ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Decision Support Systems 40 (2005) 89–105

Decision Support Systems

www.elsevier.com/locate/dsw

Meta workflows as a control and coordination mechanism for exception handling in workflow systems

Akhil Kumar[a,*], Jacques Wainer[b]

[a] Penn State University, 509 BAB Building, University Park, PA 16802, USA
[b] Institute of Computing, State University of Campinas, Campinas, 13083-970, Sao Paulo, Brazil

Available online 2 June 2004

Abstract

A higher level control and coordination mechanism is required for exception handling in workflow systems. This paper describes such a framework based on events, states, and a new kind of process called a meta workflow. Meta workflows have five kinds of meta activities and facilitate control over base workflows. We describe the framework and illustrate it with examples to show its features. The paper gives an architecture for incorporating it into existing workflows and also provides a formal semantics of execution. This framework can be used in Web services, supply chains, and inter-organizational applications where coordination requirements are complex, and flexible and adaptable workflows are needed. It is also useful for handling not just failure recovery but also various kinds of special situations, which arise frequently in web-based applications.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Workflow; Meta workflow; Coordination; Exception handling; Event-state-process (ESP) framework; WQM model; BPEL4WS

1. Introduction

Web services and supply chains [3,13,17,12] are workflows that span multiple organizations. Thus, they require coordination and control of both the data flow and control flow across multiple organizations. Exceptions are a common occurrence in workflows [4,11,15,18]. Even a seemingly simple process like a travel expense claim or order processing can become difficult to describe if one tries to cover all the special situations and exceptions in the description. This creates a very awkward process description that is hard to read and understand, and also error-prone. Therefore, most workflow systems are able to capture the simpler form of a process and tend to collapse when variations are introduced. Exceptions can be *planned* or *unplanned*. A planned exception is an abnormal situation that has been anticipated and a way for handling it has been included in the process. On the other hand, an unplanned exception is one that has not been anticipated.

In this paper, we present a formal methodology for describing exceptions in a workflow. An exception is a special situation that occurs infrequently in a workflow. The main idea is to describe a basic, primary process first and treat abnormal and infrequent situations separately as supporting workflows. Our goal is to provide support for the planned exceptions and also be able to incorporate the unplanned ones relatively easily. We introduce two new notions, meta workflows and ESP (Event-state-process). A *meta workflow* is a special, higher level control process that consists of five control commands: *start*, *terminate*, *suspend*, *resume* and *wait*

* Corresponding author.
E-mail addresses: akhilkumar@psu.edu (A. Kumar), wainer@ic.unicamp.br (J. Wainer).

0167-9236/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.dss.2004.04.006

and *suspend*. An *ESP rule* causes a meta workflow to run when an event occurs and a workflow case is in a certain state. This may cause a meta workflow to execute, and thus perform control operations like suspending certain workflows and starting other workflows, etc. Thus, we make a clear distinction between two types of workflows: *base workflows* and *meta workflows*. The base workflow corresponds to specific tasks that must be performed. The meta workflow is only for control purposes and consists of the control commands described above. In general, when it is not qualified the term workflow refers to a base workflow.

The advantages of this approach are *modularity, extensibility* and *adaptability*. The basic workflow description is kept simple while variations to the basic process are described separately in a *modular* manner. It is possible to add new ESP rules and corresponding base- and meta workflows when a new situation arises thus giving extensibility. Minor changes may be made to ESP rules (or a module) to adapt the workflow to certain situations. Such small changes would be hard to make on a monolithic workflow. Finally, the simplicity of the approach helps in minimizing errors, while at the same time making it easier to describe complex situations.

Consequently, this framework constitutes a new methodology for workflow modeling which has applications in various kinds of web services. The organization of this paper is as follows. Section 2 gives a formal description of our framework along with semantics. Then Section 3 illustrates the framework with examples. Next, Section 4 discusses an architecture and expressive power of our framework, while in Section 5 we present some discussion of this approach in the context of related work, in particular BPEL4WS. Finally, Section 6 concludes this paper.

2. Formal description

The ESP framework involves the concepts of workflows, events, states and meta activities.

2.1. Workflow and workflow instances

At the outset it is important to keep in mind the distinction between process classes (or templates) and process instances. A process definition, say "fabrication of a car" is a process class (or a template), composed of three activities, A, B, and C, to be performed in sequence. Activities A, B and C also belong to activity classes or templates. Both the fabrication process and the three activities are generic.

A particular instance of "fabrication of a car," say the car with the VIN XYZ345, is a case, or an instance of a process. Case XYZ345 (assuming the VIN is used as the case id) may be executing activity B; i.e., the activity B (for case XYZ345) is in the executing state. In the ESP framework, as we will see later, multiple processes may be invoked with the same case number. Thus, a process instance is initiated and assigned a case number, and the same case number is used as a reference when other process templates are invoked. For example, suppose the fabrication of car XYZ345 is to be canceled while B is the current activity. Hence, B must be stopped, and a new cancellation activity, say "returning the reusable parts to the inventory" might start, and other activities may follow. The case is still identified by id XYZ345, but, of course, it is no longer an instance of the fabrication process, but an instance of the cancellation process.

We use WQM (workflow query model Ref. [2]) as the language to describe workflows. In WQM, a workflow process is described in terms of 10 basic primitives or control elements: **task**, **start**, **end**, **sequence**, **split-choice**, **join-choice**, **split-parallel**, **join-parallel**, **start-while-do**, and **end-while-do** (see Ref. [2] for more details). In this paper we will use a graphic representation of WQM in which tasks are represented by rectangles, sequences by arrows, and the other control elements by ellipses with the appropriate labels for type and name. Fig. 1 is an example of such a graphical representation; a textual representation of the same workflows are possible using XML and the XML Schema for it is described in the appendix of this paper. This example describes a workflow for ordering a laptop and it will be discussed in more detail in the next section.

2.2. Activities and state of activities

The standard states for an activity within a workflow are:

- **not-ready**: if some of its prerequisite activities have not been done.

