Running BPEL Processes without Central Engines

Weihai Yu

Department of Computer Science
University of Tromsø, Norway
weihai@cs.uit.no

Abstract

WS-BPEL, or simply BPEL (Business Process Execution Language), is becoming a *de facto* standard for web services composition. It is natural to anticipate that the compositions are performed dynamically by a large number of end-users. However, the current process technology based on central process engines impedes the adoption of BPEL for this purpose. We propose an approach to execution of BPEL processes without central engines. The approach is of continuation-passing style, where continuations, or the reminder of executions, are passed along with messages for process execution. Two continuations are associated with an execution: a success continuation and a failure continuation. Recovery plans for processes are automatically generated at runtime and attached to failure continuations. *

1 Introduction

Service-oriented computing provides a programming paradigm for achieving Internet-scale interoperability. While the basic web service technology (XML, SOAP, WSDL) has reached a certain level of maturity, the next major step is to offer new services based on the composition of existing ones [16]. WS-BPEL [15], or simply BPEL, is becoming a *de facto* standard for services composition based on the workflow technology. Using BPEL, a composite service is a BPEL process that uses other services (processes) in some prescribed order.

However, the BPEL technology today can hardly be adopted by a large number of end-users for dynamically composing available services. One particular reason is that running a BPEL process requires a central BPEL engine. Firstly, there is no central administration domain between independent service providers and various service consumers. Secondly, the central engines are typically heavyweight and expensive to small end-users. Moreover, this centralized approach suffers from poor scalability to large number of concurrent processes and vulnerability to failures such as crashes of the engine and disconnections to it [1]. Techniques like replication [8] have been adopted to address the scalability and reliability issues. This, however, makes the central engine even more expensive and heavyweight, and thus less affordable by the large number of end-users.

With a decentralized execution, the service sites communicate directly with each other without the involvement of a central engine. Several decentralized approaches have been proposed. Common to most of these, a process is instantiated prior to its execution. During instantiation, proper resources and control are pre-allocated in the distributed

* *This paper was presented at the NIK-2007 conference. For more information, see //www.nik.no/.*

environment. These approaches inevitably allocate resources even for the parts that are not executed. Some process management tasks, such as fault handling and recovery, are dependent on runtime information and cannot be properly planned during instantiations. They also tend to have limited adaptability at runtime due to the complication of re-allocating the pre-allocated resources and control.

We propose a peer-to-peer approach that does not involve static process instantiation. Unnecessary pre-allocation of resources and control is avoided. The approach is of continuation-passing style, which is a common practice in the functional programming community. Basically, a continuation represents the rest of an execution at a certain point of the execution. It is automatically derived during the execution. By knowing the continuation of the current execution, the control can be passed to the proper processing entities without the involvement of a central engine.

An important task for reliable process execution is the support for recovery using compensation. To achieve this, two continuations are associated with any particular point of execution. The success continuation represents the path of execution towards the successful completion of the process. The failure continuation represents the path of execution towards the proper compensation of committed effects after certain failure events.

The rest of this paper is organized as follows. Section 2 describes the core BPEL process model and an example BPEL process. Section 3 and Section 4 present our key contribution: the abstract CEKK machine and its state transition rules. Section 5 illustrates how peer-to-peer process execution is conducted using CEKK state transition rules by walking through the example. Section 6 describes the process container that implements the CEKK machine. Section 7 discusses related work. Section 8 concludes with our contributions and some possible future work.

2 The Core BPEL Process Model

We present the core BPEL process model [15], keeping the essential elements just enough for the purpose of the presentation of our approach.

In BPEL, processes and (composite) services are synonymous. A process is an *activity*, which has a hierarchical structure as defined recursively below.

bpel_process ::= bpel_activity

bpel_activity ::= ⊥ | **`invoke(`**agent, op**`)`** | **`receive(`**op**`)`** | **`reply(`**op**`)`**

| **`sequence(`**bpel_activity*`)`**| **`flow(`**agent?, bpel_activity*`)`**

| **`scope(`**agent?, bpel_activity*, (fault_name, bpel_activity)*, bpel_activity?, bpel_activity`)`**

| **`throw(`**fault_name`)`**| **`compensate()`** | …

Basic activities include empty activities ⊥, activities for providing and invoking services. A *service* is provided through an invokable operation by an *agent* (or at a *site*) with **`receive`**.

A structured activity consists of a collection of activities, either all to be executed in some prescribed order, such as in sequence using **`sequence`** or in parallel using **`flow`**. Here, we extend the original BPEL **`flow`** activity with an optional agent (called a *join agent*) at which the parallel branches will join. If the join agent is not given (as with the original BPEL), a default one will be chosen.

