Workflow User Guide

Revision 3.0

December 1, 2006

Table of Contents

Table of Contents

Table of Contents

1 Introduction

The Institute for Genomic Research (TIGR) has many computational pipelines that need to be created, executed, and monitored on an ongoing basis. Examples include the pipeline to download and build PANDA databases, running All-vs-All searches for genome databases, running annotation pipelines, etc. Each pipeline typically includes multiple discrete steps that are executed as a combination of sequential and parallel steps either locally or farmed out to a distributed computing environment (DCE) or grid. To reduce manual intervention, allow resumption of failed pipelines, and streamline the process flow, TIGR's Annotation Software team, ANTware, has designed a system called Workflow that can be used to build, run, and monitor such process pipelines or workflows.

The Java programming language was used to build the Workflow system to enable porting this application to multiple operating systems and build rich GUIs. The system includes a set of command line and GUIs tools to create, execute, manage and monitor workflows. A workflow instance, typically referred as a workflow, is represented as a collection of one or more XML files that defines the hierarchical structure of the process pipeline. The workflow is defined as a hierarchical collection of command sets and commands where the command sets group a set of related commands in a sub-unit of work. The command represent the smallest discrete unit of work such as the execution of a blast job, loading a database, etc, and specifies the action to be taken and the data needed for undertaking this action as parameters. A workflow instance can have subflows (command sets) either that are fully contained in the instance file or that reside as separate subflow instances with a reference in the parent file. Such a workflow is referred to as a nested or hierarchical subflow.The Workflow engine steps through this XML file to identify and launch specialized processors to process all the steps in the instance. The command sets are executed by *Dispatchers* and commands are executed by *Processor*s.

The Workflow system currently includes the tools *EditTemplate, CreateWorkflow, RunWorkflow, ControlWorkflow, KillWorkflow, MonitorWorkflow, CheckWorkflow* and *CleanWorkflowRegistry. EditTemplate* is a GUI tool used to build workflow templates, *CreateWorkflow* is used to build instances of predefined workflows, while the *RunWorkflow* tool is used to execute the workflow instance. *ControlWorkflow* is used to restart one or more stopped or failed commands or command sets in a currently running workflow. *KillWorkflow* is used to kill or terminate a workflow gracefully. *MonitorWorkflow* provides a GUI to monitor the execution of one or more workflows. CheckWorkflow allows a user to determine whether a given workflow is currently running and, if so, on what machine. *CleanWorkflowRegistry* is used to clean up the java RMI registry used by workflow in the event of a catastrophic failure of the engine and also to list all the workflows running on a particular host.

The Workflow system comes with a standard set of command dispatchers and processors but is designed to allow users to build their own specialized command processors in any programming language and add them to the system. The dispatcher set includes *LocalDispatcher*, *DistributedDispatcher* and *RemoteDispatcher*. *LocalDispatcher* executes a set of commands in the workflow on the local machine either serially or in parallel, depending on the type of the command set. *DistributedDispatcher* submits a set of distributable commands in the workflow to an underlying grid (DCE). *RemoteDispatcher* submits an entire subflow to the underlying grid (DCE) as one job; the subflow must reside in a separate XML file. The set of command processors include *SystemCommandProcessor,* which is a general processor to launch various system commands, including standalone applications or utilities, WaitProcessor, which as the name suggests sleeps for the specified duration, and the *WaitForFileProcessor* which waits for the creation of the specified file and *DistributedProcessor*, which is used to launch a command in a distributed environment.

Workflow has been built to run in the background, but there are a number of situations where a user might want notification of the finish or failure of the workflow. To accommodate this, the engine has been built to send notifications via email upon completion or failure of the workflow. In other situations users may want some common action to be taken upon the completion of each step of the workflow. To make this process simple, workflow provides a mechanism of registering either stand alone utilities or special Java listener classes with the engine to receive notifications at different levels of granularity. The workflow engine includes capability for other java process to

communicate with the engine to check execution status, stop the engine, or restart failed commands. This is accomplished through Java's remote method invocation (RMI) API. As a result each instance of the workflow starts a RMI server to listen to requests from other processes.

Workflow system can be used in two scenarios, one where the user generates the instance files and uses the workflow engine for execution of the pipeline, or use the workflow to generate the instance files from template and corresponding configuration files. The template file is a skeleton XML file that defines the structure of the workflow along with the invariant parameters such as executable names, fixed parameters, while delegating the variant parameters to be stored in the configuration file. This allows the user to reuse a pipeline definition with different parameters without having to redefine the structure every time. This mechanism also allows the creation of dynamic workflows where the structure is predefined but the parameters are created by parts of the pipeline.

2 System Requirements

The workflow system has been written in Java and requires at least Java Standard Edition 5.0 or higher. The system has been tested to work under Linux, and Solaris operating systems and relies on the availability of PERL interpreter. The system uses a number of third party tools and libraries that include Log4j, Castor, JUnit, Xerces, etc., among others.

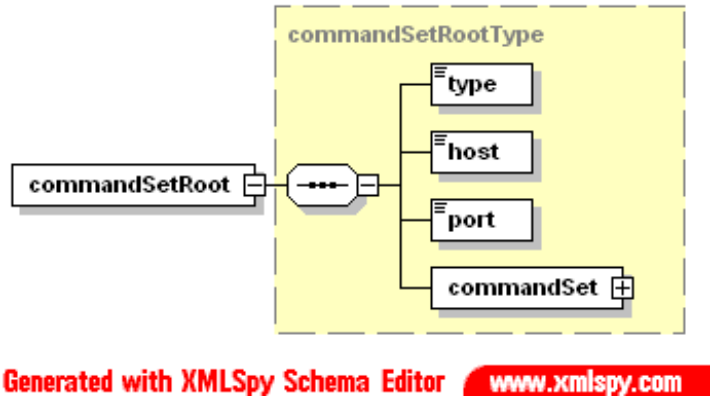
3 Workflow Instance Schema

This section describes the structure of the workflow instance XML schema. A workflow instance is an instance of *CommandSetRoot*, which represents the document and includes one and only one *CommandSet* element. Each command set element can contain one or more *Command* or *CommandSet* elements that represent components of the workflow. The *CommandSet* and *Command* elements can have other simple and complex elements as defined in the schema. Each of the XML elements of this schema are discussed in detail in this section with a graphical representation of the element. The graphical schema presented here pertains primarily to the workflow instance files, the template files use a subset of these tags, and the tables describing these elements indicate which of the elements are required for template and instance files.

3.1 CommandSetRoot

A *CommandSetRoot* is the root XML element and contains a single *CommandSet* element. **Fig 1** shows the schema diagram for the command set root. The major elements of the root element include the type, command set, the host that is executing this instance, and the port on which the workflow engine is listening for RMI calls.

Fig 1: Command Set Root Schema



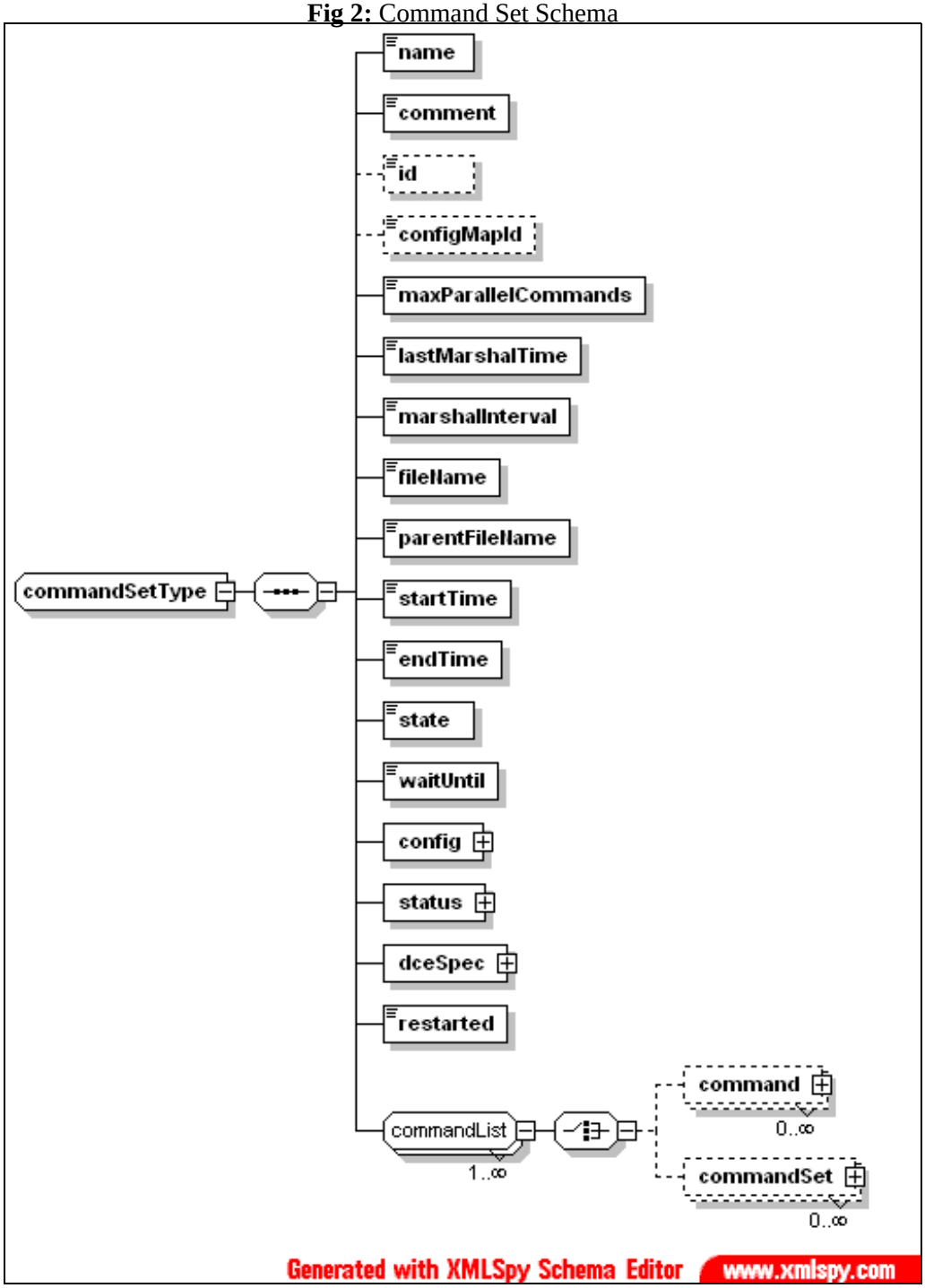
Name	Description	Data Type	Instance	Template	Notes/Valid Values
type	File type	String	required	required	*library*, *template*, *instance*
commandSet	The command set that models the workflow	**CommandSet**	required	required	only one set can be defined
host	The host on which the instance is running	String	required	-	This value is written by the engine when execution begins
port	The port on which the RMI server is listening	Integer	required	-	uses the default RMI port (1099)