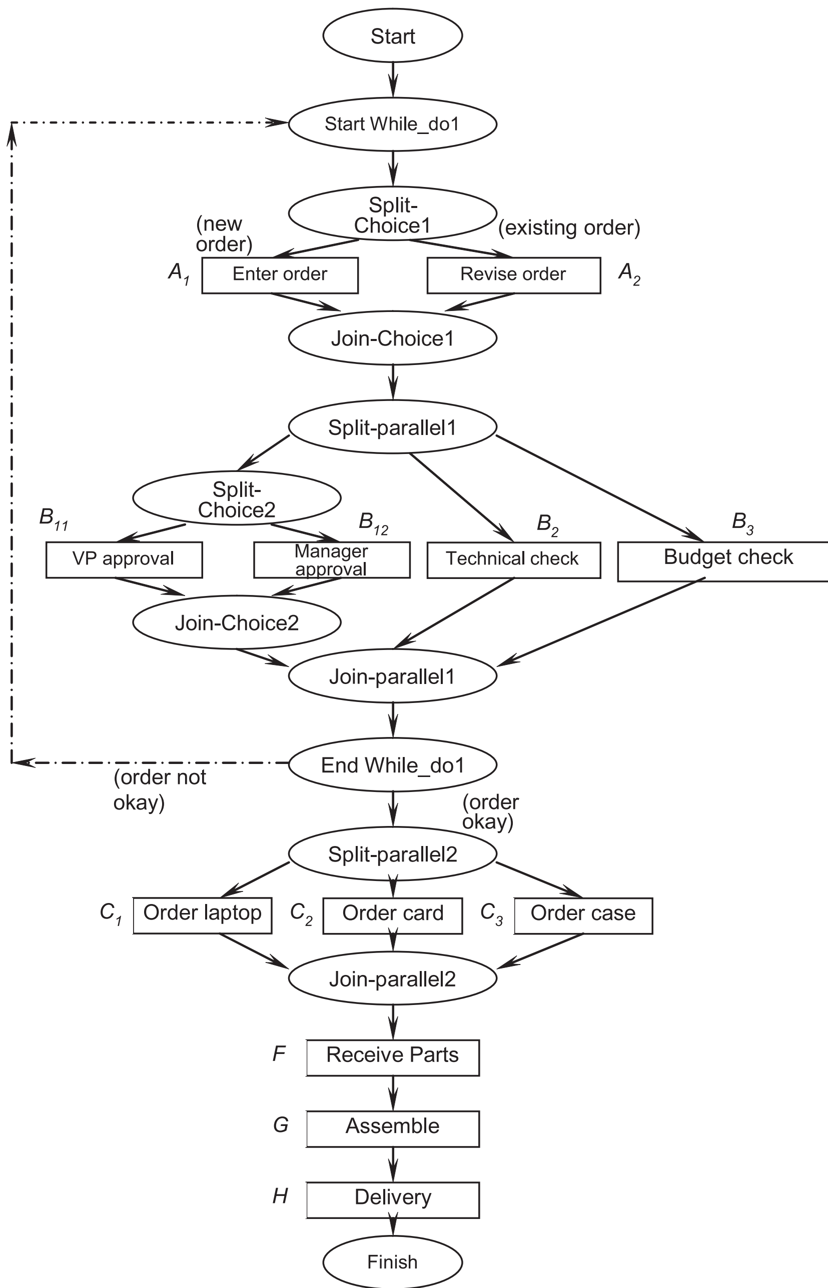

Fig. 1. An example base workflow, wf_1, using WQM model [2].

- **ready**: if all its prerequisites are done, but the activity has not yet started.
- **executing**: if the activity is executing.
- **suspended**: if the activity is suspended.
- **done**: if the execution of the activity has terminated.
- **aborted**: if the execution of the activity is aborted abnormally.

The transitions between these states are shown in Fig. 2. Because our workflow allows for loops using the *start-while-do* and *end-while-do* constructions, some activities may be executed more than one time. In the example in Fig. 1, the activities A1, A2, B11, B12, B2 and B3 may be executed more than one time for the same case. We then speak of a **round** of a *while-do* construction, and of **incarnations** of an activity. The execution of A1 in the first round of the loop is its first incarnation; the execution of A1 in the fourth round is its fourth incarnation, and so on. It may be the case that a particular activity will not have a corresponding incarnation because it was not chosen at a choice node: in the example in Fig. 1, the split choice in round 3 of the loop may decide to execute A2. In this case A1 will have no third incarnation.

Thus, for an activity, being **done** in one round is equivalent to being **not-ready** in the following round. Thus the diagram in Fig. 2 illustrates with the continuous lines the state transitions allowed within one incarnation. The dotted line indicates the only possible transition from one incarnation to another, that is, from **done** in one incarnation to **not-ready** in the next.

We extend the concept of states of activities to the other control elements next. For the *join-parallel* element:

- it will be in the **done** state if *all* its parents are in the **done** state
- it will be in the **abort** state if any of its parent is in the **abort** state
- else it will be in the **suspend** state if any parent is in the **suspend** state
- else it will be in the **executing** state if any parent is in the **executing** state. For the other control elements, the same rules apply, except for the first which is
- it will be in the **done** state if *any* parents are in the **done** state.

A workflow instance is **active** if any of its activities is not **done**; otherwise, the workflow case is **inactive**, which may mean that the case has already ended (all activities are **done**) or has not yet started. A workflow that is terminated before it completes execution is **aborted**.

2.3. Events

In the ESP framework we assume that events are instances of classes which carry the *case id* information. Thus, if case XYZ345 must be cancelled an event of class *cancellation* with *case id* XYZ345 must be generated.

We consider three super-classes of events. A *synchronous internal event* is an event generated by the workflow engine when an activity ends. An *asynchronous internal event* is generated by internal timers, for example when an activity has a deadline and it is late. Finally, an *external event* is an event generated from outside the workflow execution.

2.4. ESP framework

The ESP framework has two components: the workflow **description part** and the **ESP rules**. The description component is of the form:

Base-workflow-id: workflow-definition

The description component describes a workflow process and associates a workflow name to it so that

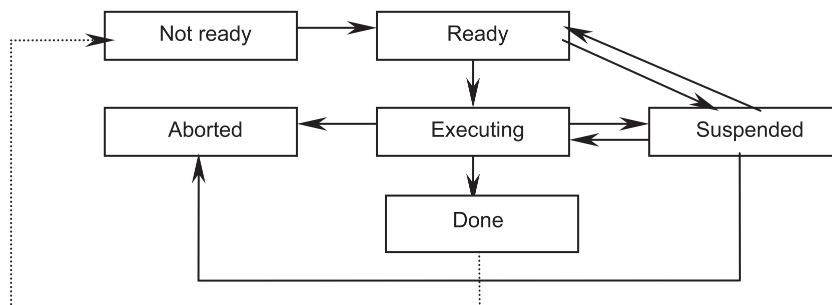

Fig. 2. State transition diagram.

the activation rules refer to it by that name. In general, there would be several base workflows. In this paper, the workflow processes are shown as figures for easy readability, although they can be written formally in XML also.

The **ESP rules** are of the form:

Base-workflow-id/event-class : state
→ meta-workflow-id

The basic semantics of an ESP rule is the following. If an event of *event-class* is received for a case (called the **current case**), and the base workflow given by the *base-workflow-id* is active for the current case, and is in the state given in the rule, then the corresponding meta workflow is started for the current case. In this case, we will say that the event was **captured by** the rule.

Furthermore, we assume the following:

- once an event is captured by a rule, no other rule is tested. Thus, the rules should be organized by priority such that the most important rule is listed first. Once an event is captured by a high priority rule it will not trigger any of the less important rules.
- an event may not be captured by any rule. In this case nothing happens.
- we assume that all ESP rules are evaluated atomically, that is, the state of the workflow does not change during the evaluation of the ESP rules. We assume that the computational infrastructure is able to implement such atomicity. A possible implementation is that whenever an event is received, the workflow engine is interrupted until all rules are evaluated. If none of the ESP rules fire, then the workflow resumes automatically. On the other hand, if a rule qualifies to fire, then the workflow is suspended automatically. Other more complex mechanisms are possible.

The name **ESP** is an acronym for **E**(events) **S**(states) **P**(process), a deviation from the more common Event, Condition, Action (ECA) rules [14].

