BPEL4Job: A Fault-Handling Design for Job Flow Management

Wei Tan[1,*], Liana Fong[2], and Norman Bobroff[2]

[1] Department of Automation, Tsinghua University, Beijing 100084, China
[2] IBM T. J. Watson Research Center, Hawthorne, NY 10532, USA
tanwei@mails.tsinghua.edu.cn, llfong@us.ibm.com, bobroff@us.ibm.com

Abstract. Workflow technology is an emerging paradigm for systematic modeling and orchestration of job flow for enterprise and scientific applications. This paper introduces BPEL4Job, a BPEL-based design for fault handling of job flow in a distributed computing environment. The features of the proposed design include: a two-stage approach for job flow modeling that separates base flow structure from fault-handling policy, a generic job proxy that isolates the interaction complexity between the flow engine and the job scheduler, and a method for migrating flow instances between different flow engines for fault handling in a distributed system. An implementation of the design based on a set of industrial products from IBM is presented and validated using a Montage application.

1 Introduction

Originating from the people-oriented business process area, the applicability of workflow technology today is increasingly broad, extending to inter and intra organizational business-to-business interactions, automatic transactional flow, etc [1]. With the advent of web services as a new application-building paradigm in a loosely-coupled, platform-independent and standardized manner, the use of workflow to orchestrate the invocation of web services is gaining importance. The Web Service Business Process Execution Language [2] (WS-BPEL or BPEL for short), proposed by OASIS as a standard for workflow orchestration, will enhance the inter-operability of workflow in distributed and heterogeneous systems. Although many custom workflow systems have been developed by the scientific application community [3-5], the inter-operability of BPEL workflow systems has attracted many researchers [1, 6-10] to experiment with BPEL for applications in distributed environments such as grid.

BPEL-based workflow is particularly relevant in orchestrating batch jobs for enterprise applications, as job flow is an integral part of the business operation. There are obvious advantages in standardizing on a common flow language, such as BPEL, for both business process and batch jobs. Although some workflow systems are used for enterprise applications [11, 12], these workflow systems use proprietary flow languages.

[*] The work was done while the author was on an internship at IBM T.J. Watson Research Center, NY, USA.

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 27–42, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The use of BPEL for job flow is not without technical challenges, as BPEL was not designed with job flow requirements. These challenges include defining a job[1] entity within BPEL, expressing data dependency (usually implicitly expressed in the job definition), and passing of large data between jobs. Another key challenge is to manage the predominately asynchronous interaction between the BPEL engine and the job scheduling partners. Finally, support for fault tolerance and recovery strategy is important due to the long-running nature of jobs, as well as the interaction of grid services with dynamic resources [13]. This paper addresses the latter issues of asynchronous interactions and fault handling in job flow by proposing a design called BPEL4Job.

BPEL4Job includes three unique features. First, a two-stage approach for job flow modeling is presented. In stage one the flow structure and fault-handling policies are modeled separately. Stage two combines and transforms the flow model and policy into an expanded flow that is then orchestrated by a BPEL-compliant engine. The advantage of this approach is that it separates the concerns of application flow modeling from fault handling. Second, a generic job proxy is inserted between the BPEL engine and the job scheduler to facilitate job submission and isolate the flow engine from the asynchronous nature of status notification, including fault events. Finally, we propose several schemes for flow-level fault handling, including a novel method for instance migration between flow engines. Instance migration is important for scalable failure recovery in a distributed environment. For example, a flow that fails due to resource unavailability may be migrated to another resource domain.

The design and implementation work in this paper is based on the IBM BPEL-compliant workflow modeler and execution engine, as well as the service oriented job scheduler.

The following section introduces BPEL4Job, the overall design approach to incorporating fault handing features into the BPEL design and execution process. Section 3 discusses integrating fault policies at the flow's design stage. Section 4 presents the fault handling scheme and especially, the technique for flow instance migration and flow re-submission. Section 5 introduces our prototype system, and demonstrates our fault handling method using the Montage application [14]. Section 6 surveys related work and Section 7 concludes the paper and suggests future directions.

2 BPEL4Job: A Fault-Handling Design for Job Flow Management

In this section, we introduce our overall design, BPEL4Job, which facilitates the advanced fault handing in BPEL both the flow modeling tools and execution environments. More specifically, BPEL4Job has the following unique features:

- Adding a flexible fault handling approach based on policies. These policies can express a range of actions from simple job retry, to how and at what point in the flow to restart for a particular type of execution failure. The policies allow options to clean or retain the state of the jobs flow in the flow engine database.

[1] The terms "job" and "job step", and "job flow" and "flow" are used interchangeably in this paper. A job flow consists of one or more jobs.

- Introducing a functional element called a 'job proxy' that connects and integrates the high level BPEL engine with the lower level job scheduler that accepts and executes jobs. The proxy captures the job status notifications from the scheduler and relays them to the BPEL engine. The proxy serves as an arbiter and filter of asynchronous events between the BPEL engine and the job scheduler.
- Supporting migration of the persisted state of a BPEL job flow to another engine. This capability provides fault tolerance by allowing a flow that has failed, for example, because of resource exhaustion in one environment to continue execution in another environment.

