Continuation-Passing Enactment of Distributed Recoverable Workflows

Weihai Yu and Jie Yang

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

Abstract. Scalability, reliability and adaptability are among the key requirements for the enactment of distributed workflows. In addition, system resources should be efficiently utilized. Central workflow engines and static analysis of workflow specifications are some of the important obstacles to meeting these requirements. We propose a fully decentralized approach to workflow enactment that is not subject to these obstacles. In addition, it supports automatic recovery. The approach is of continuation-passing style, where continuations, or the reminder of the executions, are passed along with asynchronous messages for workflow enactment. Two continuations are associated to an execution: a success continuation and a failure continuation. Recovery plans for workflows are automatically generated at runtime and included in failure continuations. A prototype is implemented.

1 Introduction

The workflow technology is now increasingly applied to areas beyond traditional business process automation. Examples include general software construction [22], enterprise-wise and inter-enterprise application integration [20], grid computing [28], e-commerce [30], and Web service composition [4][7][8]. Basically, a workflow, corresponding traditionally to a business process, consists of a number of loosely dependent activities and the control flows among them. Workflows, therefore, constitute the control concern of applications and their integrations. Applications constructed with the separation of their control concern from other concerns are more amenable to fast development and changes [22].

For serious applications, workflows should be recoverable in the sense of logic atomicity [19]. Every activity in a workflow is an atomic unit of execution whose effect is immediately committed after successful execution. If necessary, for example when the execution of some subsequent activity fails, the committed effect must be logically undone by a compensation activity.

Workflow enactment is the process of controlling the correct and reliable execution of activities by different processing entities. Traditionally, workflow enactment is

carried out by a central server known as the workflow engine. For example, if a workflow *W* consists of activity *A* at site *a* followed by activity *B* at site *b*, the workflow engine at site *w* invokes *A*, waits for the result of *A* from *a* and then invokes *B*. This central workflow engine can become a potential processing and communication bottleneck as well as a central point of failure. This centralized approach thus suffers from poor scalability to large number of concurrent workflows and vulnerability to failures either as server crashes or disconnections to it [1]. In addition, in some new distributed computing areas, such as dynamic Web services composition, there even hardly exists any central workflow engine.

With decentralized workflow enactment, the processing entities may communicate directly with each other (e.g. from *a* to *b*) to transfer data and control when necessary (e.g., after *A* finishes) in an asynchronous manner (e.g., without a return message from *b* to *a*). Several approaches to decentralized workflow enactment have been proposed. Common to most of these, a workflow specification is analyzed before execution, and proper resources and control are pre-allocated in the distributed environment. These approaches inevitably allocate resources even for the part of the workflow that is not executed. They also tend to have limited adaptability at runtime.

We propose an approach that is fully decentralized and does not involve static analysis of workflow specifications. There is no central point of performance bottleneck and failure. Unnecessary pre-allocation of resources is avoided. Furthermore, the approach is inherently more suitable for dynamic composition and adaptable execution of workflows. The approach is of continuation-passing style, which is common practice in the functional programming community. Basically, a continuation represents the rest of an execution at a certain point of the execution. They are 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 workflow engine.

In addition, our approach also supports automatic recovery of workflows. To achieve this, two continuations are associated to any particular point of execution. The success continuation represents the path of execution towards the success completion of the workflow. The failure continuation represents the path of execution towards the proper compensation of committed activities after certain failure events.

The rest of this paper is organized as follows. Section 2 describes the core workflow model used to explain the principle of our approach. Section 3 presents the abstract CEKK machine that represents states and state transitions for workflow enactment. Section 4 presents the CEKK rules for decentralized and recoverable enactment of workflows. The approach is further explained using an example in Section 5. The implementation of a prototype is briefly described in Section 6. Section 7 is a comprehensive account of related work. Section 8 consists of a conclusion of our contributions and directions for future work.

2 The Core Flow Model

Only a core model is presented here, since our goal is not a new model but rather a new enactment approach. A full-fledged model (such as one that includes a complete

list of workflow patterns [31]) is out of the scope of this paper. We choose to use the notation that is suitable for describing our approach throughout the paper.

The key abstraction in our model is *flow*. A flow corresponds to a workflow (business process or sub-process) commonly defined in the various traditional business process models. A flow has a hierarchical structure that is defined recursively below:

Flow ::= Empty Flow | Activity | Blackbox Flow

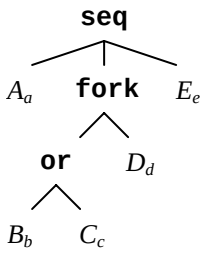
| **seq(**Flow*__)__| **fork(**join-agent**,** Flow*__)__ | **or(**Flow*__)__

| **if(**Condition, Flow, Flow**)** | **loop(**Condition, Flow*__)__

Figure 1 shows an example of a flow and its tree structure. The flow at the root of a flow tree is a top-level flow. All other flows in the tree are sub-flows. The leaves of the tree are primitive constructs.

seq(A_a**,** **fork(**e**,** **or(**B_b**,** C_c**)**, D_d**)**, E_e**)**

A: reservation at training course AdBeans
B/*C*: booking at hotel BedBreakfast/Continental
D: booking of airline ticket
E: approval by manager



a. The flow specification **b.** The tree structure

Figure 1. An example flow

At the primitive level, a flow can be an empty flow, an activity or a blackbox flow. An *empty flow* means there is nothing to do. It can be used, for example, to indicate the completion of the execution of a flow. We use $\perp$ to denote an empty flow.