2.5. Meta activities and meta workflows

What distinguishes a meta workflow from a base workflow is that a meta workflow only has five special activities, called **meta activities** that control and operate the base workflows. These meta activities are:

- **start(*wf*,[*c*])**: starts the base workflow *wf* and associates it with the current case. If the argument *c* is present, then the base workflow *wf* is started and associated with a (possibly new) case *c*. *Start(wf)* will not wait for the workflow *wf* to finish—the meta activity only starts the workflow *wf* asynchronously and then finishes.
- **terminate(*wf*)**: terminates the base workflow *wf* for the current case. The terminate meta activity aborts all executing activities, and places the base workflow in the **inactive** state.
- **suspend(*wf*, [*A*])**: suspends the activity *A* in the base workflow *wf* for the current case. If activity *A* is not executing, it has no effect. Finally, if the argument *A* is missing, all executing activities are suspended.
- **wait and suspend(*wf*, [*A*])**: this meta activity waits for the activity *A* in the base workflow *wf* to be done and then suspends whatever activity or control element follows *A* in *wf*. If the argument *A* is missing, then the process is repeated for all executing activities in *wf*.
- **resume(*wf*, [*A*])**: resumes activity *A* in the base workflow *wf*. If *A* is not in suspended state in the base workflow, the meta activity has no effect. If argument A is missing, then the workflow resumes where it was suspended. All activities are resumed asynchronously, that is, *resume(wf,A)* will not wait until A is finished.

In the meta activities above, the argument for an activity could specify an activity name (such as *A*), or be an expression *A*+, which refers to the control path of the workflow immediately after *A*. This is a syntactic abbreviation that is convenient in the case of the *wait and suspend* activity: after a *wait and suspend*(*A*) one can execute a *resume*(*A*+), which will resume at whatever task or control element follows *A*.

A meta workflow is described as a sequence of meta activities, separated by ";". The operational semantics of each of these meta activities is given in Fig. 3 along with appropriate data structures.

```
Data structures: C[ ], active[ ], ready[ ], executing[ ], done[ ], suspend[ ], abort[ ];

c = current_case();
if X = start(wf,[n]) then
        if n is specified then c = n end if
        insert c into C, if not present;
        insert wf in active(c);
        place the first activity of wf for case c in the ready state;
else if X = terminate(wf) then
        move all activities of wf to aborted;
        remove wf from active(c);
else if X = suspend(wf, [A]) then
        if A ∈ {executing, ready} then move A into suspend; end if
        if A is unspecified then
        move all activities of wf from {executing,ready} into suspend; end if
else if X = wait&suspend (wf, A) then
        wait until A belongs to done in wf for c;
        insert A+ into suspend;
        end if
else if X = resume(wf, [A]) then
        if A is specified, then move A from suspend to ready or executing;
        end if
        if A is unspecified then
        move all activities in wf from suspend to ready or executing; end if
end if
```

Fig. 3. Operational semantics of the meta activities.

2.6. State representation in ESP framework

We will define a state representation in ESP as a logical formula which may refer to the states of activities (*done*, *executing*, *ready*, *aborted*, *suspended*) and to conditions that refer to case data (that is, application data that pertains to the current case). Since we extended the concepts of states to the control elements, one can also make reference to them on such expressions, as well as use the syntactic abbreviation A+ discussed above. The symbol *Done* refers to the set of all activities and control elements that are in the *done* state; similarly the set *Executing* refers to all activities and control elements in the *executing* state, and so on. Also, all expressions make reference to the current round of a *while-do* construct.

Thus, the formula:

$$(\mathrm{A} \in \mathrm{Done} \wedge \{\mathrm{B}, \mathrm{C}\} \subseteq \mathrm{Executing} \wedge \mathrm{Cost} \leq 300,000) \vee (\{\mathrm{A}, \mathrm{B}, \mathrm{C}\} \subseteq \mathrm{Done} \wedge \neg(\mathrm{D} \in \mathrm{Done}))$$

represents the situation in which activity A is **done,** activities B and C are still **executing,** and the case data *cost* is less than or equal to \$300,000, or *A, B,* and *C* are all **done** but *D* is not.

The semantics of referring only to the current round places some limitations on the expressive power of the language. For example, one cannot refer

to different rounds (say, *A* was executed in the last round but not in the current one) or count the number of incarnations of an activity (*A* and *B* where executed more than three times). However, this limitation can easily be overcome by minor changes in syntax.

3. Examples

In the previous section, we described the meta workflows framework that consists of a collection of base workflows, ESP rules and meta workflows. The operational semantics of this framework was also discussed in detail. In this section, we give examples to illustrate our framework. These examples illustrate generic situations that often arise in workflows and also consider synchronous, external and asynchronous events. These examples are discussed in the context of a basic setting that describes a base workflow for ordering a laptop.

3.1. Example 1

Consider a base workflow, wf_1, as shown in Fig. 1 [2], for ordering a laptop. This workflow corresponds to the WQM model, and includes several tasks or activities such as entering the order, obtaining multiple approvals, placing separate orders for components of the laptop, receiving the parts, assembly and delivery. The *split-choice* node allows one path to be taken based on a condition (e.g., new/old order). The *split-parallel* node allows multiple activities to proceed in parallel. The *While-do* construct is used to expression repetition. The various activities of this workflow process have been labeled (A, B_{11}, B_{12}, etc.) in the figure for ease of reference. The subsequent examples will make reference to these activities.

3.2. Example 2: workflow modification

In this example, we consider a process that will perform an "additional budget check" after the regular "budget check" activity is done, if the budget for this order is found to be greater than $3000, since that is the maximum allowed by company policy for a laptop. In this case, the VP of Finance and the President must approve the request as an exception since it exceeds the company limit. We name this subworkflow process shown in Fig. 4 as wf_2. It will be triggered by meta workflow mwf_1 if an internal synchronous event generated when the activity B_3 (from Example 1 of Fig. 1) ends (denoted as synch_event$_{11}$ = Done(B_3)) is received by the system and the associated condition is true. The condition requires that activity B_2 also be already done, the result from B_2 should be 'okay' and the Budget amount be greater than $3000. The rationale for this is that unless the technical check is approved, there is no point in sending this case for an additional budget approval (the same might also apply to the VP/manager approval but we omit that for simplicity). Thus, event$_{11}$ is captured by an ESP rule only if this compound condition based on the state and case data evaluates to true; otherwise, the event will be disregarded. If the condition is true, the ESP rule will trigger a meta workflow to suspend wf_1 and start wf_2 to perform the additional reviews. After the additional approvals finish, the event *Done*(wf_2) will trigger the meta

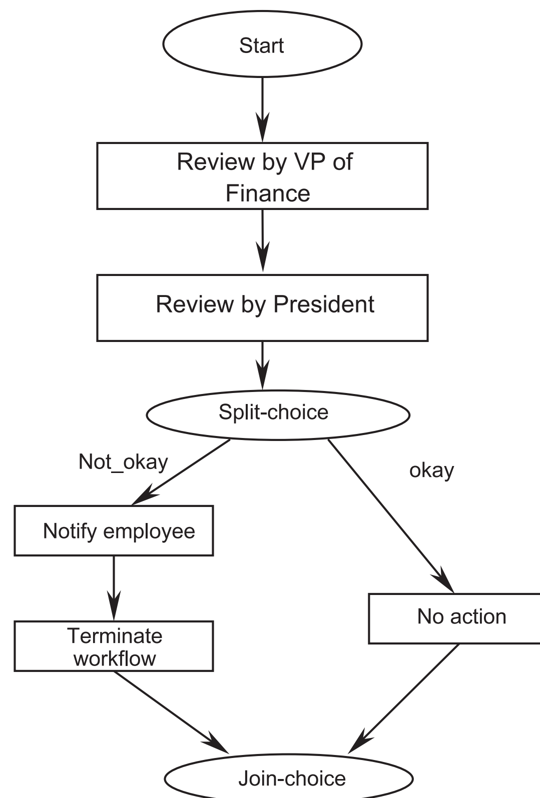

Fig. 4. A base workflow, wf_2, for additional budget approval.

workflow mwf_2 to resume the original base workflow wf_1. Now, if the condition is false, then the meta workflow is not activated and the original workflow keeps running. The second ESP rule is similar to the first and covers the case where B2 is done after B3. The states, meta workflows, and exception rules for this situation are shown in Table 1.