3.2 CommandSet

A *commandSet* is a complex element that represents one aggregate unit of work. **Fig 2** shows the schema diagram for the command set element , the elements that are associated with the command set include *name, comment, type, version, configMapID, state, status, startTime, endTime, and the commandList*. A *commandList* contains one or more *command* and *commandSet* elements. When a command list contains other command sets, the workflow is considered to be a complex workflow built up from a hierarchy of sub-workflows.

The *dceSpec*element is used to specify grid parameters for a command set. For remote command sets this *dceSpec* is used to launch the job on the grid, for other command sets the *dceSpec* is used to specify grid params for any distributed commands contained within this set, unless overridden by the *dceSpec* of a specific command.

Fig 2: Command Set Schema



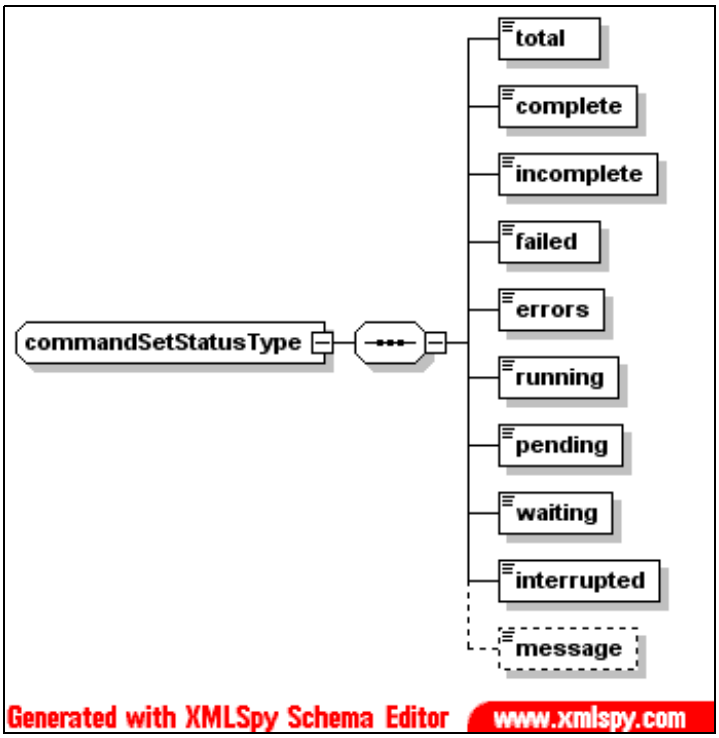
Name	Description	Data Type	Instance	Template	Notes/Valid Values
name	Command set name	String	required	required	
comment	A descriptive comment for the command set	String	optional	optional	
id	The unique identifier for this command set	Long	required	-	set by engine if none exists
maxParallelCmds	The maximum number of commands under this command set allowed to run at once	Integer	optional	optional	uses workflow default if none specified

configMapId	The config file mapping ID	String	required	-	
fileName	The filename for the file-based subflow	String	optional	optional	
parentFileName	The filename of the parent workflow	String	optional	-	set by engine during execution
startTime	Time the command set execution was started	DateTime	optional	-	created and set by engine if none exists
endTime	Time the command set execution was completed	DateTime	optional	-	created and set by engine if none exists
state	The current state of this command set	String	optional	-	created by engine if none exists. Valid values are: *incomplete, complete, error, running, failed, interrupted, pending*
waitUntil	Do not launch execution until the specified time	DateTime	optional	simple	**Not yet implemented**
config	The configuration parameters	**Config**	optional	optional	required for dynamic subflows
status	The status of this command set	**CommandSetStatus**	optional	-	created by engine if none exists
dceSpec	A DCE specification for distributed or remote command sets	**DCESpecification**	optional	optional	
commandList	The list of commands	List	required	required	*command* or *commandSet* elements
type	The type of command set	String	required	required	*serial, parallel, distributed-serial, distributed-parallel, remote-serial or remote-parallel*
version	The version number associated with this workflow	String	required	required	
restarted	This is used internally during command restarts	Boolean	-	-	Automatically generated

3.3 CommandSetStatus

This element is used to track the command set status and includes the total count of command elements in the set and the count of elements in different states. **Fig 3** shows the schema diagram for this element.

Fig 3: Command Set Status Schema



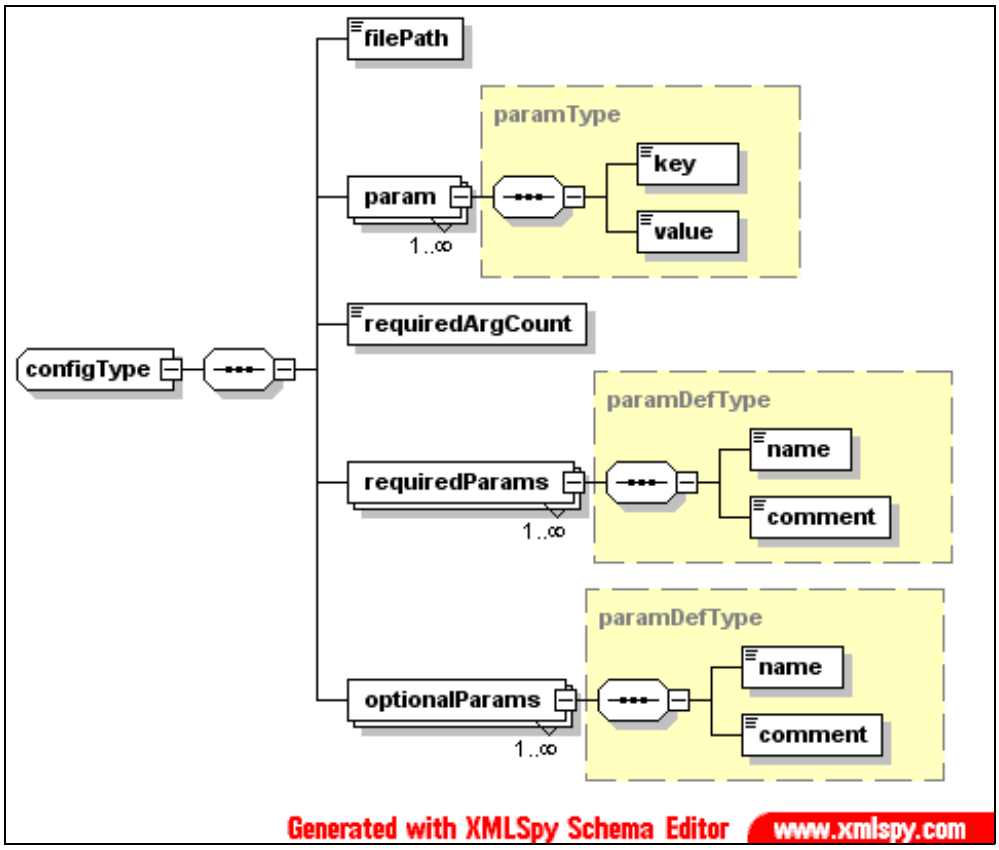
Name	Description	Data Type	Instance	Template	Notes/Valid Values
total	Count of all commands	Integer	required	required	
complete	Count of successfully completed commands	Integer	required	simple	
incomplete	Count of incomplete commands	Integer	required	simple	
failed	Count of failed commands due to exceptions	Integer	required	simple	
pending	Count of commands currently pending	Integer	required	simple	
errors	Count of commands that had non-zero return value	Integer	required	simple	
running	Count of commands currently running	Integer	required	simple	
waiting	Count of commands currently waiting	Integer	required	simple	
interrupted	Count of commands interrupted	Integer	required	simple	
message	The failure or interruption message if any	Integer	optional	simple	

3.4 Config

This complex element is used to specify the configuration for a *commandSet* or *command* elements. **Fig 4** shows the graphical representation of the config element. For command set elements the config element is used to specify parameters to build or execute the command set, for instance for dynamic file-based subflows, the *Param* element in config is used to specify the template and configuration file names that are used to build the instance file when needed. A param with the key *template* is used to specify the template file while the param with key *configfile* is used to specify the config file used to build the subflow instance.