Activities can run within a scope using **scope**, which provides a boundary for fault handling and recovery. A scope is managed by an agent. Here again, we extend the original BPEL scope with an optional scope agent. A scope can be associated with a number of activities known as event handlers, a number of fault handlers, an optional compensation handler, and a primary activity. Within a scope, a fault can be thrown using **throw** with a fault name. The fault will be caught by the scope and handled with a corresponding fault handler. A typical activity initiated by a fault handler is **compensate**, which executes the compensation operations currently installed within the scope.

Figure 1 shows a simple example process. A client at site *c* starts a process in a new scope. The primary activity of the process is an invocation of the *order* service at site *ordS*. The *order* service in turn invokes two services in parallel: *invoice* at site *invS* and *ship* at site *shpS*. When the two services are successfully done, the *order* service replies to the client and then terminates. The process also has an event handler. Upon event *cancel*, an *any* fault is thrown. When this fault is caught, a default fault handler is executed.

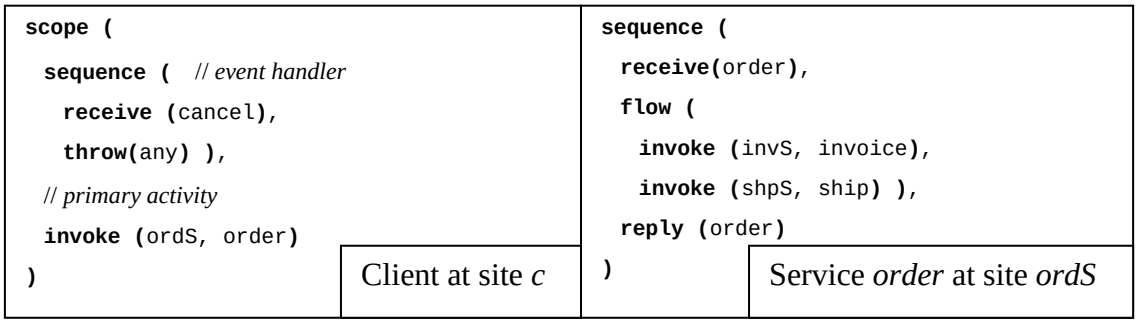

Figure 1. Example process

3 The CEKK Machine

Key to our approach is an abstract state machine called CEKK, which is built on CEK^T [9] and PCKS [11]. Execution of a BPEL process is represented as a sequence of the state transitions with the CEKK machine. This section describes the CEKK machine. The state transition rules are described in the next section.

A *global CEKK machine* defines the possible global states of a process and the possible transitions among them. It consists of a number of *local CEKK machines* that define possible states and their possible transitions locally at agents. Every active branch of the process has a corresponding local CEKK machine. The global state of the process is the aggregation of the local states. Except for two special operations (**stopall** and **compensateall**, to be described in the next section), the global state transitions are defined solely by the local state transitions. This important property assures that no global coordination among the agents is needed for most global state transitions.

A state of a local CEKK machine at agent *p* is a quadruple $<c, e, ks, kf>_p$, where *c* is called a control activity, *e* an environment, and *ks* and *kf* two continuations (thus the name CEKK with C for control, E for environment and K for continuation).

The control activity and the continuations together represent the work yet to be carried out.

A control activity *c* represents the next activity to be executed immediately. An *activity* is either a BPEL activity or an *auxiliary activity* automatically generated during execution. The use of the individual auxiliary activities will be explained in the next section when they appear in the corresponding state transitions.

activity ::= bpel_activity | auxiliary_activity

auxiliary_activity ::= **`join(`**agent, condition**`)`**

| **`eos(`**scope_id, agent, (fault_name, bpel_activity)*, bpel_activity**`)`**| **`eosf(`**scope_id, continuation**`)`**

| **`eoi(`**invocation_id, agent, op, bpel_activity, continuation**`)`**| **`eoif(`**invocation_id, agent, continuation**`)`**

A *continuation* is the reminder of execution after the control activity. *ks*, the *success continuation*, is applied when the execution of the control activity succeeds. *kf*, the *failure continuation*, is applied when the execution of the control activity fails. A continuation is represented as a stack of activities.

continuation ::= ⊥ | activity : continuation

For a continuation $k = a_n$: …a_1:a_0, we write k.head = a_n and k.tail = a_{n-1}: …a_1:a_0. An empty continuation is the same as an empty activity, ⊥. Hence k:⊥ = k. We may also write $k = k_1$:a:k_0 for a continuation containing an activity a between two partial continuations k_1 and k_0. When a continuation k is *applied*, k.head, i.e. the activity at the front, becomes the control activity of the new state.

An environment *e* is the runtime context of the process. Information contained in *e* includes bindings for process-relevant data, scopes and activity status of the current execution so far.

