_6 = (C_{11} : \langle\ \rangle)\hat{}(C_{12} : \langle\ \rangle)\hat{}(C_{13} : \langle\ \rangle)$ and $\alpha_8 = (C_{14} : \langle\ \rangle)\hat{}(C_{15} : \langle\ \rangle)$

From the above definition, we know α is a simple compensation sequence. □

Consider a simple compensation sequence $\alpha = (C_1 : \alpha_1)\hat{}\ldots\hat{}(C_n : \alpha_n)$. In order to describe its full structure, we translate the nested sequence structure of α into a tree structure; namely $tree(\alpha)$:

(1) if $\alpha = \langle\ \rangle$, then $tree(\alpha)$ is just one node;
(2) if $\alpha = (C_1 : \alpha_1)\hat{}\ldots\hat{}(C_n : \alpha_n)$, then there are n branches for the root of the tree, the names for the n edges from left to right are $C_1, \ldots\ldots, C_n$. Further, the root of $tree(\alpha_i)$ is just another node of edge C_i.

The tree structure of α in *Example* 2 is shown in the left tree below. It clearly illustrates the structure of the compensation sequence. Regarding the tree for



a simple compensation sequence, we now assign a number for each edge (called edge number). For a given edge, consider all the edges starting from the upper

point of the given edge. The edge number for a given edge is i if the given edge is the i-th edge starting from left to right. The edge number for each edge in *Example* 2 is shown in the right tree above.

The $path(\alpha)$ for any simple compensation sequence β is defined as:

$$path(\alpha) =_{df} \{ i_1{}^\frown \ldots {}^\frown i_n \mid \exists \text{ edge } C \bullet i_1, \ldots, i_n \text{ are the edge number from the root of } tree(\alpha) \text{ to the exact edge } C\}$$

The sequence $i_1{}^\frown \ldots {}^\frown i_n$ dynamically indices to the exact edge in $tree(\alpha)$. Therefore, we will use $\alpha[i_1{}^\frown \ldots {}^\frown i_n]$ to represent the corresponding edge in $tree(\alpha)$, which stands for a program. For example, in the simple compensation sequence α of *Example* 2, $2^\frown 1^\frown 2$ will identify program C_{12}.

Two sequences α_1 and α_2 are called structural equivalence, written as $\alpha_1 \approx_s \alpha_2$, if $path(\alpha_1) = path(\alpha_2)$.

However, not all compensation sequences are simple. For example, let $\alpha = ((C_1 : \langle\,\rangle) \parallel (C_2 : (C_3 : \langle\,\rangle)))^\frown(C_4 : \langle\,\rangle)$. It is easy to see that α is not simple.

To illustrate the further structure for compensation, we introduce a function $mul(\alpha)$, which contains all the simple compensation sequences for compensation sequence α:

$$mul(\langle\,\rangle) =_{df} \{\langle\,\rangle\}$$
$$mul((C_1 : \alpha_1)^\frown x) =_{df} \{(C_1 : u)^\frown t \mid u \in mul(\alpha_1) \wedge t \in mul(x)\}$$
$$mul((x \parallel y)^\frown z) =_{df} mul(x^\frown y^\frown z) \cup mul(y^\frown x^\frown z)$$

Now we introduce the concept of k-bisimulation ($k \geq 1$). Together with 0-bisimulation, they form the basis in defining program equivalence.

Definition 5. *(k-Bisimulation) A symmetric relation R is a k-Bisimulation ($k \geq 1$)* if and only if *for any $\langle P, \sigma, \alpha, \beta\rangle \, R \, \langle Q, \sigma, \alpha_1, \beta_1\rangle$*

(0) $Equiv(\alpha, \alpha_1, k-1)$;

(1) if $\langle P, \sigma, \alpha, \beta\rangle \stackrel{\surd}{\Longrightarrow} \langle P', \sigma', \alpha', \beta'\rangle$,

then $\exists Q', \alpha_1', \beta_1' \bullet \langle Q, \sigma, \alpha_1, \beta_1\rangle \stackrel{\surd}{\Longrightarrow} \langle Q', \sigma', \alpha_1', \beta_1'\rangle$ and $\langle P', \sigma', \alpha', \beta'\rangle \, R \, \langle Q', \sigma', \alpha_1', \beta_1'\rangle$ and $Equiv(\alpha', \alpha_1', k-1)$;