Although not currently used, the command element can have a config elements such as the required argument count, required params, etc., associated with it. This will be used in the future for building GUI tools which will read this information to provide form fields for required and optional elements and to validate the specified input against the configured elements.

Fig 4: Config Schema

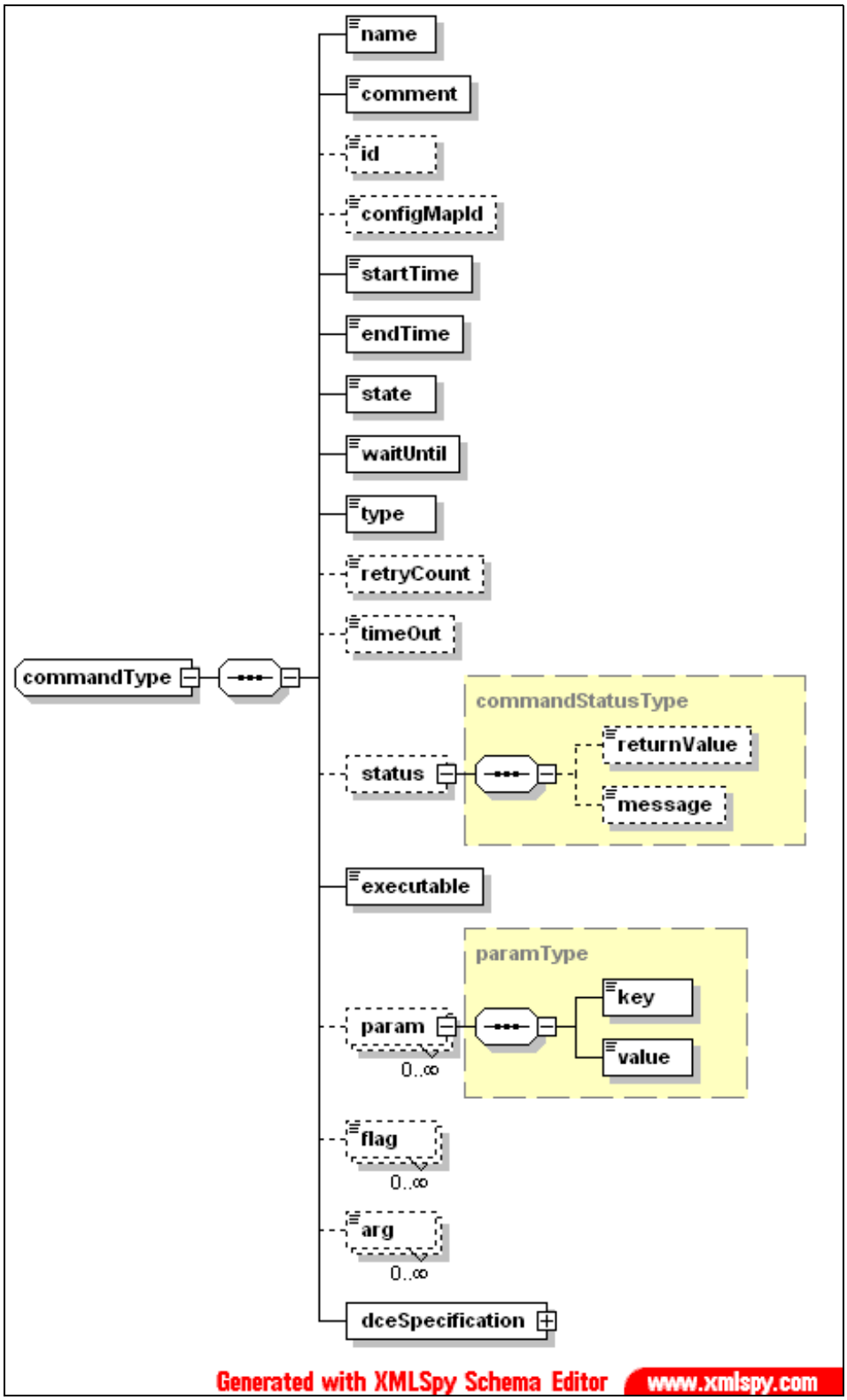


Name	Description	Data Type	Instance	Template	Notes/Valid Values
filePath	Configuration file to specify options to the processor	String	optional	optional	currently ignored
param	A parameters hash	**Param**	optional	optional	any, with special meaning attached to *template* and *configfile*.
requiredArgCount	Required argument count	Integer	optional	optional	currently ignored
requiredParams	Required parameter names and descriptions	**ParamDef**	optional	optional	currently ignored
optionalParams	Optional parameter names and descriptions	**ParamDef**	optional	optional	currently ignored

3.5 Command

A *command* element represents the smallest discrete sub-unit of work in a workflow. Each command is associated with a specific *CommandProcessor* that is invoked to execute this command. **Fig 5** shows the diagram for the command schema. The main elements of the command include the *name*, and *type* which are used to identify the particular command processor, *status, executable, and param* elements. There are other status elements such as *status, startTime, endTime, etc.* The *dceSpec* element is used to specify grid parameters if this is a distributed command. If this is a distributed command and no *dceSpec* is specified the *dceSpec* from the parent set is used if one is specified.

Fig 5: Command Schema



Name	Description	Data Type	Instance	Template	Notes/Valid Values
name	Command name used to identify processor	String	required	required	optional if the *type* is specified
comment	A descriptive comment for the command	String	optional	optional	
id	The unique identifier for this command	Long	required	-	set by engine if none present

configMapId	The config file mapping ID	String	optional	required	
startTime	Time the command set execution was started	DateTime	optional	-	created and set by engine if none exists
endTime	Time the command set execution was completed	DateTime	optional	-	created and set by engine if none exists
state	The current state of this command set	String	optional	-	*incomplete, complete, error, running, failed, interrupted, pending, waiting*
type	The type of command. An alternate key to lookup processor	String	required	required	optional if *name* is specified
waitUntil	Do not launch execution until the specified time	DateTime	optional	optional	**Not yet implemented**
retryCount	The number of times the system attempts to execute this command before giving up	Integer	optional	optional	
timeOut	The maximum time the command is allowed to execute before giving up	Integer	optional	optional	**Not yet implemented**
config	The configuration element	**Config**	optional	optional	
status	The status of this command	String	optional	optional	
executable	The name of the executable	String	optional	optional	
param	Parameters for this command	**Param**	optional	optional	
flag	The flags passed to this command	String	optional	optional	
arg	The arguments passed to this command	String	optional	optional	
dceSpec	The DCE specifications	**DCESpecification**	optional	optional	Valid only for distributed commands