4 Peer-to-Peer Process Execution with CEKK

In the course of a BPEL process execution, a local CEKK state is represented with a message. Conducting the execution of processes is the sequences of sending and interpreting messages according to the CEKK state transitions rules. Before the individual state transition rules are presented, it is useful to note that the state transitions appear in one of the following four forms:

1. *Local ongoing* — a state transition within a local CEKK machine is performed locally at agent *p*:

   $\langle c_0, e_0, ks_0, kf_0 \rangle_p \rightarrow \langle c_1, e_1, ks_1, kf_1 \rangle_p$

2. *Remote forwarding* — a state of a local CEKK machine at agent *p* is passed to a state of another local CEKK machine at agent *q*:

   $\langle c, e, ks, kf \rangle_p \rightarrow \langle c, e, ks, kf \rangle_q$

   In other words, the local CEKK machine at *p* terminates and a new local CEKK machine starts at *q* with the same state. In terms of process execution, this corresponds to a message <*c*, *e*, *ks*, *kf*> from *p* to *q*.

3. *Local divergence* — multiple parallel branches are spawned at agent *p*:

   $\langle c_0, e_0, ks_0, kf_0 \rangle_p \rightarrow$

   $\{\langle c_1, e_1, ks_1, kf_1 \rangle_p, \langle c_2, e_2, ks_2, kf_2 \rangle_p , \ldots, \langle c_n, e_n, ks_n, kf_n \rangle_p \}$

   That is, a single local CEKK machine turns now into multiple local CEKK machines at agent *p*.

4. *Local convergence* — multiple parallel branches are joined into one at agent *p*:

$$\{<c_1, e_1, ks_1, kf_1>_p, <c_2, e_2, ks_2, kf_2>_p , \ldots, <c_n, e_n, ks_n, kf_n>_p \} \rightarrow$$
$$<c_u, e_u, ks_u, kf_u>_p$$

That is, multiple local CEKK machines are converged into one at agent *p*.

Notice that remote forwarding is the only case of message sending, which is asynchronous and direct between agents. In all other cases, state transitions are carried out locally at individual agents. This explains why global coordination is not needed among the agents.

One might wonder if there is a need for convergence of multiple parallel branches at different sites. In fact, this can be reduced to a number of remote forwardings and a local convergence. In presenting the state transition rules, we ignore such obvious remote forwardings.

The CEKK state transition rules are summarized in Figure 2.

To make the presentation more readable, we focus on the use of continuations for conducting the process control flow and omit the tedious details of the environments. Thus we only present the triple $<c, ks, kf>_p$, instead of the entire CEKK quadruple. We show here the transition rules in normal executions toward the success end, except for some auxiliary activities (such as **`eosf`** and **`eoif`**) that are only encountered in the course of compensations. During compensations, the state transitions rules may be somewhat different from normal executions in how the failure continuations are updated, depending on how the failure of compensations are handled (retried, ignored etc.).

Rule SC for scope is applied at the entrance of a scope, which spawns several parallel branches, one for the primary activity and one for each event handler. If the scope agent is not explicitly given, the current agent is chosen as the scope agent. Otherwise, if the explicitly given scope agent is not the current agent, the state is first forwarded to the scope agent before rule SC is applied. In the new CEKK state of the primary activity, the control activity is the primary activity of the scope and two auxiliary activities **`eos`** and **`eosf`** (end-of-scope) are pushed into the success and failure continuations. They mark the boundary of the scope with a scope identifier *sid* and encapsulate sufficient information for the proper normal and abnormal termination of the scope.

Rule ES for end-of-scope is applied for the normal termination of a scope. All active branches of the scope (such as installed event handlers of the scope) are stopped with a special operation **`stopall`** and the failure continuation is properly updated. During the execution within the scope (i.e., before the **`eos`** auxiliary activity becomes the control activity), the compensation activities of the executed activities are automatically generated and attached to the kf_1 part of the failure continuation (i.e., after the corresponding **`eosf`**). They would have been applied if the scope must be aborted and the partially committed effects be compensated for before its normal termination. Now during normal termination of the scope, kf_1 is replaced by the higher level compensation handler *ch* of the scope (if *ch* is explicitly provided).

The auxiliary activity **`eosf`** becomes a control activity only in the course of compensation. With Rule SF (scope failure), the compensation continues beyond the current scope (here the “retry” strategy is adopted to deal with compensation failures). If the entire compensation process was initiated from this current scope (where Rule CP was applied), the compensation terminates and the execution continues by applying the success continuation (this case is not shown in the figure).

A request-reply invocation involves a remote forwarding of the invocation message and then a local convergence (rule RR for request-reply invocation) at the service provider. An invocation is executed when an **`invoke`** activity matches a corresponding **`receive`** activity. If no such pair exists upon an invocation (service not installed), a system runtime exception is thrown. The body of the invoked services is represented in the ks_1^p part (i.e., prior to the **`reply`** activity) of the success continuation of the service provider. Two auxiliary activities **`eoi`** and **`eoif`** (end-of-invocation) are attached to the success and failure continuations. They mark the boundary of the invocation with an invocation identifier *iid* and encapsulate sufficient information for proper termination of the invocation. Note that **`eoi`** replaces the **`reply`** activity and encapsulates the information contained in the **`invoke`** message.