3.3. Example 3: workflow cancellation

Cancellation is a common activity in workflows. In many situations a cancellation workflow must be invoked in order to undo the partial effect of another workflow. Let us consider that the employee cancels her order after the orders have already been placed with the laptop, card and case vendors. In this case a cancellation workflow wf_3 will be started as shown in Fig. 5. The cancellation workflow must cancel the respective orders with the three vendors, and after receiving a confirmation from them, the employee should be notified. There may also be a cancellation charge that may apply to the employee's account. The event that initiates this is a cancel event. When this event is received by the ESP module, a condition check is performed to verify that the parts have not been received. If so, meta workflow mwf_6 is started. This will first suspend the base workflow and then run the cancellation workflow. Finally, the base workflow will be terminated.

The states, meta workflows, and exception rules for this event are shown in Table 2. Note that cancellation is an external event, as opposed to an event generated by a running workflow. Also note that if the condition check fails, i.e., activity F is in *executing* or *done* state, then another ESP rule will be required to perhaps run a different workflow. For

Table 1
Formal definition of additional budget check

Events	synch_event$_{11}$ = Done(B_3)
	synch_event$_{12}$ = Done(B_2)
	synch_event$_2$ = Done(wf_2)
Meta workflows	mwf_1: suspend(wf_1) ; start(wf_2);
	mwf_2: resume(wf_1);
ESP rules	wf_1/Done(B_3): B2∈Done∧B2.result = 'okay'∧ Budget>\$3000 → mwf_1
	wf_1/Done(B_2): B3∈Done∧B2.result = 'okay'∧ Budget>\$3000 → mwf_1
	wf_2/Done(wf_2) → mwf_2

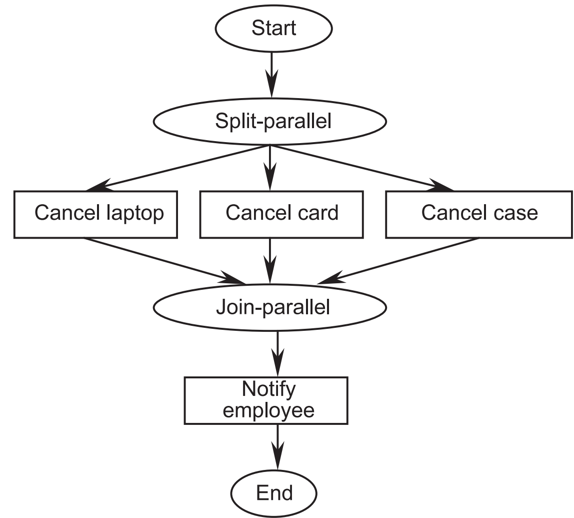

Fig. 5. A workflow, wf_3, for process cancellation.

instance, if some parts of the order have already arrived, then they may have to be returned to the vendor involving some additional steps. The workflow in Fig. 5 will need to be modified accordingly.

3.4. Example 4: change order—change an existing order

Next we consider how an order can be changed in response to a *change_order* event. This is a complex request because it might be permissible to change the order only if it has *not* gone into assembly. Therefore, when the event arrives, it is necessary to check whether the assembly task has started executing (or it might even be already done). In this case, the change order request is denied. On the other hand, if the order is neither done nor has it gone into assembly, then the main workflow *wf1* is suspended and workflow *wf4* is started. This workflow first calculates the amount of the revised order. Then, if the amount of the revised (new) order is less than the amount of the current order, then new components are ordered, any old components are returned and the workflow *wf4* (see Fig. 6) returns

Table 2
Formal definition of workflow cancellation

Events	ext_event$_3$ = Cancel
Meta workflows	mwf_3: suspend(wf_1); start(wf_3); terminate(wf_1);
ESP rules	wf_1/Cancel: $F \in ready \vee F \in not_ready \rightarrow mwf_3$

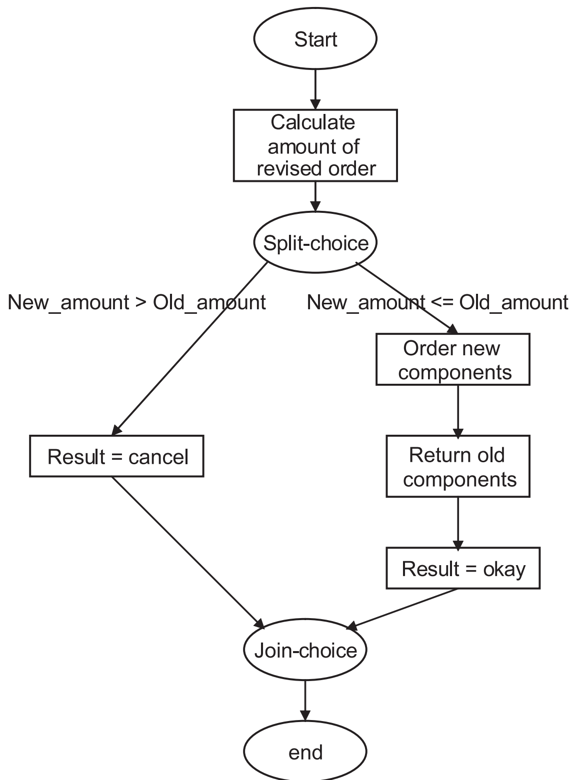

Fig. 6. A base workflow, wf_4, for change_order.

a result of 'okay.' Alternatively, if the new order amount is greater than that of the current order, the order must be cancelled and resubmitted. Next, the

Table 3
Formal definition of rules for handling change_order event

Events	ext_event_4 = change_request

Meta workflows	mwf_4: wait and suspend (*wf1*); start(wf_4);
	mwf5: start(*wf5*);
	mwf6: resume(*wf1*); start(*wf6*);
	mwf7: start(*wf3*); terminate (*wf1*); start(*wf7*);

ESP rules	wf_1/change_request: ¬(assemble ∈ executing) ∧ ¬(assemble∈done)∧join_parallel1∈done → mwf_4 *wf1*/change_request: assemble∈executing ∨ assemble∈done → *mwf5* *wf1*/done(*wf4*): result = 'okay' → *mwf6* wf1/done(*wf4*): result = 'cancel' → *mwf7*

done(wf4) event is captured, and based on the result being *okay* or *cancel*, metaworkflows *mwf6* or *mwf7*, respectively, are started. The meta workflow *mwf6* causes the main workflow *wf1* to resume and also starts *wf6* (see Fig. 7) to notify the customer that the change has been confirmed. On the other hand, meta workflow *mwf7* starts *wf3* to cancel the current order, then it terminates *wf1* and starts *wf7* (see Fig. 7) to notify the customer. Table 3 gives the states, meta workflows and exception rules for this example.