An *activity* can be either manual or automatic. A manual activity is performed by a human agent. An automatic activity is a program or service with specific interfaces. Here we restrict to automatic activities, because the treatment of manual activities is very similar, as briefly mentioned in Section 6 that describes our prototype. An activity consists of a number of elements, including the agent that is in charge of its execution, the program to be executed, as well as eventually the compensation program, the input and output data, etc. In this paper, an *agent* is the processing entity of the activity and is synonymous with the *site* at which the activity program runs. In what follows, A_a denotes an activity A to be executed by agent (at site) a. For activity A_a that can be logically undone, we use A^{-1}_a to denote the corresponding compensation activity.

A *blackbox flow* is a flow whose internal structure is only known by its own agent. Blackbox flows are useful for modular composition of flows and for integration of existing ones. They can be used for autonomy, privacy or security reasons. They also

allow different enactment mechanisms to be used for a single flow. For example, a blackbox flow can be managed by a central workflow engine while the rest of the flow is managed by the mechanism described in this paper. For the other agents, a blackbox flow behaves the same as an activity. In the rest of the paper, we treat blackbox flows as activities.

`seq` defines a sequence of sub-flows. **`fork`** spawns multiple parallel branches of sub-flows. The branches will be executed in parallel and then join at a *join agent* after their successful executions. The join agent can be automatically chosen if it is not explicitly specified. **`or`** enables execution from multiple alternative sub-flows, such that if the execution of a chosen sub-flow fails, one of the other alternative sub-flows can be chosen and executed. **`if`** defines a logical choice of execution according to a condition. A *condition* is defined on either flow-relevant data or execution status of the workflow. We defer the definition of condition on execution status to the next section. The sub-flow within a **`loop`** is repeated until the condition is evaluated to be false.

The example scenario in Figure 1 shows a flow for the arrangement of a trip for an employee to attend a training course AdBeans. The flow consists sequentially of an activity (or a blackbox sub-flow) for the reservation at the training course (*A* at *a*), followed first by a complicated sub-flow for the booking of airline tickets and a hotel room, and then by an activity (or a blackbox sub-flow) for the approval of the trip details by the manager of the employee (*E* at *e*). The complicated sub-flow consists of two parallel branches: an **`or`** sub-flow for the reservation of a room at either hotel BedBreakfast (*B* at *b*) or Continental (*C* at *c*), and an activity (or a blackbox sub-flow) for the reservation of airline tickets (*D* at *d*). The two branches will join at agent *e*. The join agent *e* can be automatically generated. Here it is explicitly specified for better readability.

3 The CEKK Machine

We introduce an abstract state machine called CEKK for distributed and recoverable flow enactment. The enactment of a flow is the process of interpreting 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 flow 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 flow has a corresponding local CEKK machine. The global state of the flow is the aggregation of the local states and the global state transitions are defined solely by the local state transitions. This important property, which will be clear when the state transitions rules are presented, assures that no global coordination among the agents is needed for 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 expression, *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 expression and the continuations together represent the work yet to be carried out.

A control expression *c* represents the next (sub-)flow to be enacted immediately. It is an expression in the core flow model extended with automatically generated continuation frames, to be described below.

A *continuation* is the reminder of execution after the control expression. *ks*, the *success continuation*, is the continuation towards the successful end of the flow. *kf*, the *failure continuation*, is the continuation towards the compensated end of the flow, after some eventual failure in the subsequent execution of the flow. A continuation is represented as a stack of continuation frames. A *continuation frame* is itself a flow as defined in the core flow model, extended with constructs automatically generated during enactment. For a continuation $k = f_n : \ldots f_1 : f_0 : \perp$, we write k.head = f_n and k.tail = $f_{n-1} : \ldots f_1 : f_0 : \perp$. When $k \neq \perp$, the last $\perp$ in k is normally omitted. When a continuation k is *applied*, k.head, i.e. the continuation frame at the top, becomes the control expression of the new state.

Formally, a continuation frame is of the form:

Continuation Frame ::= Flow | **`orc(or(`**Flow***`),`** *ks*`,` *kf***`)`**

| **`join(`***join-agent*`,`** condition**`)`**

where **`orc`** (for *or-closure*) and **`join`** frames are automatically generated during the enactment of **`or`** and **`fork`** flows respectively. A join is successful only when its condition is evaluated to be true.

An environment *e* is the runtime context of the flow. Information contained in *e* includes flow-relevant data and knowledge of status of the current execution so far. The execution status consists of a set of primitive status:

Primitive Status = Activity Status | Blackbox Flow Status | Join Status

For every primitive flow *F*, which is uniquely identified within the flow tree, we use succ(*F*), fail(*F*), none(*F*) and unknown(*F*) to denote a success, failure (aborted), not-enacted and unknown status of *F*.

A *condition*, included in **`if`** flows or **`join`** frames, can be evaluated in an environment. Of particular interest are the conditions on the current execution status:

Condition ::= Primitive Status | **`and(`**Condition***`)`** | **`or(`**Condition***`)`**

4 Distributed Workflow Enactment with CEKK

The enactment of a flow in our approach is the process of transitions of CEKK states performed by the agents. 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 flow enactment, this corresponds to a message $\langle c, e, ks, kf\rangle$ from p to q.

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

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

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

   where c_0 is a **`fork`** flow. 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:

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

   $\langle c_u, e_u, ks_u, kf_u\rangle_p$

   where $c_1, c_2, \ldots, c_n$ are **`join`** frames. 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.