$\langle \mathbf{scope(}q, eh_1 \ldots eh_n, fhs, ch, a\mathbf{)}, ks, kf \rangle_p \rightarrow_{(SC)}$ (SC)

$\{\langle a, \mathbf{eos(}sid, q, fhs, ch\mathbf{)}{:}ks, \mathbf{eosf(}sid\mathbf{)}{:}kf \rangle_q, \langle eh_1, \perp, \perp \rangle_q, \ldots, \langle eh_n, \perp, \perp \rangle_q \}$

$\langle \mathbf{eos(}sid, q, fhs, ch\mathbf{)}, ks, kf_1{:}\mathbf{eosf(}sid\mathbf{)}{:}kf_2 \rangle_p \rightarrow_{(ES)} \langle \mathbf{stopall}(sid), ks, ch{:}kf_2 \rangle_q$ (ES)

$\langle \mathbf{eosf(}sid, kf_0\mathbf{)}, ks, kf \rangle_p \rightarrow_{(SF)} \langle kf_0.head, kf_0.tail, kf_0{:}kf \rangle_p$ (SF)

$\{ \langle \mathbf{invoke(}p, op\mathbf{)}, ks^c, kf^c \rangle_c, \langle \mathbf{receive(}op\mathbf{)}, ks_1^p{:}\mathbf{reply(}op\mathbf{)}{:}ks_2^p, kf^p \rangle_p \} \rightarrow_{(RR)}$ (RR)

$\langle ks_1^p.head, ks_1^p.tail{:}\mathbf{eoi(}iid, c, op, ks^c\mathbf{)}{:}\ ks_2^p, \mathbf{eoif(}iid, c, kf^c\mathbf{)}{:}kf^p \rangle_p$

$\langle \mathbf{eoi(}iid, c, op, ks^c\mathbf{)}, ks, kf_1{:}\mathbf{eoif(}iid, c, kf^c\mathbf{)}{:}kf_2 \rangle_p \rightarrow_{(EI)}$ (EI)

$\{ \langle \mathbf{receive(}op^{-1}\mathbf{)}, kf_1{:}\mathbf{reply(}op^{-1}\mathbf{)}, \perp \rangle_p,$

$\langle ks^c.head, ks^c.tail, \mathbf{invoke(}p, op^{-1}\mathbf{)}{:}kf^c \rangle_c,$

$\langle ks.head, ks.tail, kf_2 \rangle_p \}$

$\langle \mathbf{eoif(}iid, c, kf^c\mathbf{)}, ks, kf \rangle_p \rightarrow_{(RF)} \langle kf^c.head, kf^c.tail, kf^c \rangle_c$ (RF)

$\langle \mathbf{sequence(}a_1, \ldots, a_n\mathbf{)}, ks, kf \rangle_p \rightarrow_{(SQ)} \langle a_1, a_2{:}\ldots{:}a_n{:}ks, kf \rangle_p$ (SQ)

$\langle \mathbf{flow(}ja, a_1, \ldots, a_n\mathbf{)}, ks, kf \rangle_p \rightarrow_{(FL)}$ (FL)

$\{ \langle a_1, \mathbf{join(}ja, all\_doen\mathbf{)}{:}ks, \mathbf{join(}bw\_ja, all\_undone\mathbf{)}{:}kf \rangle_p, \ldots,$

$\langle a_n, \mathbf{join(}ja, all\_done\mathbf{)}{:}ks, \mathbf{join(}bw\_ja, all\_undone\mathbf{)}{:}kf \rangle_p \}$

$\{ \langle \mathbf{join(}p, all\_done\mathbf{)}, ks, kf_1{:}\mathbf{join(}bw\_ja, all\_undone\mathbf{)}{:}kf \rangle_p, \ldots,$

$\langle \mathbf{join(}p, all\_done\mathbf{)}, ks, kf_n{:}\mathbf{join(}bw\_ja, all\_undone\mathbf{)}{:}kf \rangle_p \} \rightarrow_{(JN)}$ (JN)

$\langle ks.\mathrm{head}, ks.\mathrm{tail}, \mathbf{flow(}bw\_ja, kf_1, \ldots, kf_n\mathbf{)}{:}kf \rangle_p$

$\langle \mathbf{throw(}fn\mathbf{)}, ks_1{:}\mathbf{eos(}sid, q, fhs, ch\mathbf{)}{:}ks_0, kf_1{:}\mathbf{eosf(}sid\mathbf{)}{:}kf_0 \rangle_p \rightarrow_{(TW)}$ (TW)