3.5. Discussion

Above we discussed three different examples of varying degrees of complexity to illustrate the features of the meta workflows approach. Status notification is another common scenario in workflow applications. In our running example, if the parts arrive late, the

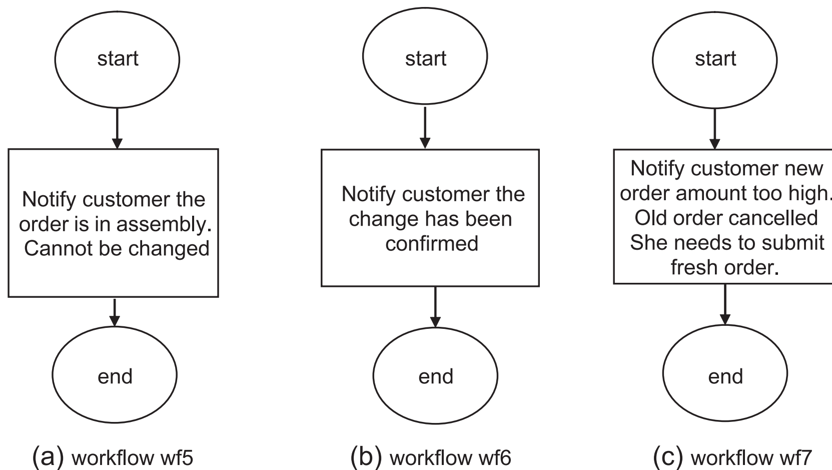

Fig. 7. Workflows *wf5*, *wf6* and *wf7*.

delivery of the final product may be potentially delayed and, hence, the assembly line, final customers, and suppliers should be notified. Therefore, after activity *F* of Fig. 1 is completed, an asynchronous event would be generated and a workflow associated with it.

These examples serve to illustrate the main idea behind this approach which is to decouple the control aspects from the process descriptions. Thus, we are able to exploit meta workflows as a mechanism for stitching together workflow snippets, where each snippet can serve as a component and be called from meta workflows. As an example, the cancel workflow was called in the *change_order* process implementation in Section 3.4. Moreover, business processes can be modified more easily by making changes to the ESP rules and adding small workflow snippets. For instance, in the *change_order* example, if the change order policy were to be modified such that changes could occur only if the parts had not yet been received, this could be achieved by only changing the ESP rules of Table 3.

4. Architecture and expressive power of ESP

4.1. Architecture

The architecture for incorporating this framework is shown in Fig. 8. The philosophy behind this architecture was to exploit an existing workflow engine and expand its functionality by adding meta workflow support to it. In this architecture we extend an existing workflow enactment service with two modules: an event support module and a meta workflow module. The workflow engine is shown on the left of the figure with standard components. The components on the right are new additions.

The ESP support module allows users to specify rules. It registers events of interest with the workflow engine. These events could be standard events like start of an activity, end of an activity, etc., and also non-standard events unique to each workflow case (such as parts arrived late, special approval required, etc.). In addition, there will also be external events like cancellation of a running workflow instance. An event is accompanied by a case-id and case data. The case data will pertain to the current state of a running instance and also specific data values pertaining to the case. The ESP module will use this information to select the first applicable rule from its database, and the corresponding meta workflows to run. The ESP module passes the meta workflow and case information to the meta workflow (MWF) execution support module. The MWF support module manages the execution of the meta activities by sending them to the main workflow engine.

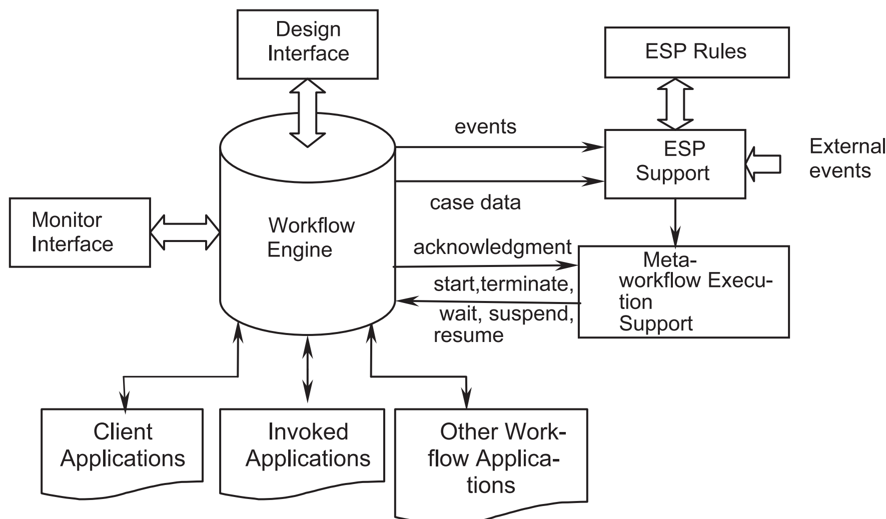

Fig. 8. An architecture for the ESP framework.

Clearly, to support this architecture, some modifications are required in an existing workflow engine. The main requirements are as follows:

- Allow registration of events of interest to ESP support module
- Send events and case data of running instances to the ESP support module
- Receive meta activities from the MWF support module and run them according to the correct semantics (see Section 2).

4.2. Expressive power of ESP

The meta activities were carefully chosen in an effort to strike a balance between usefulness of these primitives and giving users unconstrained power that may throw the workflow out of control. For example, there is no meta activity for raising an event. Thus, there is no way for a meta workflow to activate a new rule and start a new meta workflow. Because of this limitation the situation in which there is an infinite loop of event capture and event generation is impossible.

More important is the fact that it is not possible to start any activity of the base workflow, one can only suspend and possibly resume a suspended activity. This limitation is necessary in order to guarantee that some of the "good" properties of the base workflow are transferred to the resulting workflow. One such good property is termination—one would like a workflow to terminate correctly in all execution situations, instead of entering into a deadlocked state. Of course, if the situation requires that the workflow should not terminate, as in a cancellation example, one could use a meta activity of *terminate(base workflow)* which will stop the base workflow. But if only a temporary suspension of activities is needed, and if the ESP rules and meta workflows follow some simple guidelines, the temporary suspension will not cause the base workflow to go into a deadlock when resumed. The core of the guideline is that for each *suspend(wf,A)* or *wait and suspend(wf,A)* meta operation, a corresponding *resume(wf,A)* or *resume(wf,A+)* operation is executed in the meta workflow.

The proof is intuitive: *if a suspend(wf,A) is always followed eventually by a resume(wf,A) then from the base workflow's point of view, it is equivalent to activity A taking a longer time to execute*. But since by the assumption that the base workflow has been verified, and will not go into a deadlock state in any execution trace, it follows that in this execution as well in which A takes longer, it will also terminate correctly. The same argument is also valid for a *wait and suspend(wf,A)* and *resume(A+)* sequence.

