UNICORE WORKFLOW SYSTEM MANUAL

UNICORE Team

Document Version:	1.0.0
Component Version:	7.6.0
Date:	06 06 2016

Contents

The UNICORE Workflow System provides advanced workflow processing capabilities using UNICORE Grid resources. Its main components are the Workflow Engine and the Service Orchestrator. While the Workflow Engine provides high-level control constructs (for-each, while, if-then-else, etc), the Service Orchestrator contains a powerful, extensible resource broker, and deals with execution of single UNICORE jobs.

For more information about UNICORE visit http://www.unicore.eu.

1 Installing and setting up the UNICORE workflow servers

This chapter covers basic installation of the workflow system and integration of workflow services into an existing UNICORE Grid.

As a general note, the workflow services are organized into two UNICORE/X instances termed "workflow server" and "servorch server". General UNICORE configuration concepts (such as gateway integration, shared registry, attribute sources) fully apply, and you should refer to the UNICORE/X manual for details.

1.1 Prerequisites

- Java 8 (JRE or SDK) or later
- An existing UNICORE installation with Gateway, Shared Registry and one ore more UNICORE/X target systems.
- (for production use) Two server certificates for Workflow engine and Service orchestrator.
- For storing workflow input and output data you need one of
  - a "global storage" service (see below)
  - a StorageFactory service
- If you want to use the RESTful API for submitting workflows, you'll also need Unity

1.2 Updating from previous versions

Please refer to the separate release notes!

Note

On Windows, please stop and uninstall the services before updating! Uninstalling works by executing

```
workflow\bin\uninstall.bat
servorch\bin\uninstall.bat
```

In any case you need to replace the jar files and the wrapper.conf files for workflow and servorch by the new versions.

1.3 Installation

The workflow system is available either as a bundle containing workflow engine and service orchestrator, OR as separate Linux packages (deb or rpm) for workflow engine and service orchestrator.

The basic installation procedure is completely analogous to the installation of the UNICORE core servers.

If you downloaded the workflow system bundle, either use the graphical installer, or untar the tar.gz, edit configure.properties and run configure.py

- Graphical installer: during installation, you will be asked for the parameters of your UNICORE installation.
- Using the tar.gz bundle: please review the configure.properties file and edit the parameters to integrate the workflow services into your existing UNICORE environment. Then call `./configure.py` to apply your settings to the configuration files. Finally use `./install.py` to install the workflow server files to the selected installation directory.

If using the Linux packages, simply install using the package manager of your system.

1.4 Setup

After installation, there are some manual steps needed to integrate the new servers into your UNICORE installation.

- Gateway: edit `gateway/conf/connections.properties` and add the connection data for the workflow server(s). For example,

```
WORKFLOW = https://localhost:7700
SERVORCH = https://localhost:7701
```

- XUUDB: if you chose to use an XUUDB for workflow and service orchestrator, you might have to add entries to the XUUDB to allow users access to the workflow engine. Optionally, you can edit the GCID used by the workflow/servorch servers, so that existing entries in the XUUDB will match.
- Registry: if the registry is setup to use access control (which is the default), you need to allow the workflow and servorch services to register themselves in the Registry. The exact procedure depends on how you configured your Registry, please cross-reference the section "Enabling access control" in the Registry manual. If you're using default certificates and the XUUDB, the required entries can be added as follows.

```
cd xuudb
bin/admin.sh add REGISTRY <workflow>/conf/workflow.pem nobody  ↩
    server
bin/admin.sh add REGISTRY <servorch>/conf/servorch.pem nobody  ↩
    server
```

1.5 Workflow data storage

For storing workflow data (i.e. input/output files needed by the workflow tasks) a storage service instance has to be available. Currently there are two options, using a storage factory or using a shared storage instance. In fact, if multiple options are available at runtime, users using the UNICORE Rich Client (URC) can choose one when they submit their workflows.

1.5.1 Storage Factory

This is the "best" way to store workflow data. Each workflow will store its data on its own storage service instance, making management of these data simpler. The clients (UCC and URC) allow to choose the storage factory that should be used.

1.5.2 Single shared storage

The workflow system can use a single shared normal UNICORE storage service instance for storing files shared between workflow tasks.

Note
while this is simple to set up, it can create a bottleneck in your system, because there is no automated cleanup of workflow data.

The storage to be used can be configured on any UNICORE container running StorageManagement and FileTransfer services. For example, one of the target systems can be used for this purpose.

Please refer to the "Configuring shared storage services" section in the UNICORE/X manual to learn how to set up a shared storage.

1.6 Verifying the installation

If you use the UNICORE Rich Client, you should see the workflow service in the Grid Browser view, and you should be able to submit workflows to it.