The design of BPEL4Job consists of three layers: the *flow modeling* layer, the *flow execution* layer and the *job scheduling* layer, as shown in Fig. 1. First, we describe the *flow modeling* layer. The flow modeling in BPEL4Job takes a two-stage approach in modeling job flow. In the first stage, the *base flow,* the *job definitions,* and the *fault-handling policies* are defined. The base flow is a BPEL expression of the control flow of jobs for a process or an application. Each job definition describes a unit of work (e.g. an executable file together with parameters and resource requirements) to be submitted to scheduler and is expressed by a markup language such as Job Submission Description Language (JSDL) [15]. The fault-handling policies define the actions to be taken in case of job failures and can be described using the web service policy language WS-Policy [16]. In the second stage, the base flow, job definitions, and fault-handling policies are transformed into an expanded flow that is an executable BPEL process. This two-stage modeling approach has many advantages. First, the flow designer defines the job flow structure and fault-handling

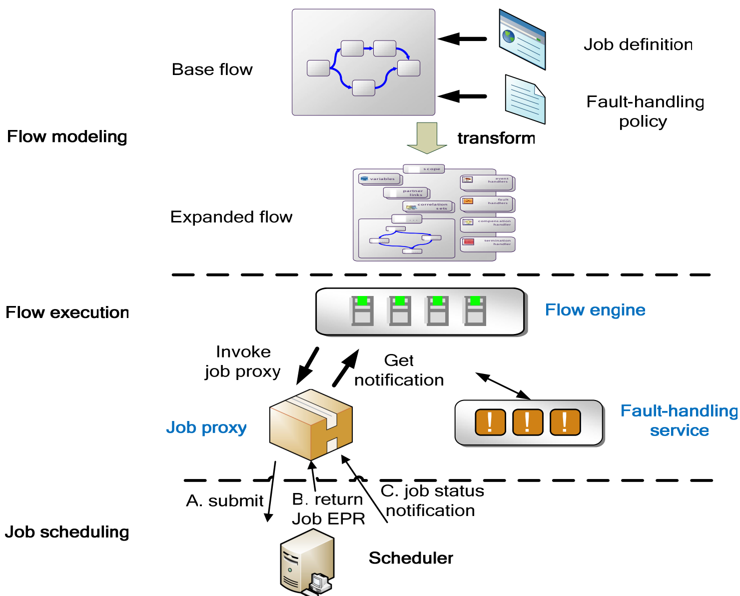


Fig. 1. BPEL4Job: fault-handling design for job flow management

policies separately, and needs not be concerned on how to implement these policies in BPEL. Second, the base flow and policies can be reused and combined if necessary. More details and examples are provided in Section 3.

The flow execution layer consists of three major components: the flow engine, the job proxy, and the fault-handling service. The flow engine executes the expanded BPEL originating in the flow modeling layer. For each job step in the expanded flow, the job proxy is invoked by the flow engine. The job proxy submits the job definition to the scheduler, listens for job status notification, and reports job success or failure to the flow engine. In the case of job failure, the flow engine invokes the fault-handling service if necessary. Otherwise, if successful, the flow engine proceeds to the next job step. The fault-handling service is discussed in Section 4.

The job-scheduling layer accepts jobs, returns a unique end-point reference (EPR) for each job, and sends notification on job status changes. We assume that the schedulers are responsible for resources matching and job execution management. Some schedulers also implement failure recovery techniques such as re-try. In BPEL4Job, we supplement this capability with a set of fault-handling techniques at the flow execution layer including re-try from another job step, as well as flow instance migration to other engines.

3 Integrating Fault-Handling Policies with Job Flow Modeling

Yu et al. [5] and Hwang et al. [17] classified the fault-handling methods of grid workflow into two levels: task level and flow level. From their work, we observe that, re-try and re-submit are the most elementary methods in these two levels respectively. Second, while several approaches [5, 18] have been proposed to deal with the task level re-try, the issue of flow level re-submit is still challenging. In this section, we provide a set of schemes to address fault-handling at both task and flow levels and to put emphasis on flow level.

BPEL4Job design considers three kinds of policies: cleanup policy, re-try policy and re-submit policy. These policies leverage the persistent flow states storage in most of the BPEL engines. Cleanup policy refers to generate fault report and delete the instance data in flow engine. Re-try technique refers to execute the same task again in case of failure. Re-submit technique refers to, in case of failure, the state of flow instance being exported from the flow engine, and restored to the same or a different engine, such that the flow can resume from the failed step without re-execution of completed steps. Other fault-handling policies such as using alternative resources, or rollback, can be built from these three fundamental ones.

As described in Section 2, our design of BPEL4Job has a two-stage approach for job flow modeling. The first stage models the flow structure and fault-handling policy separately. The second stage combines and transforms the flow model and policy into an expanded flow that is then orchestrated by an existing BPEL engine in the flow execution layer.

We now explain how the fault-handling policies are defined and integrated with the *base* flow to produce the *expanded* BPEL flow. Fig. 2 shows two exemplary fault-handling policies and a BPEL skeleton of a base flow. The first policy, named *retry-policy*, specifies that when job failure occurs, the flow will re-try from the current job step (by setting the value of element *RetryEntry* to *itself*), and after an interval of 300

seconds (by setting the value of element *RetryTimes* to *Unlimited*, and *RetryInterval* to *300s*). The second policy, named *resubmit-policy*, specifies that when job failure occurs, the flow will resume at another flow engine if desired. When it resumes, it restarts from the previous step of the failed job (by setting the value of element *RescueEntry* to *previous-step*. The base flow consists of two sequential job steps, *SubmitJob1* and *SubmitJob2*. In the base flow, the *retry-policy* is linked to *SubmitJob1* (<bpws:invoke name="SubmitJob1" faultHandling:policy="retry-policy" />), and *resubmit-policy* linked to *SubmitJob2* (<bpws:invoke name="SubmitJob2" faultHandling:policy= "resubmit-policy" />).

The re-try policy of *SubmitJob1* is realized by transforming the base flow to the expanded flow as shown in Fig. 3, and described as follows:

- Add a variable *RETRY* to indicate whether the job should be retried and set its value to *TRUE* before the job.
- Add an assign activity after the job to set variable *RETRY* to *FALSE*.
- Add a scope enclosing the job and succeeding assign activity.
- Add a While loop on top of the newly-added scope, and set the condition for the While loop to (*RETRY* == *TRUE*).
- Add a fault handler for the newly added scope to catch the fault. Advanced re-try schemes, including re-try for a given times, re-try after a given time of period, and re-try from a previous job, could all be implemented in this fault-handler block.