Below are the state transition rules of the CEKK machine:

$\langle A_p, e, ks, kf\rangle_q \rightarrow \langle A_p, e, ks, kf\rangle_p$	if $p \neq q$	(A1)
$\langle A_p, e, ks, kf\rangle_p \rightarrow$		
$\langle ks.\text{head}, \text{succ}(A_p):e, ks.\text{tail}, A^{-1}{}_p:kf\rangle_p$	if $\text{succ}(A_p)$	(A2)
$\langle kf.\text{head}, \text{fail}(A_p):e, kf.\text{tail}, kf\rangle_p$	if $\text{fail}(A_p)$	(A3)
$\langle$**`seq(`**fs**`)`**$, e, ks, kf\rangle_p \rightarrow$		
$\langle fs.\text{head}, e, ks, kf\rangle_p$	if $\lvert fs\rvert = 1$	(S1)
$\langle fs.\text{head}, e,$ **`seq(`**$fs.\text{tail}$**`)`**$:ks, kf\rangle_p$	otherwise	(S2)
$\langle$**`if(`**t, ft, ff**`)`**$, e, ks, kf\rangle_p \rightarrow$		
$\langle ft, e, ks, kf\rangle_p$	if **`eval_cond(`**t, e**`)`**	(I1)
$\langle ff, e, ks, kf\rangle_p$	otherwise	(I2)
$\langle$**`loop(`**t, f**`)`**$, e, ks, kf\rangle_p \rightarrow$		
$\langle f, e,$ **`loop(`**t, f**`)`**$:ks, kf\rangle_p$	if **`eval_cond(`**t, e**`)`**	(L1)

$\langle ks.\text{head}, e, ks.\text{tail}, kf\rangle_p$ otherwise (I2)

$\langle \textbf{or(}fs\textbf{)}, e, ks, kf\rangle_p \rightarrow$

$\langle fs.\text{head}, e, ks, kf\rangle_p$ if $|fs| = 1$ (O1)

$\langle fs.\text{head}, e, ks, \textbf{orc(or(}fs.\text{tail}\textbf{),}\ ks\textbf{,}\ kf\textbf{)}\rangle_p$ otherwise (O2)

$\langle \textbf{orc(or(}fs\textbf{)}, orcks, orckf\textbf{)}, e, ks, kf\rangle_p \rightarrow$

$\langle \textbf{or(}fs\textbf{)}, e, orcks, orckf\rangle_p$ (O3)

$\langle \textbf{fork(}q, f_1, f_2, \ldots, f_n\textbf{)}, e, ks, kf\rangle_p \rightarrow$ (F1)

$\{\langle f_1, e, join\_succ{:}ks, join\_fail{:}kf\rangle_p,$

$\langle f_2, e, join\_succ{:}ks, join\_fail{:}kf\rangle_p,$

$\ldots,$

$\langle f_n, e, join\_succ{:}ks, join\_fail{:}kf\rangle_p \}$

where

$join\_succ = \textbf{join(}q, \textbf{and(}\text{succ}(f_1), \text{succ}(f_2), \ldots, \text{succ}(f_n)\textbf{))}$

$join\_fail = \text{join}\textbf{(}p, \textbf{and(or(}\text{fail}(f_1), \text{succ}(f^{-1}{}_1)\textbf{),}$

$\textbf{or(}\text{fail}(f_2), \text{succ}(f^{-1}{}_2)\textbf{),} \ldots,$

$\textbf{or(}\text{fail}(f_n), \text{succ}(f^{-1}{}_n)\textbf{)))}$

$\langle \textbf{join(}p, jc\textbf{)}, e, ks, kf\rangle_q \rightarrow \langle \textbf{join(}p, jc\textbf{)}, e, ks, kf\rangle_p$ if $p \neq q$ (J1)

$\langle \textbf{join(}p, jc\textbf{)}, e, ks, kf\rangle_p \rightarrow$ (J2)

$\langle ks.\text{head}, \text{succ}(join_p){:}ejoin, ks.\text{tail}, kfjoin\rangle_p$ if $\textbf{eval\_cond(}jc, ejoin\textbf{)}$

where

$ejoin = join_p\textbf{.merge\_env(}e\textbf{)}$

$kfjoin = join_p\textbf{.merg\_kf(}kf\textbf{)}$

The transition rules are first grouped based on the control expressions of the CEKK states. For example, rules A1 to A3 apply to activities (or blackbox flows), rules S1 and S2 apply to **seq** flows, etc.

Applying rule A1, the execution of an activity is forwarded to the agent of that activity. If the execution of an activity A_p succeeds, rule A2 is applied; otherwise, rule A3 is applied. Rule A2 can be read like this: the environment is updated with the successful execution of A_p; the compensation activity $A^{-1}{}_p$ of A_p is pushed to the failure continuation, so that if some failure event occurs later with the flow, the committed effect of A_p will be logically undone by executing the compensation activity $A^{-1}{}_p$; the success continuation is applied. With rule A3, the failure continuation is applied when the execution of A_p fails. There could be different ways to cope with failures of the compensation activity. Here we adopt a simple strategy in

which the compensation activity is repeated forever. So in rule A3, failure continuation remains unchanged.

If a sequential flow consists of only one sub-flow, that sub-flow is enacted (rule S1). Otherwise (rule S2), the first sub-flow is enacted and the other sub-flows are pushed to the success continuation, i.e., they will be enacted after the successful execution of the first sub-flow.

For an **`if`** flow, the proper sub-flow is selected after the evaluation of the selection condition (rules I1 and I2).