Using the UNICORE commandline client, you can check whether the new servers are available and accessible:

```
ucc system-info -l
```

should include output such as

```
Checking for Workflow submission service ...
... OK, found 1 service(s)
   + https://localhost:8080/WORKFLOW/services/WorkflowFactory?res= ↩
       default_workflow_submission

Checking for Service orchestrator ...
... OK, found 1 service(s)
   + https://localhost:8080/SERVORCH/services/ServiceOrchestrator
```

To check whether the services are accessible, you can use

```
ucc wsrf getproperties https://localhost:8080/WORKFLOW/services/ ↩
    WorkflowFactory?res=default_workflow_submission
```

and get output such as

```
<rp:GetResourcePropertyDocumentResponse>
etc. etc.
```

1.6.1 Running a test job

Using UCC again, you can submit workflows

```
ucc workflow-submit /path/to/ucc/samples/date.swf
```

and get the ID of your new workflow back

```
https://localhost:8080/WORKFLOW/services/WorkflowManagement?res ↩
   =7959937b-897a-49f1-aa7d-f485491872d5
```

1.7 RESTful API

Since version 7.6.0, the Workflow server allows workflow submission and management using a RESTful API. Please refer to the section on "RESTful services" in the UNICORE/X manual (7.6 or later), since the configuration is exactly the same.

Note

If you want to submit workflows (in contrast to just checking their status) you must setup Unity for authentication, because this is currently the only way to get delegation, i.e. allow the workflow engine to work on your behalf.

You can find an API reference and usage examples on the UNICORE wiki at https://sourceforge.net/p/unicore/wiki/REST_API

2 Configuration of the Workflow server

This chapter covers configuration options for the Workflow server. Since the Workflow server is running in the same underlying environment (UNICORE Services Environment, USE), you can find basic configuration options in the UNICORE/X manual.

NOTE

The configuration files in the distribution are commented, and contain example settings for all the options listed here.

Depending on how you installed the server, the files are located ion

- `/etc/unicore/workflow` (Linux package)
- `<basedir>/workflow/conf` (standalone installer)

2.1 Workflow processing

Some details of the workflow engine's behaviour can be configured. All these settings are made in `uas.config`.

2.1.1 Limits

To avoid too many tasks submitted (possibly erroneously) from a workflow, various limits can be set.

- `workflow.maxActivitiesPerGroup` limits the total number of tasks submitted for a single group (i.e. (sub-)workflow). By default, this limit is 1000, ie. a maximum number of 1000 jobs can be created by a single group. Note, that it is not possible to limit the total number of jobs for any workflow, it can only be applied to individual parts of the workflow (such as loops).
- `workflow.forEachMaxConcurrentActivities` limits the maximum number of tasks in a for-each group that can be active at the same time (default: 20).

2.1.2 Resubmission

The workflow engine will (in some cases) resubmit failed tasks to the service orchestrator. To completely switch off the resubmission,

```
workflow.resubmitDisable=true
```

To change the maximum number of resubmissions from the default "3",

```
workflow.resubmitLimit=3
```

2.1.3 Disabling tracing

To disable sending messages to the tracer component, set

```
workflow.tracing=false
```

2.1.4 Cleanup behaviour

This controls the behaviour when a workflow is removed (automatically or by the user). By default, the workflow engine will remove all child jobs, but will keep the storage where the files are. This can be controlled using two properties

- `workflow.cleanupStorage` remove storage when workflow is destroyed (default: false)
- `workflow.cleanupJobs` remove jobs when workflow is destroyed (default: true)

2.2 XNJS settings

The workflow engine uses the XNJS library for processing workflows. The XNJS has a separate configuration file, which is controlled using the following property

```
workflow.xnjsConfiguration=conf/xnjs.xml
```

The number of threads used by the workflow engine for processing can be controlled in the `xnjs.xml` file. Note, this does not control the number of parallel activities etc, since all XNJS processing is asynchronous. The default number (4) is usually sufficient.

What is more important is the data directory where the XNJS will store its state. This should be on a fast (local) filesystem for maximum performance. Shared (NFS) directories should not be used.

These two properties are set using

```
<eng:Properties>
 <eng:Property name="XNJS.statedir" value="data/NJSSTATE"/>
 <eng:Property name="XNJS.numberofworkers" value="4"/>
</eng:Properties>
```

2.3 Location mapper

The location mapper provides a crucial service: it is used to obtain "abstract names" for files, i.e. clients and server components can define names that refer to actual files stored on some storage without having to deal with the actual file locations.

The location mapper uses its own database for storing these mappings, which can be either H2 or MySQL. The database configuration is done in `wsrflite.xml` using a set of property values named `org.chemomentum.dataManagement.locationManager.*`

2.4 Tracer

The (optional) tracer service stores timestamps for activities associated with any given workflow, for example submission time, workflow to service orchestrator submission, job submissions, etc. It is used on the clients to show time profile data to the user. The URC contains a nice user interface for interacting with this trace data.

This data is stored in a H2 database, which stores its data on the filesystem (in the usual persistence directory). Currently no other database is supported.

2.5 Property reference

A complete reference of the properties for configuring the Workflow server is given in the following table.