The guidelines for designing meta workflows can be summarized as:

1. Verify the correctness of each base workflow. As we argued above, each base workflow is usually "simple," and for the simpler ones their correctness can be verified by inspection. A complex workflow can be checked using verification techniques (see Refs. [20,16,21]).
2. Given a set of meta workflows, verify that each *suspend(wf,A)* or *wait and suspend(wf,A)* meta activity is followed eventually by a corresponding *resume(wf,A)* or *resume(wf,A+)* meta activity. This verification may in fact be complex because typically each corresponding meta activity will be in a different meta workflow (see example 3.3), which will be started by different ESP rules.
3. Optionally, a *suspend(wf1)* may be eventually followed by a *terminate(wf1)*. This is a standard primitive for controlled termination of a workflow—first, all activities are suspended and then the workflow is terminated.

5. Discussion and related work

Recently, there has been growing interest in representing workflows in XML syntax for inter-operability. Some examples of this approach are XRL [19] and BPEL4WS [22]. The workflows and meta workflows described in this paper can easily be expressed in XML syntax. We have given an XML Schema description for the workflows and meta workflows in Appendices A and B. The XML Schema in Appendix A can be used to create XML workflows like the ones used in the examples in this paper. The main elements in this XML Schema are *sequence, parallel, choice* and *while_do*. XML is a structured language and it can be used to represent structured workflows like the ones in our earlier examples using matching start and

end tags. Thus, *start-parallel* and *end-parallel* constructs shown in Fig. 1 would be represented by ⟨*Parallel*⟩ and ⟨*/Parallel*⟩ tags, respectively. The ESP rules and meta workflows can be expressed using the XML Schema description of Appendix B.

5.1. Related research on exceptions and events

Support for events plays an important role in modeling of inter-organizational workflows because events offer a convenient mechanism for coordination. The need for such support has been noted elsewhere also (see Ref. [8] for an example). This paper has proposed to integrate events with meta workflows to create a powerful coordination and control mechanism. The usefulness of this approach was illustrated through various examples. Coordination requirements can be quite complex in inter-organizational workflows [19,6,7]. Our approach can have considerable value in the context of Web services [17] and supply chain applications [3]. In Web services, meta workflows can be used to facilitate inter-operation between multiple related services (e.g., airlines, hotel and car rental reservations) that must be integrated. In supply chain applications, exceptions like missed deliveries, stock-outs, etc. arise quite often. Here our framework can assist in reacting to such new events in a systematic manner and improving the level of collaborative information sharing between partners.

A prototype event engine, called EVE, for implementing event-driven execution of distributed workflows has been presented in [5]. However, the most relevant related work to ours is the event-based inter-process communication in the context of OPERA [8]. The mechanism described in Ref. [8] allows processes to communicate by means of event based control connectors (ECCs). An ECC is associated with an event and, upon occurrence of the event, if a condition is true then another process or an activity can be invoked. Our approach is similar in spirit to this work; however, we make a clear distinction between workflows and meta workflows, which is lacking in [8]. This separation allows for a more systematic and flexible methodology for process design, and nicer semantics. Thus, meta workflows serve as a useful modeling construct for controlling multiple workflows and dealing with various kinds of special (exception) situations that often arise, including failure handling, recovery, etc.

Research on exceptions in workflows is still limited. In Ref. [18], Strong and Miller define exceptions as "cases that computer systems cannot correctly process without manual intervention." Based on a field study they make several recommendations, such as the need for more efficient exception handling routines and better support for people who have to fix exceptions. Murata and Borgida [15] describe exceptions as violations of constraints and apply ideas from exception handling in programming languages. They treat an exception as an object belonging to a class that can have attributes. Their class structure is similar to a taxonomy. Another taxonomy-based approach to handling exceptions by Klein and Dellarocas is presented in [11]. They define an exception as "any departure from a process that achieves the process goals completely and with maximum efficiency." The authors propose to create a taxonomy based on the type of exception and have corresponding strategies to be applied once an exception can be classified. The approach in Ref. [4] is based on ECA style rules. These rules are bound to the workflow for exception handling at different levels of scope.

As noted in [8] also, the event-based framework has some similarities with the ECA style rules [14]. Clearly, both approaches are based on rules and events. However, the application environments and semantics are different. ECA rules are used in active databases in the context of changing data. ECA has also been proposed in the WIDE and EVE prototype workflow systems [1,5] as a means to describe the coordination requirements in a workflow itself and to handle simple exceptions. The major difference is that an action in ECA corresponds to a sub-transaction (such as updates to the database). In workflows, instead of the action a special process called a meta workflow is executed resulting in vastly different semantics. Therefore, an attempt to implement this framework in a database that supports ECA rules is likely to be impractical and awkward.

Recently there has also been interest in adaptive workflows [9,10]. *Adaptability* and *extensibility* are important features of our framework. Since all exceptions cannot be anticipated in advance, it is possible to add additional secondary base workflows and

corresponding meta workflows to react to new situations. Moreover, workflow modules developed for one application may be reused in another application. In the next subsection we briefly discuss how our work relates to web services and in particular to BPEL4WS.

5.2. Web services and BPEL4WS

There has been considerable interest in recent years in the design and implementation of Web services. Business Process Execution Language for Web Services (BPEL4WS) has emerged as a new standard for describing web services [22]. BPEL4WS gives a syntax for describing formal specification of business processes and business interaction protocols in XML. By doing so, it extends the Web Services interaction model and enables it to support business transactions. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to-business spaces. Naturally, a web service that spans multiple organizations, or even departments, should have a way for describing the workflow associated with it. Hence, workflows are integral to web services.

BPEL4WS provides a variety of constructs for describing a workflow, the complex coordination arrangements between various activities involved in it and also the flow of data between the activities. The present discussion of BPEL4WS is primarily limited to features that are relevant to support of exception handling and meta-level control. BPEL4WS has a double concept of faults and events, which would roughly correspond to our events. Faults are events that represent “errors” and thus have a standard solution: undo what has been done. To each activity and to segments of activities (each segment can be described as a scope) one can associate a compensating activity, so that in the case of a fault, if the activity is terminated, the system will call the compensating activity to undo it. This default behavior would be replicated in our framework by creating an appropriate meta workflow to call the corresponding compensating activities. However, the ESP framework takes a broader view of events than BPEL4WS in that events are not in general “errors,” but situations where complex reasoning is required.

Events in BPEL4WS are asynchronous notifications that need complex activities to be dealt with. One can define event handlers that are associated with a scope of the workflow: if an event happens during the execution of any activity in its scope, the execution of the associated handler starts in parallel with the already executing activities in the scope. The event handler can execute any (standard) activity, can *terminate* activities in progress and can generate new events and faults. Thus, by generating a fault an event handler can cause the effect of an activity within its scope that has already terminated to be *compensated*.

ESP has a somewhat similar semantics, that is, event handlers (in our case the meta workflows) run in parallel with the standard workflow. However, our meta workflows have more complex meta activities available to them: BPEL4WS only allows for *start* of new activities and the *termination* (unconditional, not compensable) of all activities within the current scope, and indirectly, through generation of a fault, the *compensation of* activities already terminated; ESP allows also for *suspension* and *resumption of* activities, and *waiting* for them to terminate.