For a **`loop`** flow, if the loop condition is evaluated to be true, the sub-flow is enacted and same **`loop`** flow is pushed to the success continuation for later iterations. Otherwise, the **`loop`** flow has ended successfully and the success continuation is applied.

If an **`or`** flow consists of only one sub-flow, that sub-flow is enacted next (rule O1). Otherwise (rule O2), the first alternative sub-flow will be enacted next, and the failure continuation will consists of only one **`orc`** frame (the or-closure) that encapsulates the other alternative sub-flows as well as the success and failure continuations before the **`or`** flow is enacted. The failure continuation will be applied when the execution of the first alternative sub-flow fails. When applied (rule O3), the other alternative sub-flows will be enacted with the encapsulated continuations.

Enacting a **`fork`** flow spawns multiple parallel branches, each being represented with a local CEKK machine (rule F1). Upon creation, all branches have the same success and failure continuations. With the new success continuation, the remaining of the flow will be enacted after a successful join *join_succ* of the branches at the join agent. The success of the join is defined by the join condition, which states that all branches must be completed successfully. With the new failure continuation, before the old failure continuation is enacted, the join *join_fail* at the agent that originated the fork will guarantee that all braches will either fail (and their effects aborted) or their committed effects be successfully compensated.

A join is forwarded to the join agent (J1). To enforce rule J2, the join agent maintains a join environment *ejoin*. When a branch completes and is to be joined, its environment *e* is merged into *ejoin*. If the join condition is evaluated to be true in *ejoin*, the success continuation will be applied; otherwise, the join agent waits for other branches to be joined. The new failure continuation *kfjoin* is generated by merging the failure continuations of the branches, as the following:

The failure continuation of branch i (i = 1, 2, .., n) is of the form:

$$kf_i{:}join\_fail{:}kf_{common}$$

where kf_i is the failure continuation of the branch, *join_fail* was generated with rule F1, and kf_{commom} is the failure continuation before the fork was enacted. The merge of the failure continuations of the branches is of the form:

$$\texttt{fork(}p, kf_1, kf_2, \ldots kf_n\texttt{)}{:}kf_{common}$$

where *p* is the agent that spawned the branches. That is, if some failure occurs later, the successfully executed **fork** sub-flow will be compensated by this automatically generated **fork** sub-flow.

5 Running the Example

We now illustrate how the example flow in Figure 1 is enacted. Figure 2 shows the sequence diagram of the enactment process when no failure occurs. Notice that in the diagram, only the necessary messages are sent between the agents. The messages are asynchronous in the sense that there are no synchronous return messages.

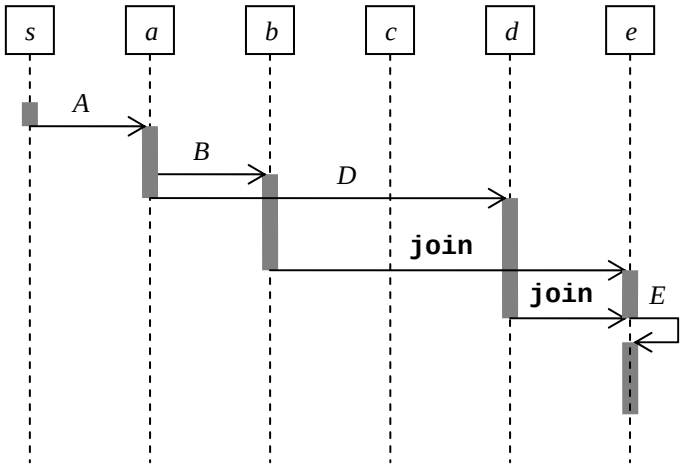


Figure 2. Sequence diagram of an enactment of the example flow

We assume that the flow is initiated at site *s*, which is, for instance, the desktop of the employee. We ignore flow relevant data in the environment. The initial global CEKK machine of the flow consists of only one local CEKK machine at agent *s*, with the **seq** flow as initial control expression, initial environment ε and empty success and failure continuations.

The first part of the flow is the reservation at the training course.

$\langle \mathbf{seq}(A_a, \mathbf{fork}(j, \mathbf{or}(B_b, C_c), D_d), E_e), \varepsilon, \perp, \perp \rangle_s$

$\rightarrow_{(S2)} \langle A_a, \varepsilon, \mathbf{seq}(\mathbf{fork}(j, \mathbf{or}(B_b, C_c), D_d), E_e), \perp \rangle_s$

$\rightarrow_{(A1)} \langle A_a, \varepsilon, \mathbf{seq}(\mathbf{fork}(j, \mathbf{or}(B_b, C_c), D_d), E_e), \perp \rangle_a$

$\rightarrow_{(A2)} \langle \mathbf{seq}(\mathbf{fork}(j, \mathbf{or}(B_b, C_c), D_d), E_e), \mathrm{succ}(A_a):\varepsilon, \perp, A^{-1}{}_a \rangle_a$

First the initial **seq** flow is enacted using rule S2, followed by a remote forwarding according to rule A1, because activity *A* for reservation at the training course is to be

executed by agent *a* at a remote site. Suppose now that the reservation is successful and its effect is committed. Rule A2 is now applied, so succ(A_a) is registered in the environment and the compensation activity $A^{-1}{}_a$ for canceling the reservation is pushed to the failure continuation.

The second part of the flow is the booking of a hotel room and airline tickets in two parallel branches.