Property name	**Type**	**Default value / mandatory**	**Description**
`workflow.cleanupJobs`	[true, false]	`true`	Whether to remove child jobs when the workflow is destroyed.
`workflow.cleanupStorage`	[true, false]	`false`	Whether to cleanup the workflow storage when the workflow is destroyed.
`workflow.forEachMaxConcurrentActivities`	integer >= 1	`100`	Maximum number of concurrent for-each iterations.
`workflow.maxActivitiesPerGroup`	integer >= 1	`1000`	Maximum number of workflow activities per activity group.
`workflow.pollingInterval`	integer >= 1	`600`	Interval in seconds for (slow) polling of job states from the service orchestrator.
`workflow.resubmitDisable`	[true, false]	`false`	Whether disable automatic re-submission of failed jobs.
`workflow.resubmitLimit`	integer >= 1	`3`	Maximum number of re-submissions of failed jobs.
`workflow.tracing`	[true, false]	`true`	Whether to send trace messages to the tracer service.
`workflow.xnjsConfiguration`	string	`conf/xnjs.xml`	XNJS configuration file.

3 Servorch server

This chapter covers configuration options for the Servorch server. Since the Servorch server is running in the same underlying environment (UNICORE Services Environment, USE), you can find basic configuration options in the UNICORE/X manual.

Additional servorch server configuration is performed in the `uas.config` file. Advanced re-configuration such as adding new brokering strategies can be done in the set of Spring configuration files `servorch/conf/spring`.

NOTE

The directory containing the Spring config files is controlled by the property `servorch.springConfig` in `uas.config`.

Depending on how you installed the server, the config files are located in

- `/etc/unicore/servorch` (Linux package)
- `<basedir>/servorch/conf` (standalone installer)

3.1 Data directories

By default, runtime data is placed into the "data" subdirectory in the service orchestrator directory. To change, there are several properties.

- The usual UNICORE data directory is set in `wsrflite.xml` in the `persistence.directory` property (default: "data")
- The local indexes created by the resource broker are placed into the directory configured in `conf/spring/attributeCache.xml`, by default this is set to "data/brokering/attributes".

3.2 Preferred file transfer protocol

If you want to change the preferred protocol, you may set

```
servorch.outcomesProtocol=BFT
```

The default "BFT" will work with any UNICORE installation. If available, "UFTP" will provide more performance.

3.3 Job processing

A number of properties control how jobs are processed by the service orchestrator.

- `servorch.jobSupervisors` controls the number of threads that act as "job supervisors". These threads are used for resource brokering, job submission, status polling and storing job outcomes. The default is "10".
- `servorch.jobUpdateInterval` controls the number of milliseconds between two job status polls. The default is "5000".
- `servorch.jobFirstUpdateInterval` is the delay in milliseconds between job submission and first status check. The default is "5000".
- `servorch.outcomesUpdateInterval` is the number of milliseconds between status polls while transferring files. The default is "5000".

3.4 Resource checking and attribute gathering interval

The service orchestrator periodically updates its internal information about available sites and their resources. The update interval is controlled in the file `conf/uas.config`

- `servorch.siteUpdateInterval` controls the number of milliseconds between two site refresh calls. The default is "20000".

3.5 Property reference

A complete reference of the properties for configuring the Servorch server is given in the following table.

Property name	**Type**	**Default value / mandatory**	**Description**
`servorch.dbClearOnStartup`	[true, false]	`false`	Whether to clear the persisted state when starting.
`servorch.jobFirstUpdateInterval`	integer >= 1	`5000`	Number of milliseconds to wait after job submission before the first status check.
`servorch.jobSupervisors`	integer >= 1	`10`	Number of threads that act as *job supervisors*. These threads are used for resource brokering, job submission, status polling and storing job outcomes.

Property name	Type	Default value / mandatory	Description
`servorch.jobUpdateInterval`	integer >= 1	`5000`	Number of milliseconds between two job status polls.
`servorch.numParallelTransfers`	integer >= 1	`5`	Maximum number of parallel data transfers.
`servorch.outcomesProtocol`	string	`BFT`	Protocol to use for storing outcomes.
`servorch.outcomesUpdateInterval`	integer >= 1	`5000`	Number of milliseconds between two progress checks while storing outcomes.
`servorch.siteUpdateInterval`	integer >= 1	`20000`	Number of milliseconds between site status polls.
`servorch.tracing`	[true, false]	`true`	Whether to send trace messages to the tracer service.

4 The "simple workflow" workflow description language

4.1 Introduction

This chapter provides an overview of the "simple workflow" XML dialect that is used to describe workflows. It will allow you to write workflows "by hand", i.e. without using the graphical UNICORE Rich client. These can be submitted for example using the UNICORE commandline client (UCC).

The workflow language is an XML dialect, the corresponding XML schema can be found in the UNICORE SourceForge code repository

After presenting all the constructs individually, several complete Section 4.7 are given.

4.2 Overview and simple constructs

The overall workflow document has the following form:

```
<Workflow xmlns="http://www.chemomentum.org/workflow/simple"
          Id="...">
  <Documentation>?
  <DeclareVariable>*
  <Activity>*
  <Transition>*
  <SubWorkflow>*
```

```
  <Option>*
</Workflow>
```

Here and in the following we use a simple notation to denote XML elements and their multiplicity, where "*" denotes zero or multiple occurences and "?" denotes zero or one occurence of a given element. In the next sections the elements of the workflow description will be discussed in detail.