$\langle fhs[fn], \mathbf{stopall(}sid\mathbf{)}{:}\ ks_0, kf_0 \rangle_q$

$\langle \mathbf{compensate()}, ks_1{:}\mathbf{eos(}sid, q, fhs, ch\mathbf{)}{:}ks_0, kf_1{:}\mathbf{eosf(}sid\mathbf{)}{:}kf_0 \rangle_p \rightarrow_{(CP)}$ (CP)

$\langle kf_1.head, kf_1.tail{:}ks_0, kf_1{:}kf_0 \rangle_q$

Figure 2. CEKK state transition rules

Rule EI for end-of-invocation is applied for the normal termination of an invocation when the body of an invoked service is successfully executed. This is a local divergence that spawns three parallel branches:

1. Installation of the compensation activity op^{-1} of the successfully committed service.
2. Resumption of the execution at the services invoker. The rest of activities at the service invoker will be resumed by applying the success continuation ks^c contained

in the original invoking message. Moreover, the invocation to the corresponding installed compensation activity op^{-1} is pushed into the failure continuation. This state transition adheres to the original BPEL semantics and consists of two steps: a remote forwarding of a reply message to the invoker *c* and the application of the success continuation. An alternative is asynchronous messaging (not shown in the figure): if the first activity in ks^c is to be executed at a site other than *c*, the reply message can be forwarded directly to the site of that activity, rather than back to the invoker *c*.

3. Execution of the remaining activities at the service provider beyond the service body.

The **`eoif`** auxiliary activity becomes a control activity only in the course of compensation. Applying Rule RF (request failure), the compensation will carry on at the invoker, for the activities prior to the invocation (now in kf^c).

Rule SQ for sequences states that the first activity in a **`sequence`** is executed first, and the rest of the activities are pushed to the success continuation. They will be executed after the successful execution of the first one.

A **`flow`** activity spawns multiple parallel branches (Rule FL). Upon creation, all branches have the same success and failure continuations with the corresponding auxiliary **`join`** activities. The parallel branches, when successfully executed, will join at join agent *ja*. The success of the join is defined by the join condition *all_done*, which states that all branches are completed successfully. If the execution of some branch fails, all the branches will be compensated for and then join at the join agent *bw_ja* (backward join agent). The join condition *all_undone* states that all braches either fail (and their effects aborted) or their committed effects are successfully compensated for. The join conditions can be evaluated in the environments in the current CEKK states (the environments are not shown in the rules). If not explicitly provided, the join agents can be selected in different ways. Possible candidates are the site initiating the **`flow`** activity, the immediate enclosing scope agent, etc.

Rule JN is applied when the **`join`** activities of all parallel branches reach the join agent. If the join condition is evaluated to be true, the success continuation will be applied; otherwise, the join agent waits for other branches to be joined. The new failure continuation includes the compensation of the successfully executed brunches in parallel.

Rule TW for throw is applied when a fault (with fault name *fn*) is thrown. All active branches within the scope are stopped with the special operation **`stopall`** before the corresponding fault handler *fhs*[*fn*] is executed.

To compensate for the successfully committed effects within a scope, the corresponding failure continuation is applied (Rule CP). The continuation ks_0 outside the current scope will be applied after the compensation.

The two special operations **`stopall`** and **`compensateall`** deserves some more explanations. **`stopall`** stops all activities within the scope, such as active branches and event handlers. The installed compensation handlers are only stopped if the current scope is a top-level process. **`compensateall`** starts **`compensate`** of all active branches. Essentially, the two special operations need to know the current agents of all active parallel branches. Details of keeping track of the current agents will be presented in Section 6.

5 Running the Example

We walk now step by step through the example process of Figure 1 to illustrate how the transition rules are applied for the process execution. For better readability, when it is clear from the context, we omit some details of certain constructs. For example, we may use **sequence**, **sequence(receive**, **..)** or **sequence(receive(***order***)** , **..)** to indicate the same sequence activity.

A process or service is started with an initial CEKK state where the control activity is the activity defining the entire process and the success and failure continuations are empty. In our example, the process and the specified service (or the sub-process) are started in active branches at sites *c* and *ordS*.

<**scope(sequence(receive(***cancel***), ..)**, **invoke(***order***))**, ⊥, ⊥>$_c$ [c0]

<**sequence(receive(***order***)**, **..)**, ⊥, ⊥>$_{ordS}$ [ordS0]

Then, the CEKK state transition rules are applied according to the control activities of the current CEKK states.

[c0] →$_{(SC)}$ { <**invoke(***order***)**, **eos**, **eosf**>$_c$ [c1]

<**sequence(receive(***cancel***)**, **throw(***any***))**, ⊥, ⊥>$_c$ } [evt0]

During the scope instantiation (Rule SC), the primary activity of the scope and the event handler are initiated ([c1] and [evt0] respectively). The auxiliary activities **eos** and **eosf** are pushed to the success and failure continuations in the CEKK state of the primary activity. They will be used later to properly terminate the scope either in normal execution or when a fault is caught.