(2) if $\langle P, \sigma, \alpha, \beta\rangle \stackrel{\eta}{\Longrightarrow} \langle P', \sigma', \alpha', \beta'\rangle$ $(\eta \in \{\tau, a\})$,

(2-1) if $\sigma \neq \sigma'$, then

$\exists Q', \alpha_1', \beta_1' \bullet \langle Q, \sigma, \alpha_1, \beta_1\rangle \stackrel{\eta}{\Longrightarrow} \langle Q', \sigma', \alpha_1', \beta_1'\rangle$ and $\langle P', \sigma', \alpha', \beta'\rangle \, R \, \langle Q', \sigma', \alpha_1', \beta_1'\rangle$ and $Equiv(\alpha', \alpha_1', k-1)$;

(2-2) if $\sigma = \sigma'$, then

either $\langle P', \sigma', \alpha', \beta'\rangle \, R \, \langle Q, \sigma, \alpha_1, \beta_1\rangle$ and $Equiv(\alpha', \alpha_1, k-1)$;

or $\exists Q', \alpha_1', \beta_1' \bullet \langle Q, \sigma, \alpha_1, \beta_1\rangle \stackrel{\eta}{\Longrightarrow} \langle Q', \sigma', \alpha_1', \beta_1'\rangle$ and $\langle P', \sigma', \alpha', \beta'\rangle \, R \, \langle Q', \sigma', \alpha_1', \beta_1'\rangle$ and $Equiv(\alpha', \alpha_1', k-1)$;

(3) if $\langle P, \sigma, \alpha, \beta\rangle \xrightarrow{\eta} \langle \boxtimes, \sigma', \alpha', \langle\rangle\rangle \quad (\eta \in \{\tau, a\})$,
then $\exists \alpha'_1 \bullet \langle Q, \sigma, \alpha, \beta\rangle \xrightarrow{\eta} \langle \boxtimes, \sigma', \alpha'_1, \langle\rangle\rangle$ *and* $Equiv(\alpha', \alpha'_1, k-1)$;

where

(a) $Equiv(\alpha_1, \alpha_2, n) =_{df} \forall u \in mul(\alpha_1) \bullet \exists v \in mul(\alpha_2) \bullet equiv(u, v, n) \wedge \forall v \in mul(\alpha_2) \bullet \exists u \in mul(\alpha_1) \bullet equiv(v, u, n)$

(b) $equiv(u, v, n) =_{df} u \approx_s v \wedge \forall t \in path(u) \bullet u[t] \approx_n v[t]$ □

Here, $equiv(u, v, n)$ indicates that the two simple compensation sequences u and v are structural equivalent (described by $u \approx_s v$). Further, it also indicates that every program in $tree(u)$ is n-bisimilar to the corresponding program in $tree(v)$.

Regarding $Equiv(\alpha_1, \alpha_2, n)$, α_1 and α_2 may not be simple compensation sequences. We use $mul(\alpha_1)$ and $mul(\alpha_2)$ to record all the simple compensation sequences generated from α_1 and α_2 respectively. Further, for every simple compensation sequence u in $mul(\alpha_1)$, there should exist a simple compensation sequence v in $mul(\alpha_2)$ such that $equiv(u, v, n)$ is satisfied and vice-versa. Therefore, $Equiv(\alpha_1, \alpha_2, n)$ stands for the n-bisimilarity for α_1 and α_2.

The key point of k-bisimulation ($k \geq 1$) is as follows. As mentioned earlier, the third element of a configuration is a sequence recording a set of programs in tree structure. In k-bisimulation ($k \geq 1$), for the sequences appearing as the third elements in the two bisimilar configurations before and after the transition, their structures should be the same. Further, before a transition (or after a transition), the corresponding processes recorded in the two sequences of two k-bisimilar configurations should be $(k-1)$-bisimilar. This shows the difference of k-bisimulation and 0-bisimulation, which is shown in item (0) and the extra information (i.e., function $Equiv(\ )$) in other items in the definition of k-bisimulation.