NOTE

The `Id` attribute is used in many workflow elements, and must be an identifier string that is UNIQUE within the workflow.

4.2.1 Documentation

The Documentation element allows to add some meta-information to the workflow description, i.e. it will be ignored by the processing engine. In detail

```
<Documentation xmlns="http://www.chemomentum.org/workflow/simple">
  <Name>?
  <Creator>?
  <CreationDate>?
  <Comment>*
</Documentation>
```

4.2.2 Activities

Activity elements have the following form

```
<Activity xmlns="http://www.chemomentum.org/workflow/simple"
          Id="..." Type="..." >
  <Option Name="...">*
  <JSDL>?
</Activity>
```

The `Id` attribute must be unique within the workflow. There are different types of activity, which are distinguished by the "Type" attribute.

- "START" denotes an explicit start activity. If no such activity is present, the processing engine will detect the proper starting activities
- "JSDL" denotes a executable (job) activity. In this case, the `JSDL` sub element holds the JSDL job definition
- "ModifyVariable" allows to modify a workflow variable. An option named "variableName" identifies the variable to be modified, and an option "expression" holds the modification expression in the Groovy programming language syntax. See also the variables section later

- "Split": this activity can have multiple outgoing transitions. All transitions with matching conditions will be followed. This is comparable to an "if() ... if() ... if()" construct in a programming language.
- "Branch": this activity can have multiple outgoing transitions. The transition with the first matching condition will be followed. This is comparable to an "if() ... elseif() ... else()" construct in a programming language
- "Merge" merges multiple flows without synchronising them
- "Synchronize" merges multiple flows and synchronises them
- "HOLD" stops further processing of the current flow until the client explicitely sends continue message.

4.2.3 Subworkflows

The workflow description allows nested sub workflows, which have the same formal structure as the main workflow

```
<SubWorkflow xmlns="http://www.chemomentum.org/workflow/simple"
             Id="..."
  <DeclareVariable>*
  <Activity>*
  <Transition>*
  <SubWorkflow>*
  <Option>*
</SubWorkflow>
```

4.2.4 JSDL activities

JSDL activities are the basic executable pieces of a workflow. The embedded JSDL job definition will be packed in a so-called work assignment and sent to a service orchestrator for processing.

```
<s:Workflow xmlns:s="http://www.chemomentum.org/workflow/simple"
          xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">

  <s:Documentation>
    <s:Comment>Simple 'Date'</s:Comment>
  </s:Documentation>

  <s:Activity Id="date1" Type="JSDL">
   <s:JSDL>
      <jsdl:JobDescription>
        <jsdl:Application>
          <jsdl:ApplicationName>Date</jsdl:ApplicationName>
        </jsdl:Application>
```

```
      </jsdl:JobDescription>
    </s:JSDL>
   </s:Activity>

</s:Workflow>
```

The processing of the JSDL activity can be influenced using Option sub-elements. Currently the following options can be used

- `IGNORE_FAILURE` if set to "true", the workflow engine will ignore any failure of the task and continue processing as if the activity had been completed successfully. NOTE: this has nothing to do with the exit code of the actual UNICORE job! Failure means for example data staging failed, or the service orchestrator did not find a matching target system for the job.
- `MAX_RESUBMITS` set to an integer value to control the number of times the activity will be retried. By default, the workflow engine will re-try three times (except in those cases where it makes no sense to retry).

4.2.5 Transitions and conditions

The basic flow of control in a workflow is handled using `Transition` elements. These reference to "From+ and `To` activities (or subflows) and may have conditions attached. If no condition is present, the transition is followed unconditionally.

The syntax is as follows.

```
<Transition xmlns="http://www.chemomentum.org/workflow/simple"
 From="..." To="..." Id="...">
  <Condition>?
</Transition>
```

The `From` and `To` attributes denote Activity or SubWorkflow Id's, and the `Id` attribute has to be workflow-unique.

The optional `Condition` element has the following syntax

```
<Condition xmlns="http://www.chemomentum.org/workflow/simple">
  <Expression>...</Expression>
</Condition>
```

where `Expression` is string-valued. The workflow engine offers some pre-defined functions that can be used in these expressions. For example you can use the exit code of a job, or check for the existence of a file within these expressions.

- `eval(expr)` Evaluates the expression "expr" in Groovy syntax, which must evaluate to a boolean. The expression may contain workflow variables

- `exitCodeEquals(activityID, value)` Allows to compare the exit code of the Grid job associated with the Activity identified by “activityID” to "value"
- `exitCodeNotEquals(activityID, value)` Allows to check the exit code of the Grid job associated with the Activity identified by "activityID", and check that it is different from "value"
- `fileExists(activityID, path)` Checks that the working directory of the Grid job associated with the given Activity contains a file "path"
- `fileLengthGreaterThanZero(activityID, path)` Checks that the working directory of the Grid job associated with the given Activity contains a file "path", which has a non-zero length
- `before(time)` and `after(time)` check whether the current time is before or after the given time (in "yyyy-MM-dd HH:mm" format)

4.3 Using workflow variables

Workflow variables need to be declared using a `DeclareVariable` element before they can be used.

```
<DeclareVariable xmlns="http://www.chemomentum.org/workflow/simple ↩
    ">
  <Name>
  <Type>
  <InitialValue>
</DeclareVariable>
```

Currently variables of type "STRING", "INTEGER" , "FLOAT" and "BOOLEAN" are supported.