Another feature that adds to the expressive power of ESP is the state description language. In BPEL4WS, an event handler is started if any of the activities within the current scope is still running. In terms of the ESP state description language, this means that BPEL4WS only allows for state descriptions that are a disjunction of expressions of the form $activity_i \in executing$, where $activity_i$ is in the current scope. Thus, in BPEL4WS it is not possible to write exception handling workflows that depend on the case data or on complex conditions regarding the state of the base workflow, especially if the base workflow has concurrently running branches.

Table 4 summarizes and compares the terminology and features of the two approaches. We perceive the main advantages of the ESP framework as being: (1) performing various actions based on the state of the workflow; and (2) the ability to exercise meta control on the base workflows. The strengths of BPEL4WS lie in formal notions of different types of handlers for events, compensation and faults. ESP does not make such a distinction and treats all handlers in the same way. A more detailed comparison between these two approaches through actual

Table 4
Comparison of main features of BPEL4WS and ESP framework

Issue	BPEL4WS	ESP Framework
Basic mechanism	Events, faults	Events
How are errors generated?	By activities—No logic for checking state	From combination of event occurrences and case data
How are exceptions handled?	Compensation handlers, fault handlers	Meta workflow + base workflow
Control of business process	No way to control running activities other than terminate	suspend, resume, wait and suspend meta activities allow fine control
Method for maintaining state	Correlation set	Case id
Scope of handlers	Handlers act based on their hierarchical scopes	No explicit notion of scope
Rethrowing of events	Can Rethrow events to a higher scope	Meta workflows do not raise events
Purpose	Handling faults and exceptions	Handing exceptions and also as a methodology
Other	Complex, heavy-weight mechanism	Simpler, light-weight framework

implementation of, say, the same application using the two approaches would be very useful, but is beyond the scope of this work.

5.3. ESP as a methodology

ESP can also be seen as a methodology to develop workflow models. The normal case is developed as a base workflow, and as exceptional cases are discovered they are added as other base workflows, metaworkflows and ESP rules. In fact, the example of workflow modification (see Section 3.2) in this paper is more of an example of incremental development of a workflow than an example of exception handling. When the ‘‘normal’’ case was thought out, the developer did not take into consideration the ‘‘exception’’ case of laptops costing more than $3000. As the workflow is put to use, this new situation is discovered, and the appropriate *incremental* modification is added to the system. The original workflow is kept unchanged, and the modifications are added as new ESP rules, meta- and base workflows. A major advantage is that a working workflow does not need to be modified with optional control structures which may introduce new errors. Our approach allows the workflow to evolve and the framework provides flexibility and support for such evolution, leading to adaptable workflows.

6. Conclusions

This paper described a framework and architecture for support of exception handling in business workflows. It is based on detecting event-state combinations that cause higher level processes called meta workflows to be executed. A meta-process consists of five meta activities for controlling and coordinating base workflow processes. We demonstrated the usefulness of the approach and gave an architecture for integrating this approach into a current workflow system. The advantages of this framework are modularity, extensibility and adaptability.

We foresee further research along several lines. First, we already have a prototype implementation of XRL (eXchangeable Routing Language) technology [19] and are planning to use this as an experimental test bed for adding the functionality proposed in this paper for meta workflows. Such a test bed would allow us to evaluate the proposal from a performance standpoint. Secondly, although we gave the semantics for execution of meta workflows themselves, rule conflicts were ignored since we assumed that the rules are ordered by priority and the first applicable rule fires. Future work could address this by developing detailed rule execution semantics. Thirdly, support for composite events would be a useful feature. At present each event is processed as soon as it occurs, and if no matching ESP rule is found it is discarded. Support for composite events would allow for action to be taken on multiple events that are related to one another.

Acknowledgements

This research was supported in part by a grant from the IBM Corporation through the Center for Supply Chain Research at Penn State University.

Appendix A. XML schema definition of base workflow (WFlow.xsd)

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.xrl.org"
            xmlns="http://www.xrl.org"
            elementFormDefault="qualified">
  <xsd:element name="start">
    <xsd:complexType>
      <xsd:choice minOccurs="1" maxOccurs="1">
        <xsd:element ref="activity" />
        <xsd:element ref="sequence" />
        <xsd:element ref="parallel" />
        <xsd:element ref="choice" />
        <xsd:element ref="while_do" />
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
<xsd:element name = "sequence" >
    <xsd:complexType>
      <xsd:choice minOccurs="1" maxOccurs="unbounded">
        <xsd:element ref="activity" />
        <xsd:element ref="sequence" />
        <xsd:element ref="parallel" />
        <xsd:element ref="choice" />
        <xsd:element ref="while_do"/>
      </xsd:choice>
    </xsd:complexType>
</xsd:element>
<xsd:element name="activity" >
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="event" minOccurs="1" maxOccurs="unbounded"/>
     </xsd:sequence>
      <xsd:attribute name = "name" type="xsd:string"/>
      <xsd:attribute name = "adresss" type="xsd:string"/>
      <xsd:attribute name = "role" type="xsd:string"/>
      <xsd:attribute name = "doc_read" type="xsd:string"/>
      <xsd:attribute name = "doc_result" type="xsd:string"/>
      <xsd:attribute name = "status">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="not_ready"/>
            <xsd:enumeration value="ready"/>
            <xsd:enumeration value="executing"/>
            <xsd:enumeration value="done"/>
            <xsd:enumeration value="suspended"/>
            <xsd:enumeration value="aborted"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name = "start_time" type="xsd:string"/>
```

```
        <xsd:attribute name = "end_time" type="xsd:string" />
      </xsd:complexType>
</xsd:element>
<xsd:element name = "parallel" >
   <xsd:complexType>
     <xsd:choice  minOccurs="1" maxOccurs="unbounded" >
       <xsd:element ref="sequence" />
       <xsd:element ref="parallel" />
       <xsd:element ref="choice" />
       <xsd:element ref="while_do" />
     </xsd:choice>
   </xsd:complexType>
</xsd:element>
<xsd:element name = "choice" >
   <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref=" choice_true"  />
        <xsd:element ref=" choice_false" />
      </xsd:sequence>
      <xsd:attribute name = "condition" type="xsd:string" />
   </xsd:complexType>
</xsd:element>
<xsd:element name = "choice_true" >
       <xsd:complexType>
          <xsd:choice minOccurs="1" maxOccurs="unbounded" >
            <xsd:element ref="sequence" />
            <xsd:element ref="parallel" />
            <xsd:element ref="choice" />
            <xsd:element ref="while_do" />
          </xsd:choice>
       </xsd:complexType>
</xsd:element>
<xsd:element name = "choice_false" >
       <xsd:complexType>
          <xsd:choice minOccurs="1" maxOccurs="unbounded" >
             <xsd:element ref="sequence" />
             <xsd:element ref="parallel" />
             <xsd:element ref="choice" />
             <xsd:element ref="while_do" />
          </xsd:choice>
       </xsd:complexType>
</xsd:element>
<xsd:element name = "while_do" >
       <xsd:complexType>
          <xsd:choice minOccurs="1" maxOccurs="unbounded" >
            <xsd:element ref="sequence" />
            <xsd:element ref="parallel" />
            <xsd:element ref="choice" />
            <xsd:element ref="while_do" />
          </xsd:choice>
          <xsd:attribute name = "condition" type="xsd:string">
       </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Appendix B. XML schema definition of ESP rules and meta workflows (MWFlow.xsd)