Definition 6. *(1) Configurations* $\langle P_1, \sigma, \alpha_1, \beta_1\rangle$ *and* $\langle P_2, \sigma, \alpha_2, \beta_2\rangle$ *are k-bisimilar* ($k \geq 1$)*, written as* $\langle P_1, \sigma, \alpha_1, \beta_1\rangle \approx_k \langle P_2, \sigma, \alpha_2, \beta_2\rangle$*, if there exists a k-bisimulation relation R such that* $\langle P_1, \sigma, \alpha_1, \beta_1\rangle \; R \; \langle P_2, \sigma, \alpha_2, \beta_2\rangle$.

(2) Programs P and Q are k-bisimilar ($k \geq 1$)*, the fact is written as* $P \approx_k Q$*, if* $\forall \sigma, \alpha_1, \alpha_2, \beta_1, \beta_2 \bullet Equiv(\alpha_1, \alpha_2, k-1) \implies \langle P, \sigma, \alpha_1, \beta_1\rangle \approx_k \langle Q, \sigma, \alpha_2, \beta_2\rangle$ □

From definition 5 and 6, k-bisimulation relies on $(k-1)$-bisimulation. Thus, 0-bisimulation is the basis for the definition of all k-bisimulations (for $k \geq 1$). Therefore, n-bisimulation ($n \geq 0$) forms a *hierarchy* structure.

Lemma 1. If $P \approx_k Q$, then $P \approx_{k-1} Q$ ($k \geq 1$). □

Definition 7. *(Program equivalence)* $\approx =_{df} \bigcap_{n \geq 0} \approx_n$ □

Two programs are equivalent, if they are n-bisimilar for any n ($n \geq 0$).

Theorem 1. $\approx$ *is a congruence.* □

This theorem indicates that "program equivalence" relation $\approx$ is preserved by all *BPEL0* processes.

5 Related Work

In recent years, many efforts have been attempted to formalize various workflow languages [1, 3, 6, 5], especially with some kinds of *compensation* concepts, which root to the Sagas and open nested transactions, and have been studied for a long time in the transaction processing world.

M. Mazzara *et al.* suggested to merge the fault and compensation handling into a general framework of even handling [14], and presented an operational semantics for their CCS-like language. In paper [12], Koshkina *et al.* analyzed the link structure carefully in BPEL, and presented a language called BPEL-calculus (a CCS-like language as well) to model and verify BPEL specifications. But they omitted the compensation and fault handling mechanisms totally.

In a recent paper [7], Bruni *et al.* presented the operational semantics for a series of languages, embodying the concept of *compensation*. However, the compensation in these languages is basic-activity-oriented (each basic activity is in company with a compensation) with no name. The compensation is triggered by a special command, and always executed in the reverse order with respect to the installation. Compared to the work of paper [6], Butler *et al.* presented a language called StAC (Structured Activity Compensation), where the semantics of StAC was defined on its semantic language. The paper [8] illustrated the link and difference between the two languages proposed by Bruni [7] and Butler [6] respectively. Our previous work [16] studied the semantics of the fault and compensation handling in BPEL specification, and presented a simple language to catch the features of BPEL related to fault and compensation handling. The big step semantics are adopted by most researchers when studying the compensation mechanism in workflow language.

Some research groups aim to model and verify the BPEL4WS program, such as [10, 3]. In paper [10], authors presented a set of tools and techniques for analyzing interactions of composite web services which are specified in BPEL. The BPEL specifications are translated into an intermediate representation, and then verified using SPIN. In paper [17], we adopted a similar approach to use model checker UPPAAL [4] to verify the properties of BPEL program including timed properties. But we find no work on verifying BPEL specification with the features of the compensation and fault handling.