Variables can be modified using an activity of type `ModifyVariable`.

For example, to increment the value of the "COUNTER" variable, the following Activity is used

```
<Activity xmlns="http://www.chemomentum.org/workflow/simple"
Type="ModifyVariable" Id="incrementCounter">
    <Option name="variableName">COUNTER</Option>
    <Option name="expression">COUNTER += 1;</Option>
</Activity>
```

The option named "expression" contains an expression in Groovy syntax (which is very close to Java).

The workflow engine will replace variables in JSDL data staging sections and environment definitions, allowing to inject variables into jobs. Examples for this mechanism will be given in the examples section.

4.4 Loop constructs

Apart from graphs constructed using `Activity` and `Transition` elements, the workflow system supports special looping constructs, for-each, while and repeat-until, which to setup allow complex workflows very easily.

4.5 While and repeat-until loops

These allow to loop a certain part of the workflow while (or until) a condition is met. A while loop looks like this

```
<s:SubWorkflow xmlns:s="http://www.chemomentum.org/workflow/simple"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               Id="while" xsi:type="s:WhileType" >

 <s:DeclareVariable Id="decl">
    <s:Name>C</s:Name>
    <s:Type>INTEGER</s:Type>
    <s:InitialValue>1</s:InitialValue>
 </s:DeclareVariable>

 <s:SubWorkflow Id="while_body">

  <s:Activity Id="job" Type="JSDL">
    <s:JSDL> ... </s:JSDL>
  </s:Activity>

  <!-- this modifies the variable used in the
       'while' loop's exit condition -->
  <s:Activity Id="mod" Type="ModifyVariable">
   <s:Option name="variableName">C</s:Option>
   <s:Option name="expression">C++;</s:Option>
  </s:Activity>

  <s:Transition From="job" To="mod" Id="job-mod"/>

 </s:SubWorkflow>

 <!-- exit condition -->
 <s:Condition>
  <s:Expression>eval(C&lt;5)</s:Expression>
 </s:Condition>

</s:SubWorkflow>
```

The necessary ingredients are that the loop body (Id="while_body" in the example) modifies the loop variable ("C" in the example), and the exit condition eventually terminates the loop.

Completely analogously, a repeat-until loop is constructed, the only syntactic difference is that the SubWorkflow" now has a different +xsi:type attribute:

```
<s:SubWorkflow xmlns:s="http://www.chemomentum.org/workflow/simple"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               Id="while" xsi:type="s:RepeatUntilType" >

 <s:DeclareVariable Id="decl">
    <s:Name>C</s:Name>
    <s:Type>INTEGER</s:Type>
    <s:InitialValue>1</s:InitialValue>
 </s:DeclareVariable>

 <s:SubWorkflow Id="repeat_body">

  <s:Activity Id="job" Type="JSDL">
    <s:JSDL> ... </s:JSDL>
  </s:Activity>

  <!-- this modifies the variable used in the
       repeat' loop's exit condition -->
  <s:Activity Id="mod" Type="ModifyVariable">
   <s:Option name="variableName">C</s:Option>
   <s:Option name="expression">C++;</s:Option>
  </s:Activity>

  <s:Transition From="job" To="mod" Id="job-mod"/>

 </s:SubWorkflow>

 <!-- exit condition -->
 <s:Condition>
  <s:Expression>eval(C&lt;5)</s:Expression>
 </s:Condition>

</s:SubWorkflow>
```

Semantically, the repeat-loop will always execute the body at least once, since the condition is checked after executing the body, while in the "while" case, the condition will be checked before executing the body.

4.6 For-each loop

The for-each loop is a complex, yet powerful feature of the workflow system, since it allows parallel execution of the loop body, and different ways of building the different iterations. Put briefly, one can loop over variables (as in the "while" and "repeat-until" case), but one can also loop over enumerated values and (most importantly) over file sets.

The basic syntax is

```
<s:SubWorkflow xmlns:s="http://www.chemomentum.org/workflow/simple"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance ↩
                   "
               Id="..." xsi:type="s:ForEachType"
               IteratorName="...">

 <-- ... activities to be looped over
    (loop body)
 -->
 <s:SubWorkflow Id="..">
 </s:SubWorkflow>

 <!-- define range to loop over -->

 <s:ValueSet> ...  </s:ValueSet>

 OR

 <s:VariableSet> ... <s:/VariableSet>

 OR

 <s:FileSet> ... <s:/FileSet>

 <!-- optional chunking -->
 <:Chunking> ... </s:Chunking>

</s:SubWorkflow>
```

The `IteratorName` attribute allows to control how the "loop iterator variable" is to be called.