[ordS0]→$_{(SQ)}$<**receive(***order***)**, **flow**:**reply(***order***)**, ⊥>$_{ordS}$ [ordS1]

[evt0]→$_{(SQ)}$ <**receive(***cancel***)**, **throw(***any***)**, ⊥>$_c$ [evt1]

Applying Rule SQ, the first activity in a **sequence** becomes the new control activity and the rest of the activities are pushed to the success continuation. Now the service and the event handler are in the states ([ordS1] and [evt1]) ready to receive invocations.

{[c1], [ordS1]} →$_{(RR)}$ <**flow**, **eoi(***ord_iid*, *c*, *order*, **eos)**, **eoif(***ord_iid*, *c*, **eosf)**>$_{ordS}$ [ordS2]

A request-reply invocation is executed with a matching **invoke-receive** pair (Rule RR). The auxiliary activities **eoi** and **eoif** are attached to the continuations marking the end of the invocation.

[ordS2]→$_{(FL)}${ <**invoke(***invoice***)**, *ksjoin*, *kfjoin* >$_{ordS}$, [ordS3]

<**invoke(***ship***)**, *ksjoin*, *kfjoin* >$_{ordS}$ } [ordS4]

where *ksjoin* =**join(***ordS*, *all_done***)**:**eoi** and *kfjoin* =**join(***ordS*, *all_undone***)**:**eoif**

The **flow** activity spawns two parallel branches, which, after their successful executions, will join at *ordS* (Rule FL). A join is done by the auxiliary activity **join**. Here a **join** activity is pushed to both the success and failure continuations.

Suppose the invocations of the services *invoice* and *ship* are successfully replied to *ordS*. Then

{[ordS3], [ordS4]}→ $_{(RR)..(RP)}$

{ <**join(***ordS*, *all_done***)**, **eoi(***ord_iid*, *c*, *order*, **eos)**,**invoke(***invoice*$^{-1}$**)**: *kfjoin* >$_{ordS}$, [ordS5]

<**join(***ordS*, *all_done***)**, **eoi(***ord_iid*, *c*, *order*, **eos)**,**invoke(***ship*$^{-1}$**)**: *kfjoin* >$_{ordS}$ } [ordS6]

{[ordS5], [ordS6]} →$_{(JN)}$

<**eoi(***ord_iid, c, order,* **eos)**, ⊥, **flow(***ordS,* **invoke(***invoice*$^{-1}$**),invoke(***ship*$^{-1}$**))**:**eoif**>$_{ordS}$[ordS7]

The join is successful when both branches have reached the join agent *ordS* and the join condition *all_done* is evaluated to be true in the current environment (Rule JN). The new failure continuation now includes a **flow** activity that compensates in parallel for the corresponding successful branches.

[ordS7] →$_{(EI)}$

{ <**receive(***order*$^{-1}$**)**,

flow(*ordS,* **invoke(***invoice*$^{-1}$**),invoke(***ship*$^{-1}$**))**:**reply(***order*$^{-1}$**)**, ⊥>$_{ordS}$ [ordC0]

<**eos(***sid***)**, ⊥, **invoke(***ordS, order*$^{-1}$**)**:**eosf(***sid***)**>$_{c}$, [c2]

<⊥, ⊥, ⊥>$_{ordS}$ } [ordS9]

With Rule EI, the compensation activities are installed ([ordC0]), the invoking process resumes its control ([c2]) and the replying processes continue with the remaining activity ([ordS9]). A service is considered to have terminated when the control activity is an empty activity, such as [ordS9].

[c2] →$_{(ES)}$ <**stopall(***sid***)**, ⊥, ⊥>$_{ordS}$

By terminating the scope (Rule ES), the special operation **stopall** stops the installed event handler [evt1]. Since the scope of this example is a top-level process, the installed compensation handlers are also stopped. If, otherwise, the scope is not a top-level process and a compensation handler is explicitly given, invocation to that compensation handler will be attached to the new failure continuation.

To see how compensation works, suppose a fault (for example, thrown by the event handler [evt1]) is caught when the process is in the state [c2]. Applying Rule TW, the corresponding fault handler is executed (if not explicitly given, a default fault handler runs **compensate** and re-throw the fault to the outer scope), and all running activities are stopped by **stopall**. **compensate** applies the failure continuation (Rule CP).

6 Process Container

Process containers implement the runtime agents for process execution using the CEKK state transition rules. The structure of a process container is shown in Figure 3.