In case of job failure, the control flow goes to the fault handler (the *Catch All* block in Fig. 3), and when the fault-handling block completes, the control flow proceeds to the beginning of the While loop. Because the newly added scope does not complete when failure occurs, the value of variable *RETRY* is still *TRUE*, so the flow will continue at the beginning of the While loop (*Submit Job1* in Fig. 2), by this means the re-try policy is realized. It is important to note that expanded flow contains all the necessary fault-handling blocks, unlike other approaches in supporting runtime fault-handling selection [18].

```
<?xml version="1.0" encoding="UTF-8" ?>
 <wsp:Policy xmlns:wsp="..." xmlns:jobFlow="..."
               name="retry-policy">
  <jobFlow:Retry wsp:Usage="wsp:Required">
   <jobFlow:RetryEntry>self</jobFlow:RetryEntry>
   <jobFlow:RetryTimes>Unlimited</jobFlow:RetryTimes>
   <jobFlow:RetryInterval>300s</jobFlow:RetryInterval>
   </jobFlow:Retry>
  </wsp:Policy>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsp:Policy xmlns:wsp="..." xmlns:jobFlow="..."
               name="resubmit-policy">
 <jobFlow:Rescue wsp:Usage="wsp:Required">
 <jobFlow:RescueEntry>previous-step?
  </jobFlow:RescueEntry>
</jobFlow:Rescue>
</wsp:Policy>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
 <bpws:process xmlns:bpws="..." xmlns:faultHandling="...">
   <bpws:partnerLinks>...</bpws:partnerLinks>
   <bpws:variables>...</bpws:variables>
    <bpws:sequence name="HiddenSequence">
     <bpws:receive createInstance="yes" name="ReceiveJobRequest" />
     <bpws:invoke name="SubmitJob1" faultHandling:policy="retry-policy" />
     <bpws:invoke name="SubmitJob2" faultHandling:policy="resubmit-policy" />
     <bpws:reply name="Reply" />
    </bpws:sequence>
</bpws:process>
```

Fig. 2. The re-try and re-submit policy, and the base flow embedded with these policies

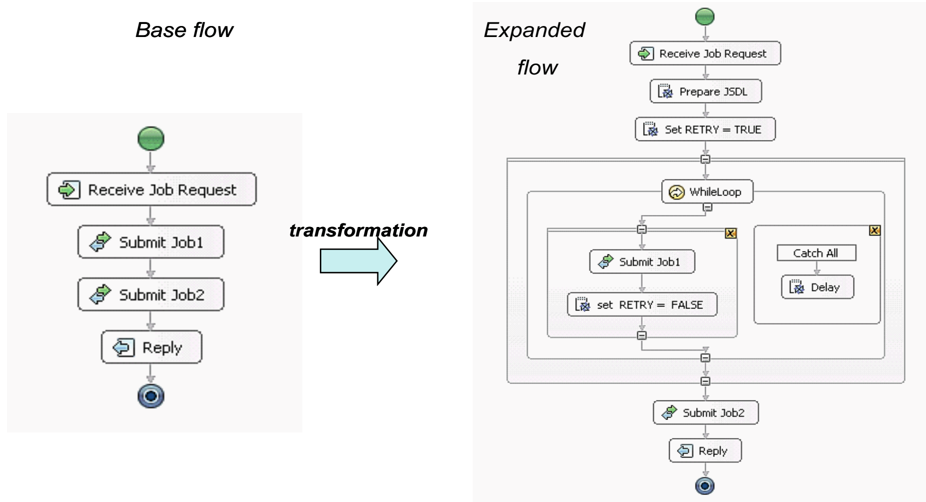


Fig. 3. The transformation to implement the re-try policy of *Job1*

4 Fault-Handling at the Flow Execution Layer in BPEL4Job

Job execution may fail due to a variety of reasons, such as resource and data unavailability, application failure, scheduler or human input error, etc. The fault handling at flow execution layer needs two mechanisms: the capability to recognize various job failures and the capability to handle the failures according to the policies defined at flow modeling layer.

In BPEL, faults can be raised by an invoked service and be caught by the invoking service. BPEL also provides a Java-style support for fault handling, using constructs like *Catch*, *Catch All*, *Throw*, *Rethrow*, etc. A BPEL fault handler catches faults and can handle them by, for example, calling a suitable fault-handling service. In addition, most of BPEL engines store persistent states of the flow and the use of states can support resumption of flow execution from a failed task. The design of fault handling in BPEL4Job would leverage the BPEL basic fault-handling features and enhance specific capabilities to recognize job failures and to handle faults according to defined policies. The following section addresses both aspects by introducing: i) the generic job proxy for job submission and job status notification (especially for fault recognition), and ii) the fault-handling schemes for various policies at the task level and flow level.

4.1 The Generic Job Proxy

The generic job proxy connects and integrates the higher-level BPEL workflow engine with the lower-level job scheduler. For each job submission invocation, the proxy submits jobs, captures the job status notifications from the scheduler, and returns the job failure/success result in a synchronous manner. It serves as an arbiter and filter of asynchronous notification events of jobs. When a job fails, the job proxy

raises a fault to the workflow engine. Then, the workflow engine would invoke fault-handling service after catching the fault.

Fig. 4 shows the control flow of a generic job proxy. The explanation is as follows:

1. Receive a job submission request.
2. Forward the job request to a scheduler, and start to listen for the job state notification from it. The state notifications from different schedulers may vary, but usually they include *Submitted*, *Waiting_For_Resources*, *Resource_Allocation_Received*, *Resource_Allocation_Failed*, *Executing*, *Failed_Execution*, *Succeeded_Execution*, etc.
3. When state notifications come, filter the states. For states indicating the success/failure of job comes, forward this information to flow engine and returns, otherwise continue listening for the notification.