$\rightarrow_{(S2)} <$ **fork(**j, **or(**B_b, C_c**)**, D_d**)**, succ(A_a):ε, **seq(**E_e**)**, $A^{-1}{}_a>_a$

$\rightarrow_{(F1)}$ {$<$ **or(**B_b,C_c**)**, succ(A_a):ε, **join(**j,js**)**:**seq(**E_e**)**, **join(**a,jf**)**:$A^{-1}{}_a>_a$,

$< D_d$, succ(A_a):ε, **join(**j,js**)**:**seq(**E_e**)**, **join(**a,jf**)**:$A^{-1}{}_a>_a$ }

where

js = **and(or(**succ(B_b)**,** succ(C_c)**)**, succ(D_d)**)**

jf = **and(or(**succ($B^{-1}{}_b$)**,** fail(C_c)**,** succ($C^{-1}{}_c$)**),**

or(fail(D_d)**,** succ($D^{-1}{}_d$)**))**

After rule S2 is applied, the control expression becomes a **fork** flow. Two parallel branches are spawned by applying rule F1, which is a local divergence. The branches have identical success and failure continuations, each with a new **join** frame. The success continuation states that the branches will join at the join agent *j* when they are successfully executed. The failure continuation states that the branches that either failed or are successfully compensated for will join at agent *a* where the branches were spawned.

The first branch is an **or** flow for hotel room booking.

$<$ **or(**B_b, C_c**)**, succ(A_a):ε, **join(**j,js**)**:**seq(**E_e**)**, **join(**a,jf**)**:$A^{-1}{}_a>_a$

$\rightarrow_{(O2)} < B_b$, succ(A_a):ε, *orcks*, **orc(or(**C_c**)**, *orcks*, *orckf***)**$>_a$

where

orcks = **join(**j,js**)**:**seq(**E_e**)**

orckf = **join(**a,jf**)**:$A^{-1}{}_a$

$\rightarrow_{(A1)} < B_b$, succ(A_a): ε, *orcks*, **orc(or(**C_c**)**, *orcks*, *orckf***)** $>_b$

$\rightarrow_{(A2)}$ <**join(**j,js**)**,succ(B_b):succ(A_a):ε,**seq(**E_e**)**, $B^{-1}{}_b$:**orc(or(**C_c**)**,*orcks*, *orckf***)**$>_b$

$\rightarrow_{(J1)}$ <**join(**j,js**)**,succ(B_b):succ(A_a):ε,**seq(**E_e**)**, $B^{-1}{}_b$:**orc(or(**C_c**)**,*orcks*, *orckf***)**$>_j$

Applying rule O2, the control expression becomes B_b for booking at BedBreakfast. It is then forwarded to agent *b* according to rule A1. Assume a room is successfully booked at BedBreakfast. After applying rule A2, the control expression becomes the **join** frame that was generated with the enactment of the **fork** flow. This is forwarded to the join agent *j* applying rule J1.

The second branch consists of only D_d for booking of airline tickets.

$< D_d$, succ(A_a):ε, **join(**j,js**)**:**seq(**E_e**)**, **join(**a,jf**)**:$A^{-1}{}_a > {}_a$

$\rightarrow_{(A1)} < D_d$, succ(A_a):ε, **join(**j,js**)**:**seq(**E_e**)**, **join(**a,jf**)**:$A^{-1}{}_a > {}_d$

$\rightarrow_{(A2)} <$ **join(**j,js**)**, succ(D_d):succ(A_a):ε, **seq(**E_e**)**, $D^{-1}{}_d$:**join(**a,jf**)**:$A^{-1}{}_a > {}_d$

$\rightarrow_{(J1)} <$ **join(**j,js**)**, succ(D_d):succ(A_a):ε, **seq(**E_e**)**, $D^{-1}{}_d$:**join(**a,jf**)**:$A^{-1}{}_a > {}_j$

After the successful execution of D_d, the state with the **join** frame as the control expression is forwarded to the join agent, similar to the first branch.

When one branch completes and reaches the join agent j, rule J2 is applied. The successful completion of the branch is registered in the join environment at j, but the join condition is not true yet, so j waits for the completion of the other branch. When the last branch reaches j, the join condition is evaluated to be true and the enactment will carry on as local convergence.

$\rightarrow_{(J2)} <$ **seq(**E_e**)**, succ($join_j$):succ(B_b):succ(D_d):succ(A_a):ε, ⊥, $kfj > {}_j$

where kfj = **fork(**a, $B^{-1}{}_b$, $D^{-1}{}_d$**)**:$A^{-1}{}_a$

The last part of the flow is the approval of the trip details.

$\rightarrow_{(S1)} < E_e$, succ($join_j$):succ(B_b):succ(D_d):succ(A_a):ε, ⊥, $kfj > {}_j$

$\rightarrow_{(A1)} < E_e$, succ($join_j$):succ(B_b):succ(D_d):succ(A_a):ε, ⊥, $kfj > {}_e$

$\rightarrow_{(A2)} <$ ⊥, succ(E_e): succ($join_j$):succ(B_b):succ(D_d):succ(A_a):ε, ⊥, $E^{-1}{}_e$:$kfj > {}_e$

Finally when the trip details are approved, the successful executions of all activities are registered in the environment and the control expression becomes an empty flow. The enactment terminates.

Next, we explain with two cases how automatic recovery works when some failure events occur.

In the first case, B_b for booking at BedBreakfast fails.