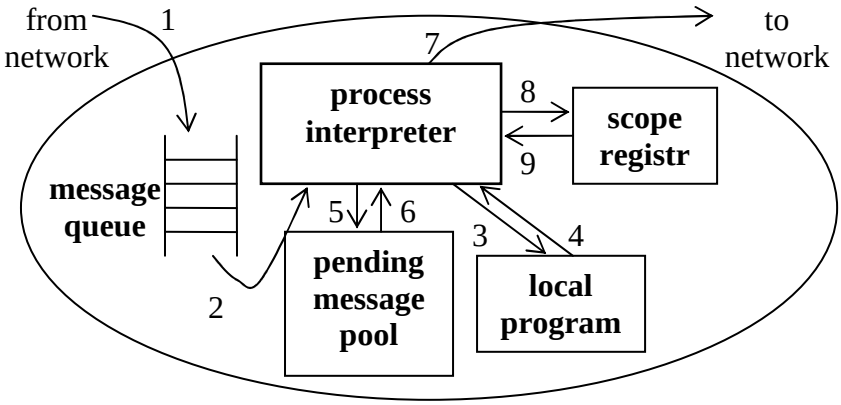

Figure 3. Structure of process container

A CEKK message from a remote site, such as an invocation request, a reply, a join or a scope instantiation, is first put in the message queue (1). A process interpreter is a pool

of threads that process the messages in the message queue. A thread dequeues a message from the message queue (2) and decides the next action according to the control activity part of the message. If the next state transition is a local convergence and some dependent message is not available yet, such as in the case of a **`receive`**, a **`join`**, etc., the dequeued message is put in the pending message pool (3). This message will be used later (4) when the dependent message is available (2 again). Before making a state transition, the thread may invoke some local procedures, such as handling the order of shipping (5, 6). After the local execution, new messages are either put in the pending message pool (3), such as receive, or sent to a remote agent (7), such as an invocation request or a reply message.

If the process container is also a scope agent managing a scope instance, it maintains the scope state in the scope registry (8, 9) in order to run the special operations **`stopall`** and **`compensateall`**. Basically, the scope state maintains the current locations of all active parallel branches. The location of a branch changes when a message is sent to a remote agent. To keep this location state up to date, when an agent sends a message to a remote agent (7), it also notifies the scope agent of the immediate enclosing scope (the scope agent can be obtained from the **`eos`** in the success continuation). To execute **`stopall`**, the scope agent asks the agents of all active branches to stop the corresponding local activities. To execute **`compensateall`**, all these agents run **`compensate`**.

7 Related Work

We classify decentralized approaches into two groups: static distribution and just-in-time distribution.

With static distribution, the process specification is analyzed and the process instantiated before execution (for example, [2][4][6][7][12][13][17][18]). During an instantiation, the resources and control are allocated in the distributed environment based on the static analysis. As a common problem to these approaches, resources are allocated even for the parts that are actually not executed (such as some of the alternative paths or when a process rolls back at an early stage). Because the distribution is based on the static structure of the process, there lacks a general mechanism for the support of features that require dynamic runtime information, such as fault handling and recovery. Furthermore, they tend to have limited adaptability at run time, because the control is mostly already in place before the execution started.

With just-in-time distribution, the information about the control of execution is carried along with the messages at runtime. Notice that in our approach an invocation is enacted with a matching **`invoke`**-**`receive`** pair, which can be materialized when they are just about to be used. In [14], part of the static specification of the process, represented as mobile code, is sent from agent to agent for further execution. This inherently impedes the kinds of processing that depend on runtime information, such as recovery. INCA [3] is a rule-based system that has some properties similar to our approach. An information carrier (INCA), which is sent from agents to agents, contains a log of the execution so far and rules for further execution. Thus the rules and the log play the role of success and failure continuations of our approach. Besides the principle difference between the approaches (rule-based versus continuation-passing), there are some subtle differences in what can be achieved. With INCA, process execution is conducted using both the rules carried in messages and pre-installed local rules. Thus INCA cannot achieve the degree of just-in-time distribution as our approach.

INCA is the only work in the just-in-time distribution group that supports automatic recovery. Automatic recovery is based on the log contained in the INCA and per-step rules (such as "if $step_i$ aborts, execute $step^{-1}_{i-1}$"). It is not obvious if more complicated rules can be generated (such as "if this is a compensation step of an alternative path within a parallel branch"). With our approach, the automatically generated recovery plan works for all such complicated cases.

Our CEKK machine is built on CEK^T [9] and PCKS [11]. The CEK^T machine supports asynchronous execution of distributed programs. Upon invocation of a remote procedure, a continuation is passed to the agent, who, after executing the procedure, applies the continuation instead of returning the control back to the caller. In [9], however, only one (distributed) thread of control is supported. The PCKS machine supports parallel executions of functional programs in a shared-memory environment.

The CEKK machine was first introduced in [19]. This current paper deals particularly with BPEL processes, which have more sophisticated features like scope management.