Of course there are much more informal work on workflow languages, and especially on BPEL. For example, [1] proposed a general framework to evaluate the capabilities and limitations of BPEL. Paper [2] presented an informal analysis from a pattern-based view on workflow language. But their work did not provide the patterns related to fault and compensation handling as well.

6 Conclusion

BPEL is one of the most important business process modelling languages, aimed to specify the business services which are formed by distributed, interoperational

and heterogeneous components over networks. One distinct feature of BPEL is its scope-based compensation handling and fully programmable compensation mechanism, which allows users to specify the compensation behaviors of processes in application-specific manners.

In this paper, we proposed a language *BPEL0* based on BPEL, and regard it as a foundation to study the scoped-base compensation languages. With the help of the key concepts of *compensation closure* and *compensation context*, the semantics of *BPEL0* has been carefully studied. Based on the semantics, the concept of bisimulation in hierarchy structure has been studied , which can be used to define the equivalence between *BPEL0* programs

Based on this work, an execution engine of *BPEL0* is being developed, and we also hope to study the verification of *BPEL0* relying on the semantic framework proposed here, which can be added into the developing of execution engine. As one future work as well, we will consider the design patterns provided by *BPEL0*, especially the patterns with compensation handling by means of our defined bisimulation relation.

References

1. W. Aalst, M. Dumas, and A. Hofstede, and P. Wohed, Analysis of web services composition languages: The case of BPEL4WS. In *Proc. of ER'03*, LNCS 2813, pp 200-215, Springer, 2003.
2. W. Aalst, A. Hofstede. YAWL: yet another workflow language. In *Inf. Syst.*, Vol.30(4), pp 245-275, 2005.
3. B. Benatallah and R. Hamadi. A Petri net-based model for web service composition. *Proc. of ADC'03*, pp 191-200, Australian Computer Society, 2003.
4. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Y. Wang. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III: Verification and Control*, pp 232-243, Springer, 1996.
5. A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web services choreographies. In *Pro. of WS-FM'04*, 2004.
6. M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Proc. of Coordination'04*, LNCS 2949, pp 87-104, Springer, 2004.
7. R. Bruni, H. Melgratti, and U. Montanari, Theoritical Foundations for Compensation in Flow Composition Languages, In *Proc. of ACM POPL'05*, 2005.
8. R. Bruni, M. Butler, C. Ferreira, C. A. R. Hoare, H. C. Melgratti, U. Montanari. Comparing Two Approaches to Compensable Flow Composition. In *Proc. of CONCUR'05*, pp 383-397, 2005.
9. BPEL4WS, *Business Process Execution Language for Web Service*. `http://www.siebel.com/bpel`, 2003.
10. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. of WWW'04*, pp 621-630, 2004.
11. H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD'87*, pp 249-259, ACM Press, 1987.
12. M. Koshkina and F. Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. In *ACM SIGSOFT Software Engineering Notes, 29(5)*, 2004.

13. F. Leymann. *WSFL: Web Serices Flow Languag.* http://www-3.ibm.com/software/solutions/webservices/pdf/WSDL.pdf.
14. M. Mazzara and R. Lucchi. A framework for generic error handling in business process. In *Proc. of WS-FM'04*, ENTCS Vol. 105, pp 133-145, Elsevier, 2004.
15. J. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. PhD thesis, Dept. of Electrical Eng. and Computer Sci., MIT, 1981.
16. Qiu Zongyan, Wang Shuling, Pu Geguang and Zhao Xiangpeng. Semantics of BPEL4WS-like Fault and Compensation Handling . In *Proc. of Formal Methods'05*, pp 350-365, Springer, 2005.
17. Pu Geguang, Zhao Xiangpeng, Wang Shuling, and Qiu Zongyan. Towards the semantics and verification of BPEL4WS. In *Proc. of WS-FM05*, 2005.
18. Pu Geguang, Zhu Huibiao, Qiu Zongyan, Wang Shuling, Zhao Xiangpeng, and He Jifeng. Theoretical Foundations of Scope-based Compensation Flow Language for Web Service. Research Report 67, School of Mathematical Sciences, Peking University, 2005.
19. S. Thatte. *XLANG: Web Service for Business Process Design.* http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.html.