The job proxy provides a compact job-submission interface to the flow engine, so that for each job the flow engine does not need to use two separate activities to submit job and query job status respectively. The function of job proxy is not limit to fault handling, and it is actually a single entrance for job schedulers and can handle the complexity stemmed from the heterogeneity of different schedulers.

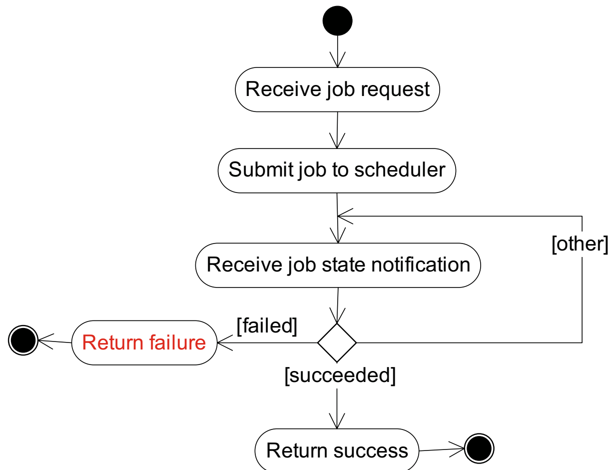


Fig. 4. Control flow of the generic job proxy

4.2 Fault-Handling Schemes in BPEL4Job

The fault-handling logical schemes of BPEL4Job are illustrated in Fig. 5, though the design is not limit to these policy schemes. When a job step is in state *Ready*, the flow engine submits it (*Submit Job*) and listens for the notification from the job proxy (*submitted*). If the job *succeeds*, flow engine *navigates to next job* and the flow proceeds. If the job *fails*, flow engine reacts according to the fault-handling policy for that job. If the policy is *cleanup*, the fault report is generated and flow instance is deleted in flow engine database. If the policy is *re-try*, the engine find the *re-try entry*

(the re-try entry is the point to re-try a single job step, it can be at current failed job step, or at some previous step which has already completed) and submit the job to the scheduler. If the policy is *re-submit*, flow engine suspends the current flow instance, export the instance data to a permanent storage (for example, to a XML or other portable formats), and delete the instance data in current flow engine database. The exported flow instance can be re-submitted to the original engine when the source of the fault has been fixed, or be re-submitted to another flow engine to resume. After the flow instance is imported to the flow engine (either the original one or a new one), the flow instance is resumed at the *re-submit entry* (similar to the re-try entry, the re-submit entry is the point to re-start a job flow, it can be at the failed job step, or at some previous step which has already completed).

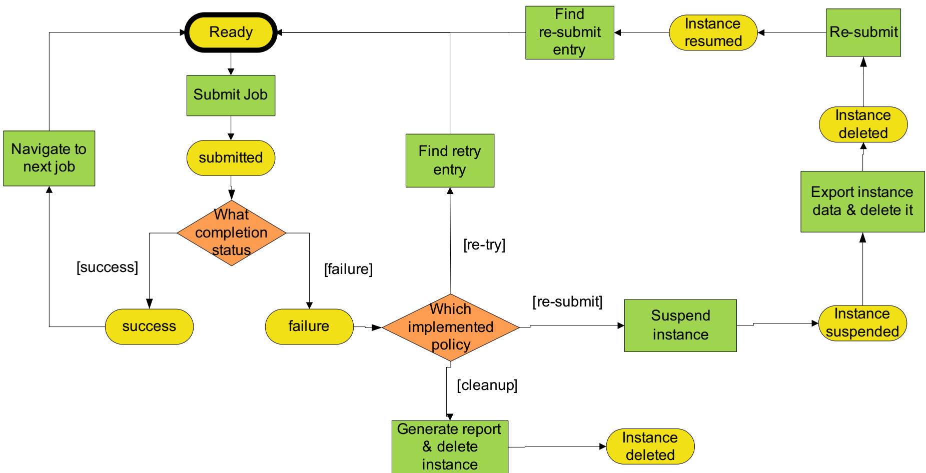


Fig. 5. Fault-handling scheme in BPEL4Job

4.2.1 Cleanup

Cleanup policy is used when the flow execution does not have any side effect resulted from failure, the user may just want to get the failure report and terminate the flow. Therefore, after the failure report is generated, the flow instance can be deleted (cleanup) from the flow engine database.

4.2.2 Task Level Re-try

We have shown the realization of a re-try policy as an example in section 3 where we explain how to integrate policy with job flow. The re-try policy is accomplished by adding a scope, a While loop and other additional constructs. Re-try policy can be extended to more advanced schemes, for example, to alter input parameters for the re-try job such as instructing the job proxy to use alternative schedulers or resources.

4.2.3 Flow Re-submission and Instance Migration

Now we investigate BPEL's capability to continue un-executed job steps without re-execution of successful job steps of a flow in the event of a fault. Many other job flow

systems support restarting a flow regardless whether or not they persist job state during execution. Here are two of the exemplary systems:

1. DAGMan [3] is the flow manager of Condor [19] jobs. While executing, DAGMan keeps in memory a list of job steps of the flow, their parent-child relationships, and their current states. When a flow fails, it produces a Rescue DAG file for re-submission with a list of the job steps along their states and reasons of failures. The Rescue DAG can then be submitted later to continue execution.
2. Platform LSF [20] supports job dependency and flow restarting with the "requeue" feature. In LSF, job steps are executed sequentially unless they have a conditional statement on the success of failure or preceding steps. If "requeue" is specified for a job flow, for example "REQUEUE_EXIT_VALUES = 99 100", the flow will be requeued if the return code of a step matches the *requeue_exit* criteria and the requeued job flow will restart from this particular step.