8 Conclusion

Our contribution is a new just-in-time distribution approach to peer-to-peer execution of BPEL processes. Without the reliance on a central engine, available services can now be dynamically composed into new services as BPEL processes. It also addresses the scalability and reliability constraints of the centralized approach, because there is no central performance bottleneck and point of failure. It does not unnecessarily pre-allocate resources as in static distribution approaches. The approach is of continuation-passing style. That is, the continuations, or the reminder of executions, are passed along in messages as part of the control information. This makes the conduction of process control flow as local operations rather than global coordination. Furthermore, our approach allows for automatic process recovery by automatically generating recovery plans into failure continuations.

Although it is widely believed that decentralized approaches are more scalable and reliable, a detailed performance study is needed to verify this. The performance study should also include comparisons between static and just-in-time distribution approaches. Typically, the choice of an appropriate mechanism would be dependent on runtime context such as workload and QoS requirements. This calls for approaches that are dynamically adaptable and re-configurable. We'd like to investigate the degree of adaptation that can be achieved by making continuations as first-class constructs.

9 References

[1] Alonso, G., A. Agrawal, A. El Abbadi, C. Mohan, "Functionality and Limitations of Current Workflow Management Systems", *IEEE Expert 12(5)*, 1997.

[2] Alonso, G., C. Mohan, R. Guenthoer, D. Agrawal, A. El Abbadi, M. Kamath, "Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management", *Proc. IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations*, Trondheim, August 1995.

[3] Barbara, D., S. Mehrotra and M. Rusinkiewicz, "INCAs: Managing Dynamic Workflows in Distributed Environments", *Journal of Database Management, Special Issues on Multidadatabases, 7(1),* 1996.

[4] Benatallah, B., M. Dumas and Q. Z. Sheng, "Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services", *Distributed and Parallel Databases*, 17(1), pp 5-37, 2005.
[5] Bernstein, P. A. and E. Newcomer, *Principles of Transaction Processing for Systems Professional*, Morgan Kaufmann, 1996.
[6] Fakas, G. J. and B. Karakostas, "A peer to peer (P2P) architecture for dynamic workflow management", *Information and Software Technology*, 46(6), pp 423-431, 2004.
[7] Gokkoca, E., M. Altinel, I. Cingil, N. Tatbul, P. Koksal, and A. Dogac, "Design and Implementation of a Distributed Workflow Enactment Service", *2nd IFCIS International Conference on Cooperative Information Systems (CoopIS 97)*, pp 89-98, June, 1997.
[8] Heinis, T., C. Pautasso and G. Alonso, "Design and Evaluation of an Autonomic Workflow Engine", *2nd International Conference on Autonomic Computing (ICAC 05)*, pp 27-38, June, 2005.
[9] Jagannathan, S., "Communication-Passing Style for Coordination Languages", *2nd International Conference on Coordination Models and Languages*, LNCS 1282, September 1997.
[10] Leymann, F. and D. Roller, "Building A Robust Workflow Management System With Persistent Queues and Stored Procedures", *International Conference on Data Engineering (ICDE 1998)*, pp 254-258, February 23-27, 1998.
[11] Moreau, J., "The PCKS-machine: An Abstract Machine for Sound Evaluation of Parallel Functional Programs with First-Class Continuations". *European Symposium on Programming (ESOP'94)*, LNCS 788, April 1994.
[12] Muth, P., D. Wodtke, J. Weißenfels, A. K. Dittrich and G. Weikum, "From Centralized Workflow Specification to Distributed Workflow Execution", *Journal of Intelligent Information Systems*, 10(2), pp 159-184, 1998.
[13] Nanda, M. G., S. Chandra and V. Sarkar, "Decentralizing Execution of Composite Web Services", *19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pp 170-187, October, 2004.
[14] Schneider, J., B. Linnert and L.-O. Burchard, "Distributed Workflow Management for Large-Scale Grid Environments", *Symposium on Applications and the Internet (SAINT 06)*, January, 2006.
[15] WS-BPEL, *Web Services Business Process Execution Language Version 2.0*, public review draft, QISIS Open http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf, August 23, 2006.
[16] Weerawarana, S. et al, *Web Services Platform Architecture*, Pearson Education Inc., 2005.
[17] Yan, J., Y. Yang, and G. K. Raikundalia, "SwinDeW – A p2p-Based Decentralized Workflow Management System", *IEEE Transactions on Systems, Man, and Cybernetics*, 36(5), pp 922-934, September, 2006.
[18] Yildiz, U. and C. Godart, "Towards Decentralized Service Orchestrations", *22nd Annual ACM Symposium on Applied Computing (SAC 2007)*, pp 1662-1666, March, 2007.
[19] Yu, W. and J. Yang, "Continuation-Passing Enactment of Distributed Recoverable Workflows", *22nd Annual ACM Symposium on Applied Computing (SAC 2007)*, pp 475-481, March, 2007.