3.6 CommandStatus

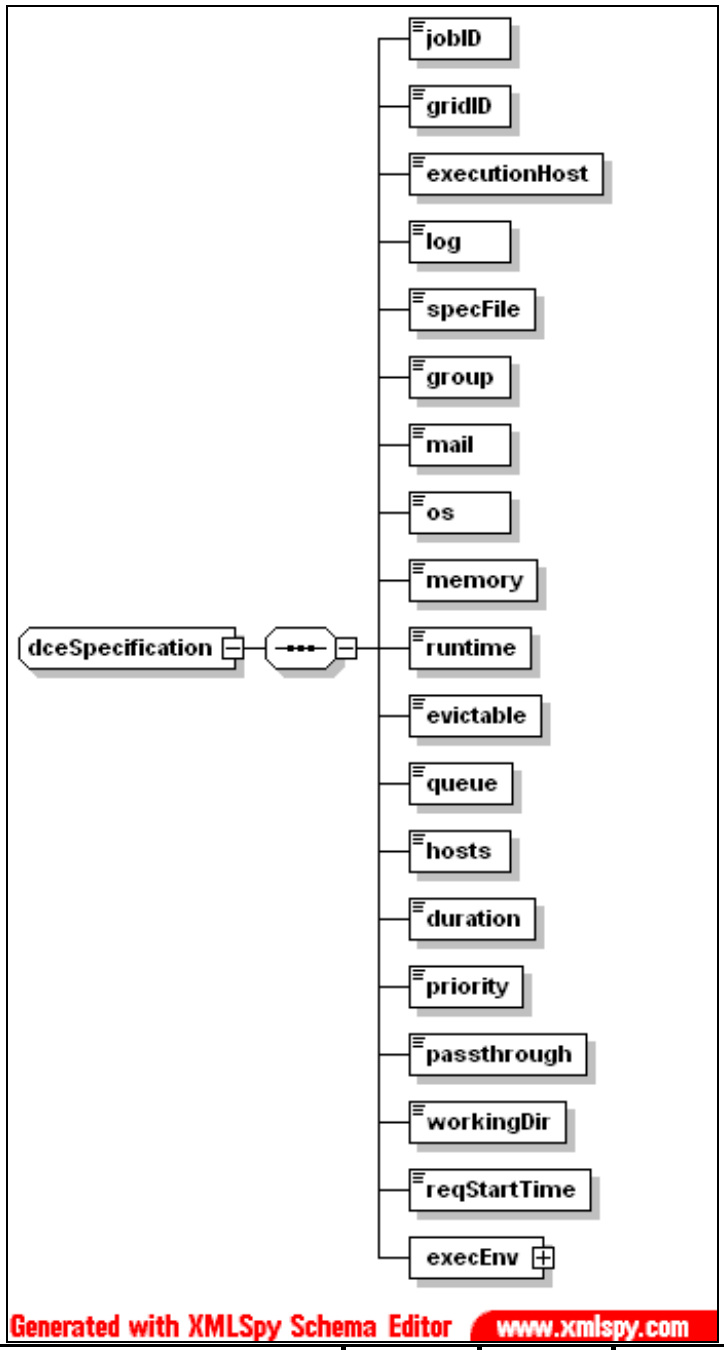
This element is used to track the status of a command and includes *returnValue, and message* as the main elements. See **Fig 5** for a graphical representation of the status schema.

Name	Description	Data Type	Instance	Template	Valid Values
returnValue	The return value of the command	Integer	-	-	set by the engine
message	A message string, typically error messages	String	-	-	set by the engine

3.7 DCESpecification

A *dceSpecification* element represents the distributed computing environment or grid specifications for a distributed command or set. A *dceSpecification* element specifies the parameters required for launching the job on the grid such as the *os, memory, hosts, priority, duration*, etc., as well as the run time values set by the engine such as the *gridID, executionHost*, etc. See **Fig 6** for a graphical representation of the *dceSpecification* element.

Fig 6: DCE Specification Schema



Name	Description	Data Type	Instance	Template	Notes/Valid Values
jobID	The jobID assigned by HTC API	Long	-	-	set by engine at runtime
executionHost	The host name which is executing the command	String	-	-	set by engine at runtime
log	The location to which *DCE log* messages are sent	String	-	-	set by engine at runtime
specFile	DCE parameters specifications file. Individually specified params will supersede params in file.	String	optional	optional	
group	Group name used for accounting and priority	String	required	optional	
mail	E-mail address for sending individual job completion notification	String	optional	optional	If not specified no notification is sent

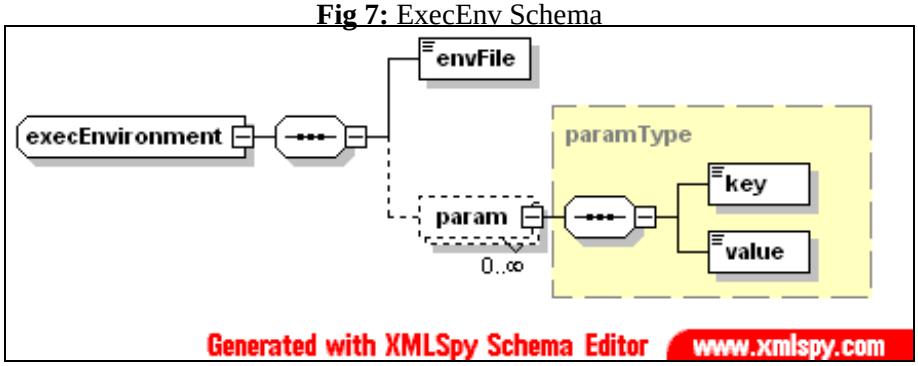
os	Operating systems on which jobs can run	String	optional	optional	CSV list which is *'or-ed'*. Default *linux* at TIGR
memory	Minimum memory requirements	String	optional	optional	Specified in megabytes, eg: 256M
runtime	Estimated running time	String	optional	optional	
evictable	Whether a job may be preempted and restarted	Boolean	required	optional	The default is true.
queue	An optional DCE queue to which job is sent	String	optional	optional	Ignored in condor, honored in SunGrid
hosts	List of hosts on which to try and execute the command	String	optional	optional	CSV list which is *'or-ed'*
duration	Estimated execution length of job	String	optional	optional	*short, medium, long, forever*. Default *medium*
priority	The priority of the command	String	optional	optional	*very low, low, medium, high, very high*
passthrough	Requirements string passed through to the underlying DCE	String	optional	optional	'*and-ed*' with other specifications if any specified
workingDir	Starting directory from where the job is launched	String	optional	optional	
reqStartTime	Actual execution start time.	DateTime	-	-	set by engine at runtime
type	The type of grid to use	String	required	optional	Default is SGE at TIGR.
gridID	The gridID for this job	String	-	-	set by engine at runtime
execEnv	Runtime environment	**ExecEnv**	optional	optional	Inherits user environment if none specified. If specified, replaces user environment

3.8 ExecEnvironment

The complex element *execEnvironment* is used to specify the runtime environment for executing a distributed command or set on the grid. There are two ways to specify the environment, through a file specified in the *envFile* element, or as a list *params* as *key*, *value* pairs that are set as the environment variables before execution. The default environment passed to the grid is the users shell environment, if this element is specified it will override the user's environment. See **Fig: 7** for a graphical representation of this element.

Name	Description	Data Type	Instance	Template	Notes/Valid Values
envFile	File to specify environment as key-value pairs	String	optional	optional	
param	One or more parameter objects	**Param**	optional	optional	

Fig 7: ExecEnv Schema



3.9 Param

This complex element is used to specify various parameters and their values. A param element typically is converted to a hash table of keys and values. When a particular parameter can have multiple values, the values can be specified a comma separated list, or multiple value elements sequentially.

Name	Description	Data Type	Instance	Template	Notes/Valid Values
key	The key for the parameter	String	required	required	
value	The value for this key	String	required	required	

3.10 ParamDef

This complex element is used to specify various parameter definition. A param def is used to specify the optional definition of required and optional parameters for command config elements.

Name	Description	Data Type	Instance	Template	Notes/Valid Values
key	The key for the parameter	String	required	required	
comment	The comment associated with this key	String	required	required	

4 Workflow Template File

A workflow template file is a skeleton XML file that defines the basic structure of the pipeline and conforms to the XML schema defined in *commandSet.xsd* (see the Workflow XML Schema section for details). The template file includes the typical pipeline invariant elements such as the name, type, fixed command parameters, etc. The template contains a *CommandSetRoot* root element. The root element should always have one and only one *CommandSet* element which represents the workflow. This command set may contain one or more *Command* class elements, which include, *CommandSet* and *Command*.

The following is an example of simple workflow template that contains two commands that are executed sequentially.

```
<?xml version="1.0" encoding="UTF-8"?>
<commandSetRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation='commandSet.xsd'>
  <commandSet type="serial">
    <name>myflow</name>
        <configMapId>1</configMapId>
        <command>
          <name>FirstCommand</name>
          <configMapId>1.1</configMapId>
        </command>
        <command>
          <name>SecondCommand</name>
          <configMapId>1.2</configMapId>
        </command>
  </commandSet>
</commandSetRoot>
```

In the template definition, a *CommandSet* must include the ***type***, ***name***, and ***configMapId*** elements. The *type* attribute can be one of *serial*, *parallel*, *distributed-serial*, *distributed-parallel*, *remote-serial* or *remote-parallel*. A *serial* type indicates that the commands within the set will be processed sequentially, while *parallel*, indicates that the commands will be processed simultaneously. Similarly, *distributed-serial* indicates that the commands within the set should be executed on the DCE sequentially, while *distributed-parallel* indicates that the commands of set should be executed on the DCE simultaneously. Sets declared to be remote-serial or remote-parallel will be launched on the grid as separate workflow instances. The *name* element identifies the descriptive name of the command set. The *configMapId* is a unique string for each command class element and maps to a section of the configuration file that has the parameters required for instantiating this element.

The following is an example of a file-based hierarchical template that contains one command and a command set that should be executed simultaneously. The subflow template and configuration file are specified in the *config* element of the command set while the subflow file name is specified in the *fileName* element.