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.xrl.org"
           xmlns="http://www.xrl.org
           elementFormDefault="qualified">
<xsd:element name="esp">
    <xsd:complexType>
        <xsd:sequence>
           <xsd:element ref="event" />
           <xsd:element ref="state" />
           <xsd:element ref="meta_wf" />
        </xsd:sequence>
     </xsd:complexType>
</xsd:element>
<xsd:element name="event">
    <xsd:attribute name = "event_id" type="xsd:string" />
</xsd:element>
<xsd:element name="state">
    <xsd:attribute name = "condition" type="xsd:string" />
</xsd:element>
<xsd:element name="meta_wf">
    <xsd:complexType>
       <xsd: choice minOccurs="1" maxOccurs="unbounded" >
          <xsd:element ref="start" />
          <xsd:element ref="suspend" />
          <xsd:element ref="resume" />
          <xsd:element ref="wait_and_suspend" />
          <xsd:element ref="terminate" />
       </xsd:choice>
    </xsd:complexType>
</xsd:element>
<xsd:element name="start">
    <xsd:attribute name = "wflow_name" type="xsd:string" />
    <xsd:attribute name = "activity_name" type="xsd:string" />
</xsd:element>
<xsd:element name="suspend">
    <xsd:attribute name = "wflow_name" type="xsd:string" />
    <xsd:attribute name = "activity_name" type="xsd:string" />
</xsd:element>
<xsd:element name="resume">
    <xsd:attribute name = "wflow_name" type="xsd:string" />
    <xsd:attribute name = "activity_name" type="xsd:string" />
</xsd:element>
<xsd:element name="wait_and_suspend">
    <xsd:attribute name = "wflow_name" type="xsd:string" />
    <xsd:attribute name = "activity_name" type="xsd:string" />
</xsd:element>
<xsd:element name="terminate">
    <xsd:attribute name = "wflow_name" type="xsd:string" />
    <xsd:attribute name = "activity_name" type="xsd:string" />
</xsd:element>
</xsd:schema>
```

References

[1] F. Casati, S. Ceri, S. Paraboschi, S. Pozzi, Specification and implementation of exceptions in workflow management systems, ACM Transactions on Database Systems 24 (3) (1999 Sept.) 405–451.

[2] V. Christophides, R. Hull, A. Kumar, Querying and splicing of XML workflows, CoopIS, (2001) 386–402.

[3] T. Curran, A. Ladd, SAP R/3: Understanding Enterprise Supply Chain Management, Prentice-Hall, Upper Saddle River, NJ, 2000.

[4] K.W.C. Dickson, Q. Li, K. Karlapalem, A meta modeling approach to workflow management systems supporting exception handling, Information Systems 24 (2) (1999) 159–184.

[5] A. Geppert, D. Tombros, Event-based distributed workflow execution with EVE, Technical Report 96.5, University of Zurich, 1996.

[6] P. Grefen, K. Aberer, Y. Hoffner, H. Ludwig, Crossflow: cross-organizational workflow management in dynamic virtual enterprises, International Journal of Computer Systems Science and Engineering vol. 15 (5),CRL Publishing, London, 2000 (September), pp. 277–290.

[7] B. Gronemann, G. Joeris, S. Scheil, M. Steinfort, H. Wache, Supporting cross-organizational engineering processes by distributed collaborative workflow management—The MOKASSIN approach, Proc. of 2nd Symposium on Concurrent Mulitdisciplinary Engineering (CME'99), Bremen, Germany, 1999.

[8] C. Hagen, G. Alonso, Exception handling in workflow management systems, IEEE Transactions on Software Engineering 26 (10) (2000) 943–958.

[9] Y. Han, A. Sheth, C. Bussler, A taxonomy of adaptive workflow management, Workshop on Adaptive Workflow Systems, ACM Conference on Computer Supported Cooperative Work, Seattle, Washington, USA, 1998.

[10] G. Joeris, O. Herzog, Managing evolving workflow specifications, 3rd IFCIS Intl. Conf. on Cooperative Information Systems (CoopIS'98), 1998, pp. 310–319.

[11] M. Klein, C. Dellarocas, A knowledge-based approach to handling exceptions in workflow systems, Computer Supported Cooperative Work (CSCW) 9 (3/4) (2000) 399–412.

[12] A. Lazcano, G. Alonso, H. Schuldt, C. Schuler, The WISE approach to electronic commerce, Computer Systems Science and Engineering 15 (5) (2000) 345–357.

[13] H. Lee, S. Whang, Information Sharing in Supply Chains, Stanford Graduate School of Business, Research paper (1998) 1549.

[14] D.R. McCarthy, U. Dayal, The architecture of an active database system, Proc. ACM SIGMOD Conf. on Management of Data, Portland, 1989, pp. 215–224.

[15] T. Murata, A. Borgida, Handling of irregularities in human centered systems: a unified framework for data and processes, IEEE Transactions on Software Engineering 26 (10) (2000 October) 959–977.

[16] W. Sadiq, M.E. Orlowska, Analyzing process models using graph reduction techniques, Information Systems 25 (2) (2000 June) 117–134 (Elsevier).

[17] M. Sayal, F. Casati, U. Dayal, M. Shan, Integrating workflow management systems with business-to-business interaction standards, Proceedings of ICDE 2002, San Jose, California, 2002 (February).

[18] D. Strong, D. Miller, S. Miller, Exceptions and exception handling in computerized information processes, ACM Transactions on Information Systems 13 (2) (1995) 206–233.

[19] W.M.P. van der Aalst, A. Kumar, XML based schema definition for support of organizational workflow, Information Systems Research 14 (1) (2003 March) 23–46.

[20] W.M.P. van der Aalst, A.H.M. ter Hofstede, Verification of workflow task structures: a Petri-net based approach, Information Systems 25 (1) (2000) 43–69.

[21] H.M.W. Verbeek, T. Basten, W.M.P. van der Aalst, Diagnosing workflow processes using Woflan, Computer Journal 44 (4) (2001) 46–279.

[22] Web Services, BPEL specification version 1.1, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel.

Akhil Kumar is a professor of Information Systems at the Smeal College of Business at Penn State University. He has a Ph.D. from Berkeley and has published about 60 papers in the areas of database systems and structures, data replication, machine learning and workflow systems in leading academic journals, and international conferences and workshops. He serves on the editorial boards for three information technology academic journals. His current research interests are in design, analysis and verification of workflow processes, design and implementation of e-services, and web-based technologies.

Jacques Wainer is an associate professor at the Institute of Computing in the State University of Campinas (UNICAMP), Brazil. His academic interests and publications are in the areas of collaborative computing, artificial intelligence, and medical informatics. He is also a visiting professor at the Department of Medical Informatics at the Federal University of Sao Paulo (UNIFESP). Dr. Wainer has consulted for many companies in the area of workflow systems and artificial intelligence.