$<$ B_b, succ(A_a):ε, *orcks*, **orc(or(**C_c**)**, *orcks*, *orckf***)** $> {}_b$

$\rightarrow_{(A3)} <$**orc(or(**C_c**)**,*orcks*,*orckf***)**,fail(B_b):succ(A_a):ε,⊥,**orc(or(**C_c**)**,*orcks*,*orckf***)**$> {}_b$

$\rightarrow_{(O3)} <$ **or(**C_c**)**, fail(B_b):succ(A_a):ε, **join(**j,js**)**:**seq(**E_e**)**, **join(**a,jf**)**:$A^{-1}{}_a > {}_b$

$\rightarrow_{(O1)} < C_c$, fail(B_b):succ(A_a):ε, **join(**j,js**)**:**seq(**E_e**)**, **join(**a,jf**)**:$A^{-1}{}_a > {}_b$

Rule A3 is now applied instead of rule A2. After the failure continuation is applied, the control expression becomes an **orc** frame. The **orc** frame encapsulates the alternative sub-flow for booking at Continental as well as the success and failure continuations, so that the flow can be enacted either forward when the alternative sub-flow succeeds or backward when the alternative sub-flow also fails.

In the second case, E_e for approving the trip details fails.

$<$ E_e, succ($join_j$):succ(B_b):succ(D_d):succ(A_a):ε, ⊥, $kfj > {}_e$

$\rightarrow_{(A3)} <$**fork(**a,$B^{-1}{}_b$,$D^{-1}{}_d$**)**,fail(E_e):succ($join_j$):succ(B_b):succ(D_d):succ(A_a):ε, $A^{-1}{}_a$,$kfj > {}_e$

Rule A3 is now applied and the booking of the airline tickets and the hotel room will be cancelled in parallel followed by the cancellation of the reservation at the training course

6 Implementation

We have implemented a prototype for distributed flow enactment based on the state transition rules of the CEKK machine. The local architecture of the prototype at each site is shown in Figure 3. A CEKK state is represented in a message. The flows to be enacted at the site are first put in the message queue of that site (1). An agent is a thread (or a pool of threads) of control that performs the enactment of flows delivered to this site. To enact a flow, it dequeues a message from the message queue (2), decides the next action according to the control expression and updates the message based on the state transition rules. For a local activity, it invokes the program of the activity (3). For a manual activity, the activity program manages a worklist and interacts with human users. The return message from the activity program is put back into the message queue (4). The state transition rules A2 and A3 are applied later when the return message is dequeued (2 again). For a state transition of the form local ongoing, divergence or convergence, the updated messages are enqueued back to the message queue (5). For a remote forwarding, the message is sent to the site of the corresponding agent (6).

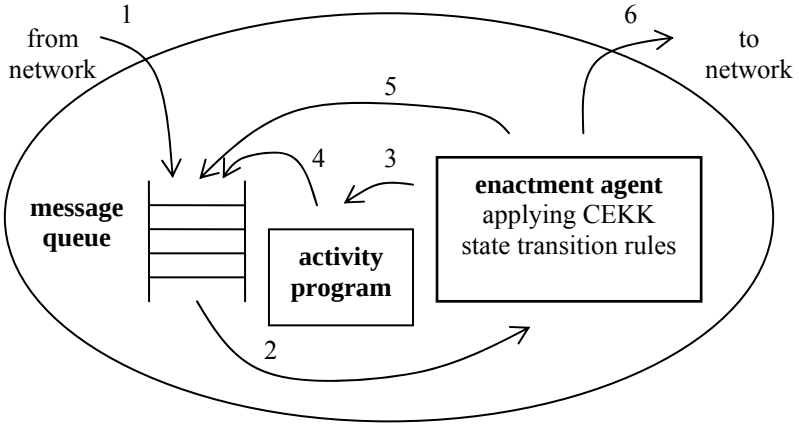


Figure 3. Local architecture at a site for flow enactment

For the joining of multiple parallel branches, the enactment agent maintains a persistent join state to build the new environment and failure continuation according to rule J2. The persistent join state is updated when a join message of a branch is processed. A join is successful when the join message of the last branch is processed. Currently, timeout at join agents is used for the detection of failures in parallel

branches. More sophisticated failure detection mechanisms are needed when timeout is inappropriate.

In the current implementation, messages are in the form similar to the notation used throughout this paper. This could be enhanced by BPEL4WS specifications extended with continuations. For our proof-of-concept prototype to be practically useful, some further extensions are necessary. For example, recoverable message queues [5], which have been successfully used for distributed workflow executions in logically centralized approaches [2][21], can be used for reliable message transmission and message queuing. The two-phase commit protocol used in recoverable message queues makes the system tolerant to communication failures and partial system crashes during remote forwarding. In a dynamic environment, the agent of an activity might not be known. Instead, it may be described by a number of properties. In such cases, some activity discovery mechanisms (similar to [4][7]) must be adopted to locate the agent before a message is sent for enactment.

7 Related Work

We first discuss related work on workflow enactment and then on applications of continuations.

The concept of workflow can be used as a methodology for software development, because it separates the control concern from the other concerns of the applications. Micro-workflow [22] provides a software framework to achieve this. The framework uses the trampolined continuation-passing style [12] for workflow enactment, similar to our work. There is limited support for distribution: remote workflows are enacted using proxies, similar to synchronous remote procedure calls. There is no support for recovery of workflows.

More often, workflows are used in distributed environments. Workflow enactment is typically achieved by workflow engines on dedicated centralized nodes, although this is generally regarded as neither scalable nor reliable [1]. In a typical implementation [20], all information about a workflow is stored in a database at the workflow engine. Information stored include: workflow specifications, workflow and activity instances and their execution states. Workflow enactment is the process of receiving messages from agents, consulting and updating the database, and sending messages to the appropriate agents for further execution. Some workflow engines adopt rule-based approaches.