Parent Template	Child Template
<?xml version="1.0" encoding="UTF-8"?> <commandSetRoot> <commandSet type="parallel"> <name>myflow</name> <configMapId>1</configMapId> <command> <name>RunUnixCommand</name> <configMapId>1.1</configMapId> </command> <commandSet type="serial">	<?xml version="1.0" encoding="UTF-8"?> <commandSetRoot> <commandSet type="serial"> <name>sub-workflow</name> <configMapId>1.2</configMapId> <command> <name>RunUnixCommand</name> <configMapId>1.2.1</configMapId> </command> </commandSet> </commandSetRoot>

```
      <name>sub-workflow</name>
      <configMapId>1.2</configMapId>
      <config>
        <param>
          <key>template</key>
          <value>subflow_templ.xml</value>
        </param>
        <param>
          <key>configfile</key>
          <value>subflow_templ.ini</value>
        </param>
      </config>
      <fileName>subflow.xml</fileName>
    </commandSet>
  </commandSet>
</commandSetRoot>
```

5 Configuration File

A configuration file or config file is an INI style file that complements a template file and is used by the workflow builder to create a workflow instance file. This file typically specifies values for workflow elements that change between to workflow invocations, such as file names, parameters, etc. The file contains multiple sections identified by unique section headers with names that map to the *configMapId* of the workflow template. Within each section, the parameters are specified as *key-value* pairs, one per line. The equals sign (=) separates the *key* and its *value*. Multiple values can be specified as a comma-separated list or as multiple occurrences of the key in that section.

The workflow builder uses the elements in template and configuration file to build the instance file element. When both the template and configuration file specify a value for an element, the config file values are considered to be additive and appended to the list of values specified in the template. However, in instances where there can only be one value for an element, such as the *executable*, *name*, or *type*, the config file value supersedes the template file value.

The config file is also the place where command or command-set iterators can be specified. Iterators cause the replication of commands or command sets with different parameters and are useful when the same operations are performed iteratively on a list of files, etc. See the Iterators section for details.

The table below shows an example of a template file and its corresponding configuration file:

Template File	Configuration File
<?xml version="1.0" encoding="UTF-8"?> <commandSetRoot> <commandSet type="serial"> <name>myflow</name> <configMapId>1</configMapId> <command> <name>FirstCommand</name> <configMapId>1.1</configMapId> <executable>yank.pl</executable> </command> <command> <name>SecondCommand</name> <configMapId>1.2</configMapId> <executable>ls</executable> </command> </commandSet> </commandSetRoot>	[1] name=myflow config.filePath=hello [1.1] name=FirstCommand param.--conf=my.conf param.--outfile=my.out [1.2] name=SecondCommand arg=/home/amahurka

5.1 Simple Values

For simple XML elements such as *name* and *retryCount* that take a single value, the key in the config file maps to the name of the XML element, and the value associated with the key is assigned as the value of the XML element. For example, if a command in the template file contains an XML element *name* with value *FirstCommand*, the entry in the configuration file would look like *name=FirstCommand*.

5.2 Complex Values

For complex elements like *Param* or *Config* (see examples below) which contain one or more sub-elements, the key for an XML element should contain a "dot" (.)-separated path that includes all the XML elements in the tree path. For a *Param* element with key *'--in'* and value *'infile'* in the example, the entry in the config file would be param

--in=infile. For example the entry for *Config* param *template* with value *subflow_templ.xml* the entry in the config file would be *config.param.templateFile=subflow_templ.xml*. If a key has more than one value, the values can be specified as a comma-separated list or as multiple sequential instances of the value line.

```
<param>
  <key>--in</key>
  <value>infile</value>
</param>

<config>
  <filePath></filePath>
  <param>
    <key>template</key>
        <value>subflow_templ.xml</value>
  </param>
</config>
```

The following table shows example configuration files for building hierarchical workflow. The top section shows the template and config file for the parent, while the second row shows the template and config file for the child flow.

Template File	Config File
<?xml version="1.0" encoding="UTF-8"?> <commandSetRoot> <commandSet type="parallel"> <name>myflow</name> <configMapId>1</configMapId> <command> <name>RunUnixCommand</name> <configMapId>1.1</configMapId> </command> <commandSet type="serial"> <name>sub-workflow</name> <configMapId>1.2</configMapId> <config> <param> <key>template</key> <value>subflow_templ.xml</value></param> </config> <fileName>subflow.xml</fileName> </commandSet> </commandSet> </commandSetRoot>	[1] ; Top level command set name=myflow [1.1] ; First command name=RunUnixCommand param.--config=yank.conf executable=yank.pl [1.2] ; A sub-workflow specified as a command set name=sub-workflow
<?xml version="1.0" encoding="UTF-8"?> <commandSetRoot> <commandSet type="serial"> <name>sub-workflow</name> <configMapId>1.2</configMapId> <command> <name>RunUnixCommand</name> <configMapId>1.2.1</configMapId> </command> </commandSet> </commandSetRoot>	[1.2] ; Sub-workflow command set name=sub-workflow [1.2.1] ; First command name=RunUnixCommand executable=ls arg=/home/user

5.3 Iterators

In most computational pipelines we encounter situations where a single command or an entire pipeline is run on a collection of data repeatedly. For instance, to perform an all-vs-all BLAST search for a set of genomes the same blast pipeline is run once per organism. Within the pipeline one might break the data set into smaller chunks, say one per chromosome, and run the BLAST search per cheroots. Instead of manually creating instances with these repeating command sets and commands, workflow provides a mechanism for using an existing pipeline definition and iterating over the entire data set using this pipeline. For instance the entire pipeline might include the execution of the BLAST pipeline once per organism where in the organism pipeline we split the organism data into smaller data sets based on chromosomes, or some other logical unit, and then run BLAST search for each of the units. The first type of iteration would be a command set iteration while the second type of iteration is considered a command iteration. These two types of iterators, command iterators and command-set iterators, are discussed in this section. Iteration is accomplished within workflow by specifying the iterator parameters in the configuration files with special element names. These are used by the builder to elaborate the iteration to specific commands and sets.

5.3.1 Command Iterator

A command iterator can be used to iterate over a list of parameters and execute a specific command for each of the items in the list. For instance in the above example we iterate over the individual chromosome files to run BLAST search on each of the files. Instead of repeating that many commands in template and config files, a template file can simply specify the single BLAST command, and the config file can use an iterator parameter to specify the list of chromosome files over which the BLAST command iterates.

The following table shows an example of a workflow template and corresponding config file with an iterator command:

Template File	Config File
<?xml version="1.0" encoding="UTF-8"?> <commandSetRoot> <commandSet type="parallel"> <name>Blast Proteins</name> <configMapId>1</configMapId> <command> <name>Blast</name> <configMapId>1.1</configMapId> <executable>blastp</executable> </command> </commandSet> </commandSetRoot>	[1] ; Top level command set name=Blast Proteins config.filePath=hello [1.1] ; First command name=Blast command.iterate = param.-i, param.-o param.-i=first.chr, second.chr param.-o=first.chr.out, second.chr.out param.-d=AllGroup.niaa

There are two requirements for specifying command iterators. The first is that the command section in the config or template file (section 1.1 in the above example) have an entry with the key '**command.iterate**' to indicate that this command is an iterator command. The value associated with this key is a comma-separated list of iterator parameters (*param.-i, param.-o* in the example). The second requirement is that the iterator parameters specify the list of values to iterate over (Note: The number of items in the iteration list should be the same for each of the iterator parameters). In this case we want to iterate over two chromosome (*first.chr, second.chr)* files and write the output to two files (*first.chr.out, second.chr.out*).

The major elements of the resulting instance file are shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<commandSetRoot>
```

```
  <commandSet type="parallel">
    <name>Blast Proteins</name>
    <configMapId>1</configMapId>
    <command><name>Blast</name>
      <configMapId>1.1</configMapId>
      <id>12344</id>
      <executable>blastp</executable>
      <param>
        <key>-i</key>
        <value>first.chr</value>
      </param>
      <param>
        <key>-o</key>
        <value>first.chr.out</value>
      </param>
      <param>
        <key>-d</key>
        <value>AllGroup.niaa</value>
      </param>
    </command>
    <command><name>Blast</name>
      <configMapId>1.1</configMapId>
      <id>12345</id>
      <executable>blastp</executable>
      <param>
        <key>-i</key>
        <value>second.chr</value>
      </param>
      <param>
        <key>-o</key>
        <value>second.chr.out</value>
      </param>
      <param>
        <key>-d</key>
        <value>AllGroup.niaa</value>
      </param>
    </command>
  </commandSet>
</commandSetRoot>
```

When specifying a list of arguments or flags to iterate over, a position number is appended to the *arg* or *flag* parameter. For instance if the the first argument is the input file and the second argument is the output file, the section map looks as follows:

```
[1.1]
; First command
name=MyProcess
command.iterate = arg.1, arg.2
arg.1 = first.in, second.in
arg.2 = first.out, second.out
```

5.3.2 Command-Set Iterator

A command-set iterator can used to iterate over a list and execute a specific sub-flow for each item in the list. For instance in the above discusses example we would have one command set or workflow for each of the organisms. Instead of repeating that many command sets in template and config files, a template file can simply specify the single command set, and the config file can use a command-set iterator to specify the list of files over which the command set iterates.

The following table shows an example of a workflow template and corresponding config file with command set iterators:

Template File	Config File
<?xml version="1.0" encoding="UTF-8"?> <commandSetRoot> <commandSet type="parallel"> <name>My Iterative Blast Pipeline</name> <configMapId>1</configMapId> <commandSet type="serial"> <name>OrganismSubflow</name> <configMapId>1.1</configMapId> <config> <param> <key>template</key> <value>org_subflow_template.xml</value> </param> </config> </commandSet> </commandSet> </commandSetRoot>	[1] ; Top level command set name=My Iterative Blast Pipeline [1.1] ; Subflow name=OrganismSubflow commandset.iterate=fileName, config.param.configfil config.param.configfile=h_sapiens.ini, b_taurus.ini fileName = h_sapiens.xml, b_taurus.xml

There are two requirements for specifying command-set iterators. The first is that the command-set section in the config or template file (section 1.1 in the above example) have a parameter with the name **'commandset.iterate'** to indicate that this command set is an iterator command set. The value associated with this entry is a comma-separated list of iterator parameters (*config.param.configfile*, *fileName* in the example). The second requirement is that the iterator parameters specify the list of values to iterate over (Note: The number of items in the iteration list should be the same for each of the iterator parameters). In this case we want to iterate over two organisms, *h_sapiens*, and *b_taurus*, the config files (*h_sapiens.ini*, *b_taurus.ini*) contain the parameters that are used to generate the organism pipelines (*h_sapiens.xml*, *b_taurus.xml*).

The major elements of the resulting pipeline instance file are shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<commandSetRoot>
  <commandSet type="parallel">
    <name>My Iterative Flow</name>
    <configMapId>1</configMapId>
    <id>1</id>
    <commandSet type="serial">
      <name>MySubflow</name>
      <configMapId>1.1</configMapId>
      <id>2</id>
      <fileName>h_sapiens.xml</fileName>
      <config>
        <param>
          <key>template</key>
          <value>org_subflow_template.xml</value>
        </param>
        <param>
          <key>configfile</key>
          <value>h_sapiens.ini</value>
        </param>
      </config>
    </commandSet>
    <commandSet type="serial">
      <name>MySubflow</name>
      <configMapId>1.1</configMapId>
```

```
      <id>3</id>
      <fileName>b_taurus.xml</fileName>
      <config>
        <param>
          <key>template</key>
          <value>org_subflow_template.xml</value>
        </param>
        <param>
          <key>configfile</key>
          <value>b_taurus.ini</value>
        </param>
      </config>
    </commandSet>
  </commandSet>
</commandSetRoot>
```

6 Dynamic Workflows

Workflow system allows the creation and execution of dynamic workflows. Dynamic workflows in this instance are defined as workflows that cannot be elaborated or fully defined at the onset of a pipeline execution but become defined as the process progresses. In a future release we may add the ability for branching based on conditions, etc.

For instance in the example of an all-vs-all search discusses in the previous section, at the outset we may not know how many organisms exist in our data collection, or within an organism how many chromosomes exist. One possibility would be for a pre-process step to do all of this identification and build the necessary configuration files, but another possibility is for one step of the pipeline to create these files as the need arises. So, at the top level there may be a step that identifies the number of organisms and build the organism subflow config files. Once the execution of the pipeline for an organism begins the first step might be to split the data file based on organisms and build the config file to iterate over the chromosomes. To support the second possibility, the workflow engine can be instructed to delay the creation of instance files till the point of execution. While there is no right way of doing it the latter approach in our experience has proven to be a more scalable and elegant approach.

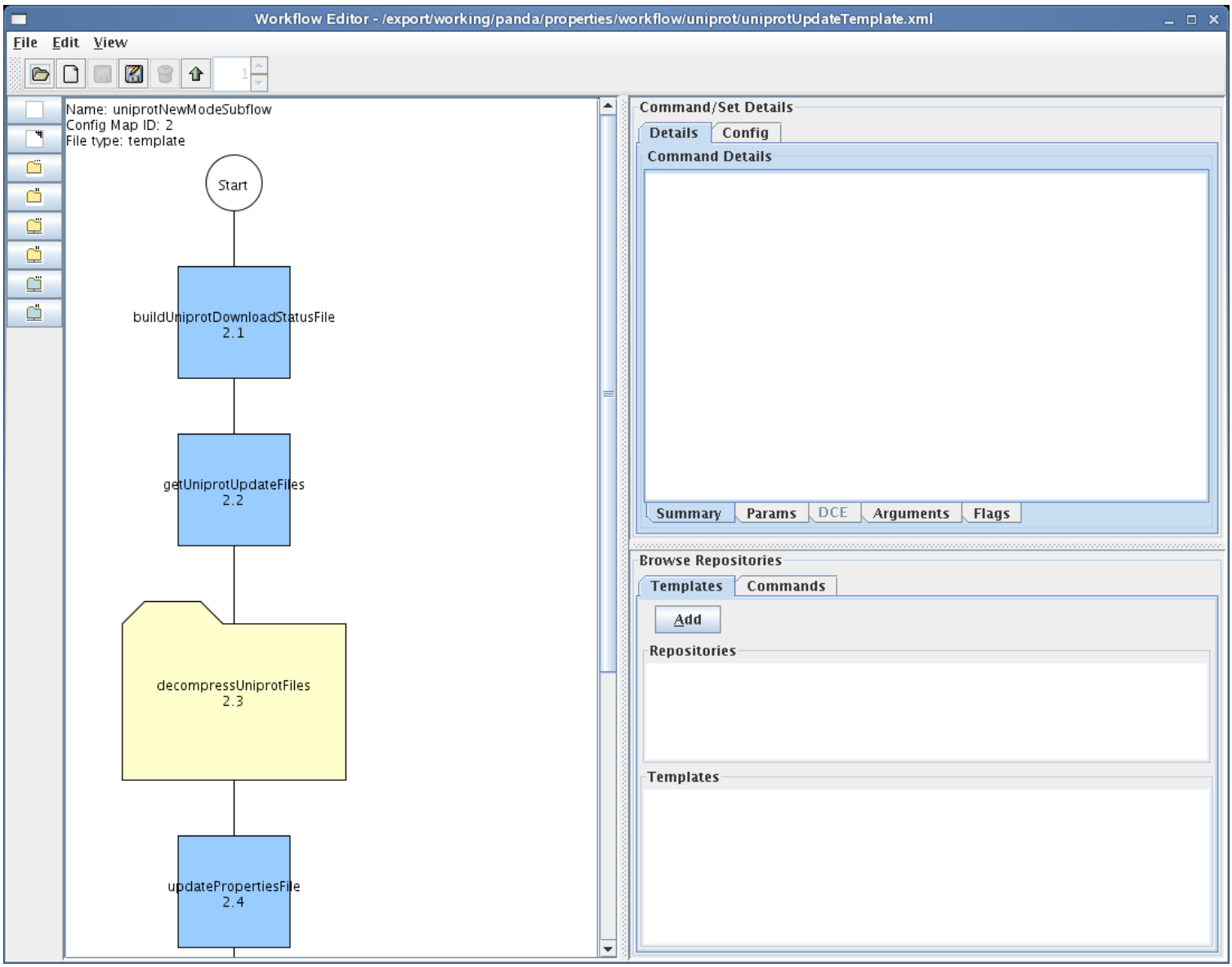
7 Tools

This section discusses the various workflow command-line and GUI tools used to build, manage, monitor, and execute workflows. This section describes the tools, their invocation, command line options and some example invocations.

7.1 *Template Editor*

This GUI tool allows the user to view, create, edit, and print workflow templates. Additionally, this tool can be used to create initial config files associated with the templates. The editor can be used in two modes, the *Single Level* mode in which a single level of the template file can be viewed and edited, and the *Multi Level* mode in which the template can be viewed at an arbitrary depth level. The screen shot of the Template Editor screen shown in Fig 8 displaying a template in the *Single Level* mode. The commands in the workflow are represented as rectangles while the command sets are represented by a folder. The figure shows a workflow with three commands and a command set that is visible.

Fig 8: Template Editor - Single Level Mode



7.1.1 Single Level View

This section discusses the main layout of the editor in the single level view in which templates can be viewed, edited, and created. The GUI in the single level view contains three areas, the **Display Pane** on the left, which shows the graphical representation of the template, the **Details Pane** on the top-right hand side which shows the details for the element currently selected in the display pane, and the **Repository Pane**, at the bottom-right that shows the selected templates and commands in the repositories. Along the left hand side of the GUI is the **Tool Bar** which includes buttons for adding command or command set elements to the template, and along the top is the **Menu** and the **Menu Bar** with the most commonly used menu items.