BPEL4Job supports re-submit and facilitates instance migration if desire. The motivation to do job flow re-submission and instance migration is two-fold. The first reason is the performance issue. For long-running job flows, flow instance data is stored in the flow engine's database. This instance data include instance state information, the navigated activities, the value of messages/variables, etc. Depending on the flow definition and the run-time data used in the instance, a relatively large amount of data can be created with each instance. Unlike business processes, scientists may submit job flows in very large numbers and may not return to handle the flows immediately. A strategy for removing the failed flow instance out of the database is desirable to lessen the burden on the data storage or database.

The second reason is for job flow re-submission to a different engine. When a job flow instance *f* fails during the execution, the flow user or administrator may find that resource needed for *f* to proceed is unavailable in current resource domain. Thus, an alternative is to export and delete *f* in current flow engine, choose another resource domain in grid environments, re-submit *f* to the flow engine in that domain and resume it. (See Fig. 6 for an example.)

In order to realize flow re-submission, we introduce the concept of *instance migration*. Instance migration refers to the technique to export job flow instance data in one flow engine, and import it into anther one so that the flow instance can resume in it. When we do instance migration, the challenge is to collect sufficient data from the source flow engine, so that the target engine could re-build the status of the on-going job flow. The job flow instance database schemas vary with the different implementation, and in Fig. 7 we give a conceptual and high-level flow instance data model. Next section presents our implementation based on IBM Webshpere Process Server [21].

In Fig. 7, a process instance (or flow instance) has an attribute named *ProcessInstanceID*, and an attribute *ProcessTemplateID* to refer to the process template it belongs. A process instance can consist of multiple activity instances, task instances, correlation set instances, scope instances, partnerlink instances, variable instances, etc. Each of these instances has an attribute *ProcessInstanceID* to refer to the process instance it belongs.

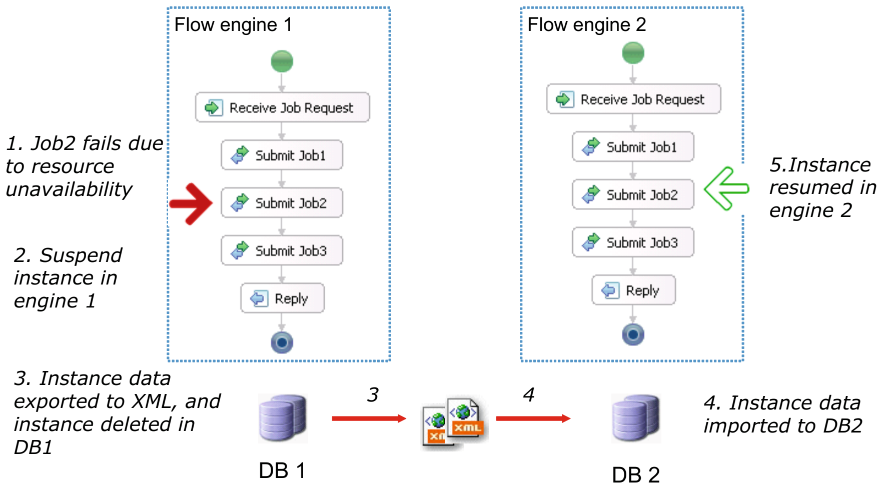


Fig. 6. An illustration of instance migration and flow re-submission

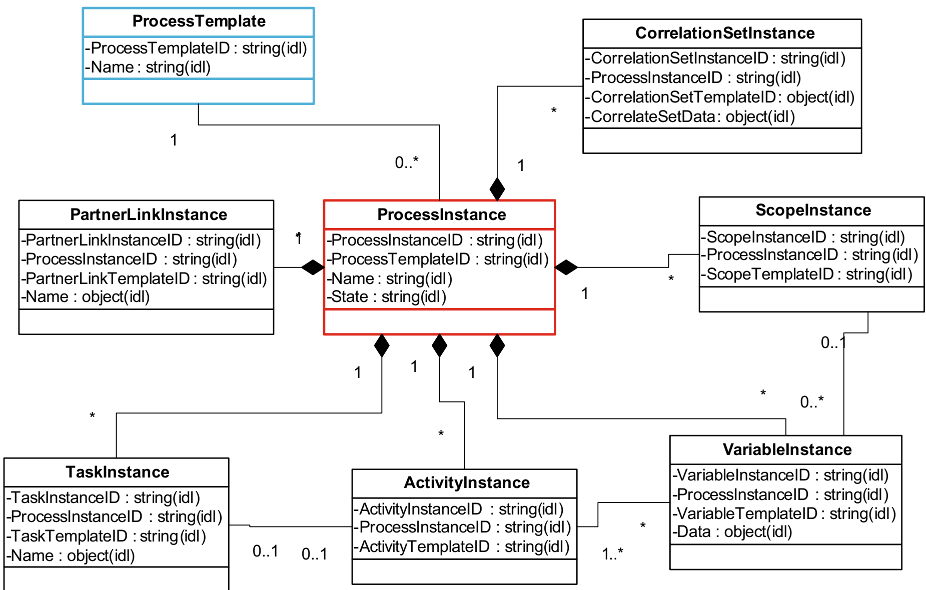


Fig. 7. Class diagram of flow instance data model

5 System Implementation and Case Study

A system is developed to validate the design of BPEL4Job. In our implementation, IBM Websphere Integration Developer (WID) [22] is used as BPEL modeling tool, IBM Websphere Process Server (WPS) [21] as BPEL engine, and IBM Tivoli Dynamic Workload Broker (ITDWB) [23] as job scheduler. In flow modeling layer, a