To cope with the demanding requirements of scalability and reliability, some techniques are applied to the logically centralized approach, such as replication or hot pooling. For example, in [16], the execution of a workflow engine is spread on a cluster of servers coordinated via a shared tuple space.

Apart from the logically centralized approaches, decentralized workflow enactment has also received much attention in research. We classify them into two groups: compile-time distribution and run-time distribution.

With compile-time distribution, the workflow specification is analyzed and instantiated before execution. During a workflow instantiation, the necessary

resources and control are allocated in the distributed environment based on the analysis. The distribution of the enactment mechanisms either forms a hierarchy of sub-engines (Aurora [23], γ-calculus [26]), coordinating peer sub-engines based on a partitioning of the original specification ([8], flowcharts [10], METUflow [13], Mentor [25], SwinDew [32]) or directly down to the agents or other primitive units (Excotica/FMQM [2], Self-Serv [4], ORBwork [18]). As a common problem to these approaches, resources are allocated even for the part of the workflow that is actually not executed (such as some of the alternative paths or when a workflow rolls back at an early stage). They also tend to have limited adaptability at run time, because the control is mostly already in place before the execution started.

With run-time distribution, the information about the control of execution is carried along with the messages at runtime, as what happens with our approach. In AltaVista Works [6] and WORM [28], part of the static specification of the workflow (as mobile code in [28]) is sent from agent to agent for further enactment. This inherently disallows the kinds of processing that depend on runtime information, such as automatic recovery. In [14], the information passed along to the agents is very similar to the CEKK states: the part to be processed now (like control expression) and the unprocessed part (like success continuation). However, only the sequential structure is presented. Nested parallel branches are converted into a sequence structure, which seriously limited its general usage. INCA [3] is a rule-based system that has many properties very closed 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 enactment. Thus the rules and the log play the role of success and fail 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. INCA rules only prescribe one level of control. For nested structures, a new INCA is created for the enactment of a next-level sub-flow, which, after execution, will return the control back to the invoking agent. That is, message passing between different levels in the nested structure occurs in a synchronous manner.

INCA is the only work in the second group that supports automatic recovery of workflows. 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").

The distinct features of our approach, as compare to the related approaches for decentralized workflow enactment, are: (1) It does not involve an analysis of the workflow specification before execution (as opposed to the approaches in the first group), so it does not unnecessarily pre-allocated resources for the part of the workflow that is not executed and it is inherently more suitable for fast development and dynamic adaptation. (2) It builds on a theoretically elegant abstraction, continuation, so it can treat the whole workflow with different structures in a uniform manner. Consequently, our approach does not require global coordination and only asynchronous messages are sent when necessary, whereas other approaches typically involve synchronous return messages á la remote procedure calls. (3) It provides

automatic recovery of workflows (as opposed to all approaches in the second group except INCA).

Continuation has a long history [27], with applications in language theory, compiler and interpreter design, and web server implementations. Here we relate only to the applications and extensions most relevant to our work.

Continuations have been used in efficient implementations of web servers "to invert back the inversion of control" ([9] provides a comprehensive account of this work). This provides better scalability of web servers, because a server, by keeping the continuation in some shared data structure, does not have to hold a thread waiting for the next call from the same client. Links [9] extends this approach in the so-called resumption-passing style. In Links, the continuation is passed to the client, which is later passed back to the server for the next call (like a cookie). This provides even better scalability when there is a pool of servers, because another server can resume execution from the next call.

Our CEKK machine is built on CEK^T [17] and PCKS [24], which are extensions to the CEK machine [11]. With the communication-passing style based on the CEK^T machine [17], a distributed program can be executed asynchronously. A continuation is passed to the agent with a procedure call, which, after execution, will apply the continuation to the next agent instead of returning the control back to the caller. In [17], however, only one (distributed) thread of control is supported. The PCKS machine supports parallel executions of functional programs in a shared-memory environment [24].

Success and failure continuations have been applied in the execution (with backtracking) of logical programs [29] and the description of denotational semantics of stateflows [15]. There, the use of success and failure continuations is similar to the treatment of `or` flows in our work. To our knowledge, there has been no use of failure continuations for the purpose of recovery.

Our contributions to the work on continuations are: (1) its application is extended to distributed enactment of workflows (as opposed to distributed or parallel functional programs) and (2) success and failure continuations are introduced for automatic recovery.

8 Conclusion and Future Work

Workflows are increasingly applied in various new areas of distributed computing where scalability, reliability and adaptability are among the key requirements. Traditionally, and still with current commercial products, workflow enactment is realized with centralized workflow engines, which introduce a performance bottleneck and a central point of failure. In areas like Web services composition, assuming the existence of central workflow engines is sometimes impractical.

There have been increasing research efforts on decentralized (distributed) workflow enactment. These approaches can be classified as either compile-time distribution or run-time distribution. In the compile-time distribution approaches, the workflow specification is analyzed, and the necessary control information and resources are pre-

allocated before execution. Resources are thus allocated or consumed even for the part of the workflow that is not executed. There is also limited adaptability at runtime. In the run-time distribution approaches, the control information is passed along with messages during execution. Run-time distribution approaches are potentially more scalable (no central performance bottleneck), more reliable (no central point of failure), more adaptable (no static analysis) and have better utilization of system resources (no routing through a central engine, no unnecessary pre-allocation of resources etc.).