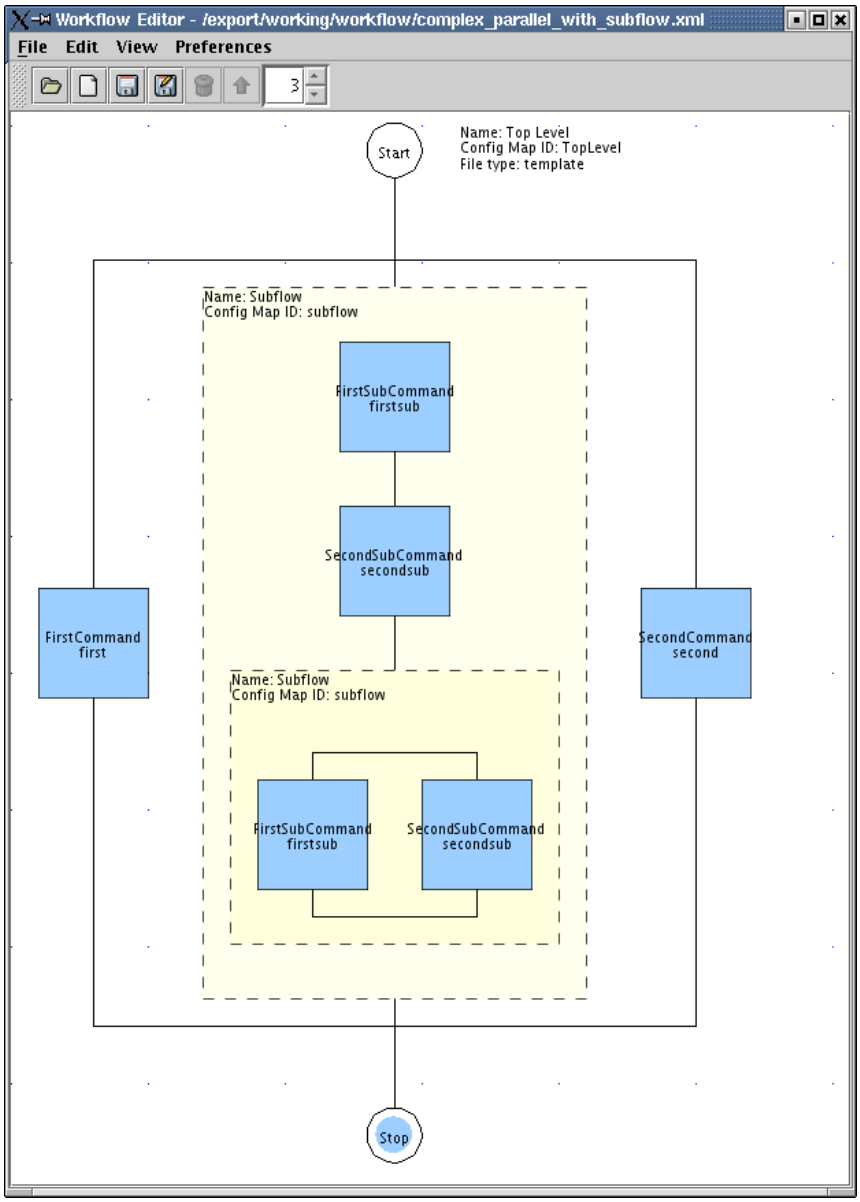
7.1.1.1 Move Up and Down Hierarchy

In the *Single Level Mode* only one level of a workflow hierarchy is displayed. To move down the hierarchy double-click on the desired command set represented as a folder. This will switch the view to display the contents of the selected command set. To move up the hierarchy click on the **Move Up** icon. This will switch the view to the parent of the currently displayed command set.

7.1.2 Multi Level View

To view the template and its subflows in an expanded view switch to the *Multi Level View*. In this view the user can choose a depth level to which the template is expanded. Each depth level is displayed in a darker shade and is surrounded by a dotted line. Fig 9 shows the expanded view of a template expanded to the third depth level. The multi level mode is very useful to see the overall structure of a pipeline and can be used to document the pipeline.

Fig 9: Template Editor - Multi Level Mode



7.1.2.1 Change Depth Level

To change the depth of the expanded view change the depth level in the **Spinner** control in the Menu Bar. This will change the depth to which the template is displayed. Currently the maximum depth to which a template can be expanded is 16.

7.1.3 Menus

This section discusses the main menu items in the three main menu categories, **File, Edit,** and **View.** The menu items are organized by category with a brief description of each of the menu items.

7.1.3.1 File Menu

The file menu contains the menu items for creating, opening, closing, saving, and printing the templates. Each option in this section is discussed below.

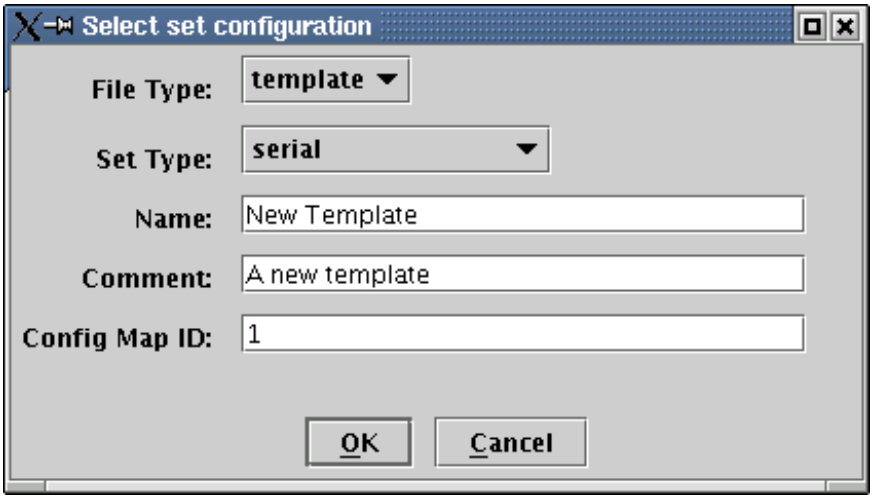
7.1.3.1.1 Open

To open an existing template select the *File -> Open* menu item. Alternately, the user can click on the icon, or use the keyboard shortcut *Ctrl-O,* to open an existing template file. This brings up a *Open File Dialog* which can be used to navigate and select the desired template file. When this menu item is selected any open template file is closed and displays the contents of the selected template in the Display Pane.

7.1.3.1.2 Create New

To create a new *template* or *library* select the *File -> New* menu item. Alternately, the user can click on the icon, or use the keyboard shortcut *Ctrl-N,* to create a new template. This brings up the *Select Set Configuration Dialog* shown in Fig. 10 which can be used to specify the type of the file *template* or *library*, and other initial values. When the user clicks on the ***OK*** button, this closes any open template and displays the new template in the Display Pane. A *library* is a special template file that contains a collection of commands that can be reused to build templates.

Fig 10: Template Editor - Select Set Configuration Dialog



7.1.3.1.3 Save

To save an open template click the *File -> Save* menu item. Alternately the user can click on the icon, or use the keyboard shortcut *Ctrl-S,* to save the file. If this is a new file the *Save File Dialog* is displayed which can be used to navigate and select the desired template filename. This selection only saves the current file. If this file is a subflow of a parent flow, the parent flow is not saved.

7.1.3.1.4 Save As

To save an open template to a different file click the *File -> Save As* menu item. Alternately the user can click on the icon, or use the keyboard shortcut *Ctrl-A,* to save the file. The *Save File Dialog* is displayed which can be used to navigate and select the desired template filename. This selection only saves the current file. If this file is a subflow of a parent flow, the parent flow is not saved.

7.1.3.1.5 Save All

To save all modified files select the *File -> Save All* menu item. Alternately the user can use the keyboard shortcut *Ctrl-L,* to save the file. If this is a new file the *Save File Dialog* is displayed which can be used to navigate and select the desired template filename. This selection recursively saves all new and modified template files starting from the top-level template file.

7.1.3.1.6 Print

To print the the template select the *File -> Print* menu item. Alternately, the user can use the keyboard shortcut *Ctrl-P*, to print the set. Based on the current view this will cause the current command set or the expanded view of the workflow to the selected depth to be printed to the selected printer.

7.1.3.1.7 Create Configs

To create base config files associated with template files, or update existing config file, select the *File -> Create Configs* menu item. Alternately, the user can use the keyboard shortcut *Ctrl-G*. This will recursively create or update config files associated with template files. If no config file name is specified, a name is derived from the template file name by replacing the *'xml'* extension with *'ini'* extension. The config element of the command set is modified to reflect the config file associated with the template. All pre-defined parameters are added to the config file along with the name and the section header. As this may cause the template files to be modified, the user must save all the changes to template file after this operation.

7.1.3.1.8 Close

To close the current top-level template select the *File -> Close* menu item. Alternately the user can use the keyboard shortcut *Alt-C*, to close template. This will close the current open template, if there an unsaved changes the user is presented with the option to abort the close operation so the user can save the changes before closing.

7.1.3.1.9 Exit

To exit the application select the *File -> Exit* menu item. Alternately the user can use the keyboard shortcut *Alt-X*, to exit. This will close the application, if there an unsaved changes the user is presented with the option to abort the exit operation so the user can save the changes before exiting.

7.1.3.2 Edit Menu Items

This section discusses the edit menu items of the editor that include the functions to delete, copy, and paste elements of the workflow template. This menu is only enabled in the *Single Level* mode as editing is disabled in the *Multi Level* mode.