4.6.1 The ValueSet element

Using `ValueSet`, iteration over a fixed set of strings can be defined. The main use for this is parameter sweeps, i.e. executing the same job multiple times with different arguments or environment variables.

```
<s:ValueSet xmlns:s="http://www.chemomentum.org/workflow/simple">

 <s:Value>10</s:Value>
 <s:Value>20</s:Value>
 <s:Value>30</s:Value>
 <s:Value>40</s:Value>

</s:ValueSet>
```

In each iteration, the workflow variables "CURRENT_ITERATOR_VALUE" and "CURRENT_ITERATOR_INDEX" will be set to the current value and index.

4.6.2 The `VariableSet` element

The `VariableSet` allows to define the iteration range using a variable, similar to a for-loop in a programming language.

```
<s:VariableSet xmlns:s="http://www.chemomentum.org/workflow/ ↩
    simple">
    <s:VariableName>C</s:VariableName>
    <s:Type>INTEGER</s:Type>
    <s:StartValue>0</s:StartValue>
    <s:Expression>C++</s:Expression>
    <s:EndCondition>C&lt;5</s:EndCondition>
</s:VariableSet>
```

The sub-elements should be self-explanatory.

In each iteration, the workflow variables "CURRENT_ITERATOR_VALUE" and "CURRENT_ITERATOR_INDEX" will be set to the current value and index.

4.6.3 The `FileSet` element

This is a very useful variation of the for-each loop which allows to loop over a set of files, optionally chunking together several files in a single iteration.

The basic structure of a `FileSet` definition is this

```
<s:FileSet xmlns:s="http://www.chemomentum.org/workflow/simple"
           recurse="true|false" indirection="true|false">
    <s:Base> ... <s:/Base>
    <s:Include>?
    <s:Exclude>?
</s:FileSet>
```

The `Base` element defines a base of the filenames, which will be resolved at runtime, and complemented according to the `Includes` and/or `Excludes` elements. The `recurse` attribute allows to control whether the resolution should be done recursively into any subdirectories. The `indirection` attribute is explained below.

For example to recursively collect all PDF files (but not the file named "ununsed.pdf") in a certain directory on a storage:

```
<s:FileSet xmlns:s="http://www.chemomentum.org/workflow/simple"
           recurse="true">
    <s:Base>BFT:https://mysite/services/StorageManagement?res ↩
        =123#/files/pdf/</s:Base>
    <s:Include>*.pdf</s:Include>
    <s:Exclude>unused.pdf</s:Exclude>
</s:FileSet>
```

The following variables are set where ITERATOR_NAME is the loop iterator name defined in the SubWorkflow as shown above.

- ITERATOR_NAME is set to the current iteration index (1, 2, 3, ...)
- ITERATOR_NAME_VALUE is set to the current full file path
- ITERATOR_NAME_FILENAME is set to the current file name (last element of the path)

4.6.4 Indirection

Sometimes the list of files that should be looped over is not known at workflow design time, but will be computed at runtime. Or, you wish simply to list the files in a file, and not put them all in your workflow description. The indirection attribute on a FileSet allows to do just that. If indirection is set to true, the workflow engine will load the given file(s) in the fileset at runtime, and read the actual list of files to iterate over from them. As an example, you might have a file filelist.txt containing a list of UNICORE SMS files and logical files:

```
BFT:https://someserver#/file1
...
BFT:https://someserver#/fileN
c9m:${WORKFLOW_ID}/files/file1
...
c9m:${WORKFLOW_ID}/files/fileN
```

and the fileset

```
<s:FileSet xmlns:s="http://www.chemomentum.org/workflow/simple"
           recurse="false" indirection="true">
    <s:Base>BFT:https://someserver/services/StorageManagement?res ↩
        =123#/</s:Base>
    <s:Include>filelist.txt</s:Include>
</s:FileSet>
```

You can have more than one file list.

4.6.5 Chunking

Chunking allows to group sets of files into a single iteration, for example for efficiency reasons. The number of files in a chunk can be controlled, alternatively the size of the chunk in kbytes can be set.

```
<s:Chunking xmlns:s="http://www.chemomentum.org/workflow/simple">
    <s:Chunksize> ... </s:Chunksize>
    <s:IsKbytes>true|false</s:IsKbytes>
    <s:FilenameFormat> ... </s:FilenameFormat>
    <s:ComputeChunksize>_expression_</s:ComputeChunksize>
</s:Chunking>
```

The `Chunksize` element is either the number of files in a chunk, or (if `IsKbytes` is set to "true") the size of a chunk in kbytes.

For example:

To process 10 files per iteration:

```
<s:Chunking xmlns:s="http://www.chemomentum.org/workflow/simple">
    <s:Chunksize>10</s:Chunksize>
    <s:IsKbytes>false</s:IsKbytes>
</s:Chunking>
```

To process 2000 kBytes of data per iteration:

```
<s:Chunking xmlns:s="http://www.chemomentum.org/workflow/simple">
    <s:Chunksize>2000</s:Chunksize>
    <s:IsKbytes>true</s:IsKbytes>
</s:Chunking>
```

If required the chunksize can also be computed at runtime using the expression given in the `ComputeChunksize` element. In the expression, two special variables may be used. The `TOTAL_NUMBER` variable holds the total number of files iterated over, while the `TOTAL_SIZE` variable holds the aggregated file size in kbytes. The script must return an integer-valued result. The `IsKbytes` element is used to choose whether the chunk size is interpreted as data size or as number of files.