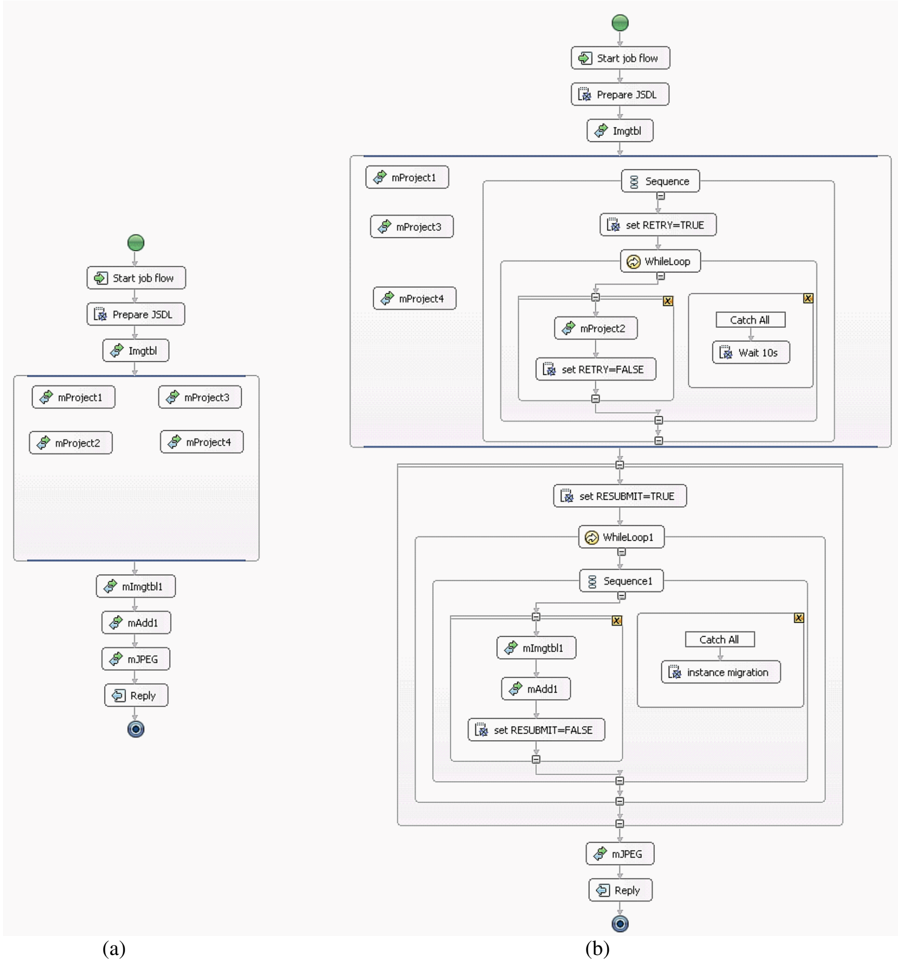
WID plug-in is developed to facilitate the use of JSDL for job step definition and the use of WS-Policy for policy definition. In flow execution layer, a generic job proxy is devised, and a fault-handling service is developed to implement the fault-handling schemes proposed in Section 4. For the job scheduling layer, we use ITDWB which provide job management web service API including job submission and job status notification.

We take an example from Montage astronomy mosaic generation application [14], named *m101 Mosaic*, to demonstrate the implementation of BPEL4Job. This example application takes several raw images (we use four images in our exemplary job flow), reprojects them and then combines them into a mosaic. We model the procedure of this application into a BPEL-based job flow (Fig. 8(a)). The first job, *mImgtbl*, generates an image metadata table describing the content of all the four raw images. Followed are four parallel jobs (*mProject1*, *mProject2*, *mProject3*, and *mProject4*), each of which reprojects one image. After all the images have been reprojected, a new metadata table is generated by job *mImgtbl1*, then job *mAdd1* generates a mosaic from the reprojected images, and finally job *mJPEG* transforms the mosaic into jpeg format.

Then we define fault-handling policies for job *mProject2* and *mAdd1*, respectively. The policy for job *mProject2* is to re-try after 10 seconds in case of failure; for job *mAdd1*, the policy is to re-submit the flow to another engine and re-start from its preceding job *mImgtbl1*. It is more logical to apply the re-submit policy on the flow scope such that re-submit will be triggered in any failed job step. But, we believe these two scenarios here are illustrative enough to demonstrate our different fault handling policies.

In Fig, 8, we show that the base flow plus the two policies are transformed into an expanded flow with JSDL and fault handling capability (Fig. 8 (b)). For space limit consideration, here we only give the JSDL definition of job *mAdd1* (Fig. 8(c)).

We will demonstrate the effects in migrating instance between two WPS servers, i.e., from server *saba10* to server *weitan*. The Montage job flow is instantiated at *saba10*, and when *mAdd1* fails, the flow instance is migrated to *weitan*. We use Business Process Choreographer (BPC) explorer [24] to monitor the orchestration of the Montage flow. The Montage flow is initiated with the name *Montage_saga10*. When job *mProject2* fails, the flow will automatically re-try it after 10 seconds (as discussed in Section 3). When job *mAdd1* fails, the fault-handling service suspends the flow instance at *saba10* (Fig. 9 (a)), and the flow instance data is exported into a XML file named *rescue.xml* (the size is about 560KB). When the user decides that *Montage_saga10* should be re-submit to server *weitan*, the fault-handling service imports *rescue.xml* to *weitan* (see Fig. 9 (b) for the BPC explorer at *weitan*, please be noted that the flow instance is restored from *saba10* to *weitan*). Then *Montage_saga10* will resume in *weitan* following the policy, that is, to restart from job *mImgtbl1* (Fig. 9 (c)). If we compare Fig. 9 (a) and (c), we could find jobs *mImgtbl1* and *mAdd1* are activated (submitted) at different time on two servers (for example, job *mImgtbl1* is activated on *saba10* at 5/8/07 4:26:28 PM and on *weitan* at 5/8/07 10:36:40 PM), this shows that when *Montage_saga10* is resumed at *weitan*, jobs *mAdd1* and *mImgtbl1* are executed for a second time (and the BPC explorer only show the latest execution time of them). That is to say, when *Montage_saga10* is resumed on *weitan*, the flow is re-started from the preceding job of *mAdd1*, i.e., *mImgtbl1*.




```
  <?xml version="1.0" encoding="UTF-8" ?>
- <jsdl:jobDefinition xmlns:jsdl="http://www.ibm.com/xmlns/prod/scheduling/1.0/jsdl"
      xmlns:jsdle="http://www.ibm.com/xmlns/prod/scheduling/1.0/jsdle" name="mAdd1">
  - <jsdl:application name="executable">
  - <jsdle:executable path="/opt/Montage_v3.0/bin/mAdd">
  - <jsdle:arguments>
    <jsdle:value>-p</jsdle:value>
    <jsdle:value>/opt/m101/projdir</jsdle:value>
    <jsdle:value>/opt/m101/images.tbl</jsdle:value>
    <jsdle:value>/opt/m101/template.hdr</jsdle:value>
    <jsdle:value>/opt/m101/final/m101.fits</jsdle:value>
    </jsdle:arguments>
    </jsdle:executable>
  </jsdl:application>
</jsdl:jobDefinition>
```