Our contribution is a new run-time distribution approach to distributed workflow enactment. 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 workflow enactment as local operations rather than global coordination. Furthermore, our approach allows for automatic workflow recovery by automatically generating recovery plans into failure continuations.

Our current results are promising but still preliminary. There remain a number of interesting open issues:

Although it is widely believed that decentralized workflow enactment is more scalable and reliable, a detailed performance study is needed to confirm this. The performance study should also include comparisons between compile-time and run-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 a dynamically adaptable and re-configurable approach, for example, by using the blackbox flows in our core flow model.

Our approach does not require global coordination for workflow enactment. Some other tasks for workflow management, for example, monitoring and query of workflows, may still need distributed global coordination among parallel branches of a workflow. We are currently working on a scheme for delegating these tasks to fork and join agents.

Our approach has some special requirements on security and privacy, since the control information as continuations is now passed along. Part of the solutions might lie in the use of blackbox flows in the core flow model, because control information within the blackbox flow is unknown outside the agent of the blackbox flow.

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] Böszörményi, L., R. Eisner and H. Groiss, “Adding Distribution to a Workflow Management System”, *10th International Workshop on Database & Expert Systems Applications (DEXA 99)*, pp. 17-21, September, 1999.

[7] Casati, F. and M.-C. Shan, “Dynamic and Adaptive Composition of e-Services”, *Information Systems*, 26(3), pp 143-163, 2001.

[8] Chafle, G., S. Chandra and V. Mann, “Decentralized Orchestration of Composite Web Services”, *13th international World Wide Web conference (Alternate track papers & posters)*, pp 134-143, May, 2004.

[9] Cooper, E., S. Lindley, P. Wadler and J. Yallop, “Links: Web Programming Without Tiers”, submitted to ICFP 2006.

[10] Dong, G., R. Hull, B. Kumar, J. Su and G. Zhou, “A Framework for Optimizing Distributed Workflow Executions”, *7th International Workshop on Database Programming Languages*, LNCS 1949, Springer-Verlag, September, 1999, pp 152-167.

[11] Felleisen, M. and D. P. Friedman, "Control operators, the SECD-machine, and the lambda-calculus". *3rd Working Conference on the Formal Description of Programming Concepts*, August 1986.

[12] Ganz, S. E., D. P. Friedman and M. Wand. "Trampolined Style". *International Conference on Functional Programming (ICFP 99)*, September 1999.

[13] 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.

[14] Guo, L., D. Robertson and Y.-H. Chen-Burger "A Novel Approach For Enacting Distributed Business Workflows Using BPEL4WS on the Multi-Agent Platform", *IEEE Conference on E-Business Engineering (ICEBE 2005)*, October, 2005.

[15] Hamon, G., “A Denotational Semantics for Stateflow”, ACM Conference on Embedded Software (EMSOFT 05), pp 164-172, September, 2005.

[16] 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.

[17] Jagannathan, S., "Communication-Passing Style for Coordination Languages", *2nd International Conference on Coordination Models and Languages*, LNCS 1282, Springer-Verlag, September 1997.

[18] Kochut, K. J., A. P. Sheth, and J. A. Miller, "ORBWork: A CORBA-based Fully Distributed, Scalable and Dynamic Workflow Enactment Service for METEOR". Technical Report UGA-CS-TR-98-006, Dept. of Computer Science, Univ. of Georgia, 1998.

[19] Korth, H. F., E. Levy and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions", *16th International Conference on Very Large Data Bases (VLDB 90)*, pp 95-106 August, 1990.

[20] Leymann, F. and D. Roller *Production Workflow: Concepts and Techniques*, Prentice Hall, 2000.

[21] Leymann, F. and D. Roller, "Building A Robust Workflow Management System With Persistent Queues and Stored Procedures", *International Conference on Data Engineering (ICDE 1998)*, February 23-27, 1998, pp 254-258.

[22] Manolescu, D. A., "Workflow Enactment with Continuation and Future Objects", *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, November, 2002, pp 40-51.

[23] Marazakis, M., D. Papadakis and C. Nikolaou, "Aurora: An Architecture for Dynamic and Adaptive Work Sessions in Open Environments", *9th International Conference on Database and Expert Systems Applications (DEXA 98)*, pp. 480-491, LNCS 1460 Springer-Verlag, August, 1998.

[24] 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, Springer-Verlag, April 1994.

[25] 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.

[26] Németh, Z., C. Pérez and T. Priol, "Workflow Enactment Based on a Chemical Metaphor", *3rd International Conference on Software Engineering and Formal Methods (SEFM'05)*, September, 2005.

[27] Reynolds, J. C., "The Discoveries of Continuations", *Lisp and Symbolic Computation* 6(3-4), pp 233-248, 1993.

[28] 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.

[29] Todoran, E. and N. Papaspyrou, "Continuations for Parallel Logic Programming", 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000), pp 257-267, September, 2000.

[30] Van der Aalst, W. M., "Process-Oriented Architectures for Electronic Commerce and Interorganizational Workflow", *Information Systems*, 24(18), pp 639-671, December, 1999.

[31] Van der Aalst, W. M., A. H. M. ter Hofstede, B. Kiepuszewski and A. P. Barros, "Workflow Patterns", *Distributed and Parallel Databases*, 14(1), pp 5-51, 2003.

[32] Yan, J., Y. Yang, and G. Raikundalia, "Enacting Business processes in a Decentralised Environment with p2p-Based Workflow Support", *4th International Conference on Web-Age Information Management (WAIM 03)*, LNCS 2762, pp 290-297, Springer-Verlag, September, 2003.