For example:

To choose a larger chunksize if a certain total file size is exceeded:

```
<s:Chunking xmlns:s="http://www.chemomentum.org/workflow/simple">
    <s:IsKbytes>true</s:IsKbytes>
    <s:ComputeChunksize>
       if(TOTAL_SIZE>50*1024)return 5*1024;
       return 2048;
    </s:ComputeChunksize>
</s:Chunking>
```

The optional `FilenameFormat` allows to control how the individual files (which are staged into the job directory) should be named. By default, the index is prepended, i.e. "inputfile" would be named "1_inputfile" to "N_inputfile" in each chunk. The pattern uses the variables respectively. For example, if you have a set of PDF files, and you want them to be named "file_1.pdf" to "file_N.pdf", you could use the pattern

```
<s:FilenameFormat>file_{0}.pdf</s:FilenameFormat>
```

or, if you prefer to keep the existing extensions, but append an index to the name,

```
<s:FilenameFormat>{1}{0}.{2}</s:FilenameFormat>
```

4.7 Examples

This section collects a few simple example workflows. They are intended to be submitted using UCC.

4.7.1 Simple "diamond" graph

This example shows how to use transitions for building simple workflow graphs. It consists of four "Date" jobs arranged in a diamond shape, i.e. "date2a" and "date2b" are executed roughly in parallel. A "Split" activity is inserted to divide the control flow into two parallel branches.

All "stdout" files are staged out to the workflow storage.

```
<s:Workflow xmlns:s="http://www.chemomentum.org/workflow/simple"
          xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">

  <s:Documentation>
    <s:Comment>Simple diamond graph</s:Comment>
  </s:Documentation>

  <s:Activity Id="date1" Type="JSDL">
   <s:JSDL>
      <jsdl:JobDescription>
        <jsdl:Application>
          <jsdl:ApplicationName>Date</jsdl:ApplicationName>
          <jsdl:ApplicationVersion>1.0</jsdl:ApplicationVersion>
        </jsdl:Application>
       <jsdl:DataStaging>
         <jsdl:FileName>stdout</jsdl:FileName>
         <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
         <jsdl:Target>
           <jsdl:URI>c9m:${WORKFLOW_ID}/date1.out</jsdl:URI>
         </jsdl:Target>
         </jsdl:DataStaging>
      </jsdl:JobDescription>
    </s:JSDL>
   </s:Activity>

  <Activity Id="split" Type="Split"/>

  <s:Activity Id="date2a" Type="JSDL">
   <s:JSDL>
      <jsdl:JobDescription>
        <jsdl:Application>
         <jsdl:ApplicationName>Date</jsdl:ApplicationName>
        </jsdl:Application>
       <jsdl:DataStaging>
         <jsdl:FileName>stdout</jsdl:FileName>
         <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
```

```
          <jsdl:Target>
            <jsdl:URI>c9m:${WORKFLOW_ID}/date2a.out</jsdl:URI>
          </jsdl:Target>
          </jsdl:DataStaging>
       </jsdl:JobDescription>
     </s:JSDL>
    </s:Activity>

  <s:Activity Id="date2b" Type="JSDL">
    <s:JSDL>
       <jsdl:JobDescription>
         <jsdl:Application>
          <jsdl:ApplicationName>Date</jsdl:ApplicationName>
         </jsdl:Application>
        <jsdl:DataStaging>
          <jsdl:FileName>stdout</jsdl:FileName>
          <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
          <jsdl:Target>
            <jsdl:URI>c9m:${WORKFLOW_ID}/date2b.out</jsdl:URI>
          </jsdl:Target>
          </jsdl:DataStaging>
       </jsdl:JobDescription>
     </s:JSDL>
    </s:Activity>

  <s:Activity Id="date3" Type="JSDL">
    <s:JSDL>
       <jsdl:JobDescription>
         <jsdl:Application>
          <jsdl:ApplicationName>Date</jsdl:ApplicationName>
         </jsdl:Application>
        <jsdl:DataStaging>
          <jsdl:FileName>stdout</jsdl:FileName>
          <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
          <jsdl:Target>
            <jsdl:URI>c9m:${WORKFLOW_ID}/date3.out</jsdl:URI>
          </jsdl:Target>
          </jsdl:DataStaging>
       </jsdl:JobDescription>
     </s:JSDL>
    </s:Activity>

  <s:Transition Id="date1-split" From="date1" To="split"/>
  <s:Transition Id="split-date2a" From="split" To="date2a"/>
  <s:Transition Id="split-date2b" From="split" To="date2b"/>
  <s:Transition Id="date2b-date3" From="date2b" To="date3"/>
  <s:Transition Id="date2a-date3" From="date2a" To="date3"/>

</s:Workflow>
```