(c)

Fig. 8. Sample Montage application: (a) base flow (b) expanded flow (c) JSDL description of job *mAdd1*

Details | Process Input Message | Activities | Events | Related Processes | Tasks | Custom Properties

Activity Name	State	Kind	Owner	Activated
MAdd1	Failed	Invoke		5/8/07 4:26:44 PM
MImgtbl1	Finished	Invoke		5/8/07 4:26:28 PM
MProject2	Finished	Invoke		5/8/07 4:25:21 PM
MProject4	Finished	Invoke		5/8/07 4:25:18 PM
MProject3	Finished	Invoke		5/8/07 4:25:18 PM
MProject1	Finished	Invoke		5/8/07 4:25:18 PM
Imgtbl	Finished	Invoke		5/8/07 4:23:51 PM
Startjobflow	Finished	Receive		5/8/07 4:23:50 PM

(a) *Montage_saba10* initiated at *saba10*

Details | Process Input Message | Activities | Events | Related Processes | Tasks | Custom Properties

Activity Name	State	Kind	Owner	Activated
MAdd1	Failed	Invoke		5/8/07 4:26:44 PM
MImgtbl1	Finished	Invoke		5/8/07 4:26:28 PM
MProject2	Finished	Invoke		5/8/07 4:25:21 PM
MProject4	Finished	Invoke		5/8/07 4:25:18 PM
MProject3	Finished	Invoke		5/8/07 4:25:18 PM
MProject1	Finished	Invoke		5/8/07 4:25:18 PM
Imgtbl	Finished	Invoke		5/8/07 4:23:51 PM
Startjobflow	Finished	Receive		5/8/07 4:23:50 PM

(b) *Montage_saba10* re-submitted to *weitan*

Details | Process Input Message | Process Output Message | Activities | Events | Related Processes | Tasks | Custom

Activity Name	State	Kind	Owner	Activated
Reply	Finished	Reply		5/8/07 10:37:31 PM
MJPEG1	Finished	Invoke		5/8/07 10:37:16 PM
MAdd1	Finished	Invoke		5/8/07 10:37:01 PM
MImgtbl1	Finished	Invoke		5/8/07 10:36:40 PM
MProject2	Finished	Invoke		5/8/07 4:25:21 PM
MProject4	Finished	Invoke		5/8/07 4:25:18 PM
MProject3	Finished	Invoke		5/8/07 4:25:18 PM
MProject1	Finished	Invoke		5/8/07 4:25:18 PM
Imgtbl	Finished	Invoke		5/8/07 4:23:51 PM
Startjobflow	Finished	Receive		5/8/07 4:23:50 PM

(c) *Montage_saba10* re-started and completed at *weitan*

Fig. 9. The BPC explorer to illustrate flow instance migration between *saba10* and *weitan*

6 Related Works

Most works on using BPEL for job flow can be classified into two categories. The first approach [8] extends BPEL model elements, which make the flow model intuitive and simple. However, the workflow engine needs to be modified to deal with the model extension for jobs. The second approach [7, 25, 26] uses standard BPEL activity, so that the models are less intuitive and sometimes verbose to meet the needs of job flow. However, these models adhere to the standard BPEL and thus portable among BPEL-compliant flow engines. Our work falls into the second category of approach. However, the two-stage modeling approach gracefully hides the complexity to deal with jobs submission and fault-handling, while keep the advantage of using existing BPEL engine.

Sedna [10] is a BPEL-based environment for visual scientific workflow modeling. Domain specific abstraction layers are added in Sedna to increase the expressiveness of BPEL for scientific workflows. This method is similar to our two-stage approach. However, fault-handling issue is not addressed in that work.

TRAP/BPEL [18] is a framework that supports runtime selection of equivalent services for monitored services. An exemplary usage of this framework is for selection of recovery services when monitored services fail. By introducing a proxy as the generic fault handler, the logic in the proxy can dynamically select various recovery services according to some configurable recovery polices during runtime. Unlike the runtime dynamic support in TRAP/BPEL, the fault-handling services and policies for job flow are specified during modeling time in BPEL4Job. We require process and application flow modelers to provide directives on the scope (e.g. task or flow level) and types (e.g. re-try, re-submit) of fault recovery.

GridSam [27] provided a set of generic web services for job submission and monitoring. Our generic job proxy takes inspiration from this work. However, in our job proxy, job submission and job status query are combined into a single synchronous scheduling service invocation, with which the job failure/success status is returned. This approach provides a more compact job-submission interface to the flow engine, so that for each job submission the flow engine does not need to use two separate activities to submit job and query job status respectively.

DAGMan used in Condor is popular in many grid job management systems to manage job flow. The fault handling mechanism in DAGMan is re-try and rescue workflow (a kind of re-submit). Our idea of flow re-submission is similar to rescue DAG. Unlike DAGMan, our approach is policy-based and needs to consider the persistent states of job flows in BPEL-compliant engines.

7 Conclusion and Future Work

In this paper, we address two challenging issues in using WS-BPEL for job flow orchestration: the predominantly asynchronous interactions with job execution on dynamic resources, and the fault handling in job flow. We propose a design, called BPEL4Job, to illustrate our approach. BPEL4Job has three unique features: a two-stage approach for job flow modeling with integration with fault-handling policies, a generic job proxy to facilitate the asynchronous nature of job submission and job status notification, and a rich set of fault handling schemes including a novel method for instance migration between different flow engines in distributed system environment.

One direction of future work includes support for the definition and enforcement of more complicated fault-handling policies other than the proposed clean-up, re-try and re-submit. Our solution to instance migration can be extended to other related scenarios such as load balance between flow engines and versioning support for long-running processes. For the versioning support for long-running BPEL processes, if a template of a long-running BPEL process changes during the execution of many instances, the process instances that conform to the old template may need to be migrated to conform to the new one.

References

1. Leymann, F.: Choreography for the Grid: towards fitting BPEL to the resource framework. Concurrency and Computation-Practice & Experience 18(10), 1201–1217 (2006)
2. Jordan, D., et al.: Web Services Business Process Execution Language Version 2.0 (2007), Available from: http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf

3. Couvares, P., et al.: Workflow Management in Condor. In: Taylor, I.J., et al. (eds.) Workflows for e-Science, Springer, Heidelberg (2007)
4. Oinn, T., et al.: Taverna/myGrid: Aligning a Workflow System with the Life Sciences Community. In: Taylor, I.J., et al. (eds.) Workflows for e-Science, pp. 300–319. Springer, Heidelberg (2007)
5. Yu, J., Buyya, R.: A taxonomy of scientific workflow systems for grid computing. Journal of Grid Computing 34(3), 44–49 (2006)
6. Slominski, A.: Adapting BPEL to Scientific Workflows. In: Taylor, I.J., et al. (eds.) Workflows for e-Science, pp. 212–230. Springer, Heidelberg (2007)
7. Amnuaykanjanasin, P., Nupairoj, N.: The BPEL orchestrating framework for secured grid services. In: ITCC 2005. International Conference on Information Technology: Coding and Computing (2005)
8. Dörnemann, T., et al.: Grid Workflow Modelling Using Grid-Specific BPEL Extensions. In: German e-Science Conference 2007, Baden-Baden (2007)
9. Emmerich, W., et al.: Grid Service Orchestration using the Business Process Execution Language (BPEL). In: UCL-CS Research Note RN/05/07, University College London, UK (2005)
10. Wassermann, B., et al.: Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling. In: Taylor, I.J., et al. (eds.) Workflows for e-Science, pp. 428–449. Springer, Heidelberg (2007)
11. Gucer, V., Lowry, M.A., Knudsen, F.B.: End-to-End Scheduling with IBM Tivoli Workload Scheduler Version 8.2., pp. 33–34. IBM Press (2004)
12. BMCSoftware: Meet Your Business Needs Successfully With CONTROL-M For z/OS. Available from: www.bmc.com/USA/Promotions/attachments/controlm_for_os390_and_zOS.pdf
13. Slomiski, A.: On using BPEL extensibility to implement OGSI and WSRF Grid workflows. Concurrency and Computation: Practice & Experience 18(10), 1229–1241 (2006)
14. Montage Tutorial: m101 Mosaic (2007), Available from: http://montage.ipac.caltech.edu/docs/ m101tutorial.html
15. Anjomshoaa, A., et al.: Job Submission Description Language (JSDL) Specification v1.0. Proposed Recommendation from the JSDL Working Group (2005), Available from http://www.gridforum.org/documents/GFD.56.pdf
16. W3C: Web Services Policy 1.2 - Framework (WS-Policy) (2006), Available from http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/
17. Soonwook, H., Kesselman, C.: Grid workflow: a flexible failure handling framework for the grid. In: HPDC'03. 12th IEEE International Symposium on High Performance Distributed Computing, Seattle, WA USA (2003)
18. Ezenwoye, O., Sadjadi, S.M.: TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services. In: WEBIST-2007. International Conference on Web Information Systems and Technologies, Barcelona, Spain (2007)
19. Condor. Available from: http://www.cs.wisc.edu/condor/
20. Platform LSF. Available from: http://www-cecpv.u-strasbg.fr/Documentations/lsf/html/lsf6.1_admin/E_jobrequeue.html
21. IBM Websphere Process Server. Available from: http://www-306.ibm.com/software/integration/wps/
22. IBM Websphere Integration Developer. Available from: http://www-306.ibm.com/software/ integration/ wid/

23. IBM Tivoli Dynamic Workload Broker. Available from: http://www-306.ibm.com/software/tivoli/products/dynamic-workload-broker/index.html
24. Starting to use the Business Process Choreographer Explorer (2007), Available from: http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wsps.ins.doc/doc/bpc/t7stwcl.html
25. Kuo-Ming, C., et al.: Analysis of grid service composition with BPEL4WS. In: 18th International Conference on Advanced Information Networking and Applications (2004)
26. Tan, K.L.L., Turner, K.J.: Orchestrating Grid Services using BPEL and Globus Toolkit 4. In: 7th PGNet Symposium (2006)
27. GridSAM - Grid Job Submission and Monitoring Web Service (2007), Available from: http://gridsam.sourceforge.net/2.0.1/index.html