4.7.2 While loop example using workflow variables

The next example shows some uses of workflow variables in a while loop. The loop variable "C" is copied into the job's environment. Another possible use is to use workflow variables in data staging sections, for example to name files.

```
<s:Workflow xmlns:s="http://www.chemomentum.org/workflow/simple"
            xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
            xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl- ↩
                posix"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<s:Activity Id="start" Type="START"/>

<s:SubWorkflow Id="while" xsi:type="s:WhileType" IteratorName="C">

 <s:DeclareVariable Id="decl">
    <s:Name>C</s:Name>
    <s:Type>INTEGER</s:Type>
    <s:InitialValue>0</s:InitialValue>
 </s:DeclareVariable>

 <s:SubWorkflow Id="while_body">

  <s:Activity Id="job" Type="JSDL">
    <s:JSDL>
      <jsdl:JobDescription>
        <jsdl:Application>
           <jsdl1:POSIXApplication>
             <jsdl1:Executable>/bin/echo</jsdl1:Executable>
             <jsdl1:Argument>$TEST</jsdl1:Argument>
             <jsdl1:Environment name="TEST">${C}</jsdl1:Environment ↩
                 >
           </jsdl1:POSIXApplication>
        </jsdl:Application>

         <jsdl:DataStaging>
          <jsdl:FileName>stdout</jsdl:FileName>
         <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
          <jsdl:Target>
            <jsdl:URI>c9m:${WORKFLOW_ID}/out_${C}</jsdl:URI>
          </jsdl:Target>
        </jsdl:DataStaging>

      </jsdl:JobDescription>
    </s:JSDL>
  </s:Activity>

  <!-- this modifies the variable used in the while loop's exit  ↩
      condition -->
```

```
  <s:Activity Id="mod" Type="ModifyVariable">
   <s:Option name="variableName">C</s:Option>
   <s:Option name="expression">C++;</s:Option>
  </s:Activity>

  <s:Transition From="job" To="mod" Id="job-mod"/>

  </s:SubWorkflow>

  <!-- exit condition -->
  <s:Condition>
   <s:Expression>eval(C&lt;=5)</s:Expression>
  </s:Condition>

</s:SubWorkflow>

<s:Transition From="start" To="while" Id="start-while"/>

</s:Workflow>
```

The output files (named using "global" identifiers) can be downloaded using UCC, for example (replace WFID by the real workflow ID obtained after submission)

```
ucc get-file -s c9m:WFID/out_1 -t ./out_1
```

4.7.3 For-each loop example

The next example shows how to use the for-each loop to loop over a set of files. The jobs will stage-in the current file. Also, the name of the current file is placed into the job environment.

```
<s:Workflow xmlns:s="http://www.chemomentum.org/workflow/simple"
            xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
            xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl- ↩
                posix"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<s:Activity Id="start" Type="START"/>

<s:SubWorkflow Id="for" xsi:type="s:ForEachType" IteratorName="IT">

 <s:SubWorkflow Id="for_body">

  <s:Activity Id="job" Type="JSDL">
    <s:JSDL>
      <jsdl:JobDescription>
        <jsdl:Application>
           <jsdl1:POSIXApplication>
             <jsdl1:Executable>/bin/echo</jsdl1:Executable>
```

```
            <jsdl1:Argument>Processing file: $NAME</jsdl1:Argument ↩
                >
            <!-- put current filename into environemt -->
            <jsdl1:Environment name="NAME">${IT_FILENAME}</jsdl1: ↩
                Environment>
          </jsdl1:POSIXApplication>
        </jsdl:Application>

        <!-- stage in the current file -->
        <jsdl:DataStaging>
          <jsdl:FileName>infile</jsdl:FileName>
         <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
          <jsdl:Source>
            <!-- use variable containing file path -->
            <jsdl:URI>${IT_VALUE}</jsdl:URI>
          </jsdl:Source>
        </jsdl:DataStaging>

        <jsdl:DataStaging>
          <jsdl:FileName>stdout</jsdl:FileName>
         <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
          <jsdl:Target>
            <!-- iterator variable contains iteration index -->
            <jsdl:URI>c9m:${WORKFLOW_ID}/out_${IT}</jsdl:URI>
          </jsdl:Target>
        </jsdl:DataStaging>

      </jsdl:JobDescription>
    </s:JSDL>
  </s:Activity>

  </s:SubWorkflow>

  <!-- file set defining which files to loop over -->
  <s:FileSet recurse="false">
   <s:Base>https://mygateway.de:7700/MYSITE/services/ ↩
       StorageManagement?res=default_storage#/</s:Base>
  <s:Include>/myfiles/*</s:Include>

 </s:FileSet>


</s:SubWorkflow>

<s:Transition From="start" To="for" Id="start-for"/>

</s:Workflow>
```

5 Updating an existing UNICORE workflow installation

This chapter covers the steps required to update an existing workflow installation (v7.x).

5.1 Stop the servers

Stop the workflow and servorch servers now.

5.2 Backup

You should make a backup of your existing data, and, if necessary, your config files.

5.3 Update jar files

The Java libraries have to be replaced with the new versions.

5.4 Update config files

Compare the config files from the new version to your existing one. Check the changelog for new features that might require updates to config files.

5.5 Restart the servers

Restart the workflow and servorch servers, and check the logs for any suspicious error messages!