7.1.3.2.1 Delete

To delete an element from the current template select the element and then choose *Edit -> Delete* menu item. Alternately the user can click on the  icon. This will delete the selected element from the current template.

7.1.3.2.2 Cut

To cut an element from the current template into the clipboard select the element in the display pane and choose *Edit -> Cut* menu item. Alternately the user can use the keyboard shortcut *Ctrl-X*. This will move the selected element from the current template to the application clipboard.

7.1.3.2.3 Copy

To copy an element from the current template into the clipboard select the element in the display pane and choose *Edit -> Copy* menu item. Alternately the user can use the keyboard shortcut *Ctrl-C*. This will copy the selected element from the current template to the clipboard.

7.1.3.2.4 Paste

To paste an element from clipboard to the current template choose *Edit -> Paste* menu item. Alternately the user use the keyboard shortcut *Ctrl-V*. This will paste the element from the clipboard to the end of the current template.

7.1.3.2.5 Move

To move the location of an element select the element and then drag and drop the element to the new location. This will rearrange the elements in the command set.

7.1.3.3 View Menu Items

This section discusses the menu items on the **View Menu** that include items to zoom in and out and switch the views.

7.1.3.3.1 Single Level

To switch the view to single level mode select *View -> Single Level* menu item. Alternately the user can use the keyboard shortcut *Alt-S*. This will switch the current view to the single level mode where the template can be edited.

7.1.3.3.2 Multi Level

To switch the view to multi level mode select *View -> Multi Level* menu item. Alternately the user can use the keyboard shortcut *Alt-M*. This will switch the current view to the multi level mode where the template can be viewed in an exploded view at an arbitrary depth (Note: This view disables editing functions of the editor).

7.1.3.3.3 Zoom In

To zoom in or increase the magnification of the current view select *View -> Zoom In* menu item. Alternately the user can use the keyboard shortcut *Ctrl*-=. This will increase the magnification of the view by 20%.

7.1.3.3.4 Zoom Out

To zoom out or decrease the magnification of the current view select *View -> Zoom Out* menu item. Alternately the user can use the keyboard shortcut *Ctrl*--. This will decrease the magnification of the view by 20%.

7.1.3.3.5 Zoom Level

To select a specific zoom level from a set of predefined zoom levels select the zoom level under the *Zoom Level* submenu. This will change the magnification of the template to the specified magnification level.

7.1.4 Edit Template

This section discusses the essential editing capabilities of the template editor. The section is organized around the major functions that one can perform such as adding commands, command sets, etc.

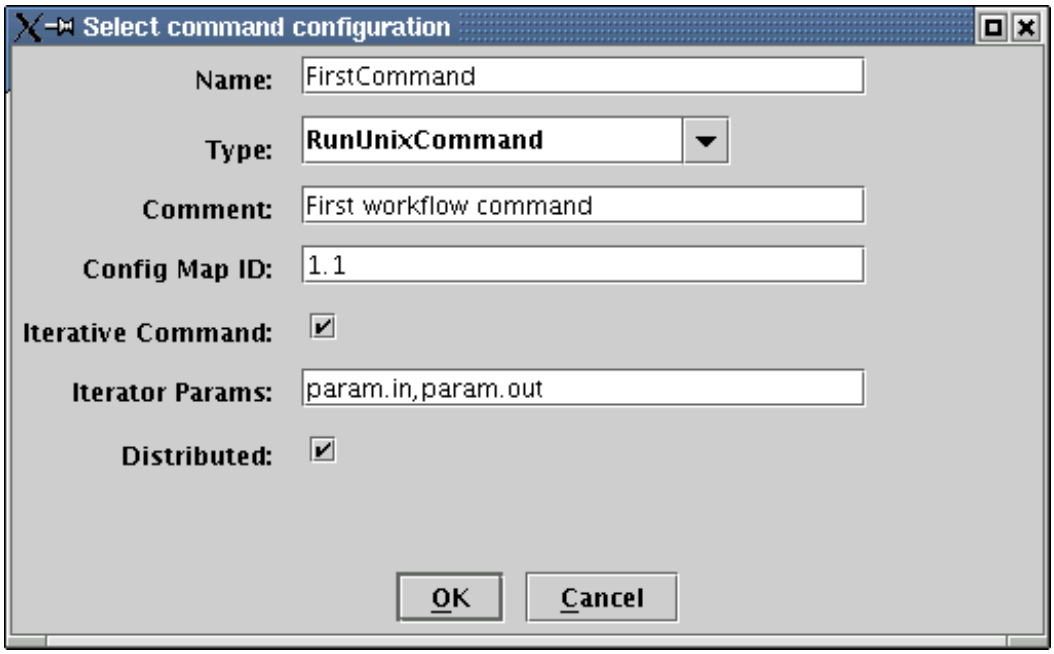
7.1.4.1 Add Command

To add a new command to the template drag and drop one of the following command icons from the **Tool Bar** to the desired location in the display panel.

 - Local Command
 - Distributed Command

This brings up a *Select Command Configuration Dialog* shown in Fig. 11. The user can enter the **Name** of the command, select a **Type** of command from the drop down list (generated by reading the standard and custom command processor lookup files), the optional **Comment**, and the required **Config Map ID** fields**.** If this is an iterative command the user can click on the **Iterative Command** check box. This enables the **Iterator Params** text field where the user can enter a comma separated list if the iterator parameter names. Fig. 11 shows that the new command is an iterator command that has two iterator params, *param.in* and *param.out.* Finally, the user can click on the **Distributed** check box to specify that this is a distributed command. When the user clicks on the ***OK*** button, the new command is inserted at the specified location.

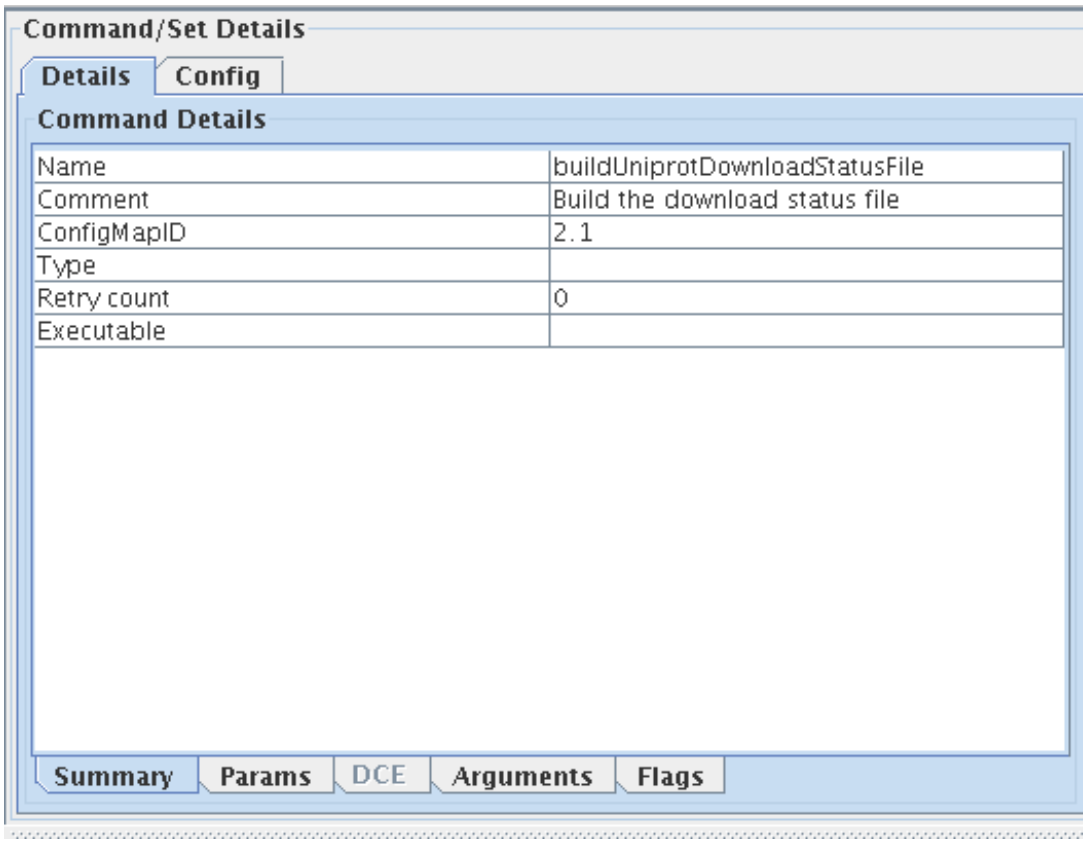
Fig 11: Template Editor - Select Command Configuration Dialog



7.1.4.2 Edit Command Details

To edit the details of a command, select the command in the Display Panel. This will select the command and the details of the command are displayed in the details pane where they can be edited. To edit the command summary click on the **Summary** tab and edit the appropriate fields. See Fig. 12 to see the summary fields which include the name, comment, configMapID, type, retry count, and executable name of the command that can be edited.

Fig 12: Template Editor - Command Summary Editing

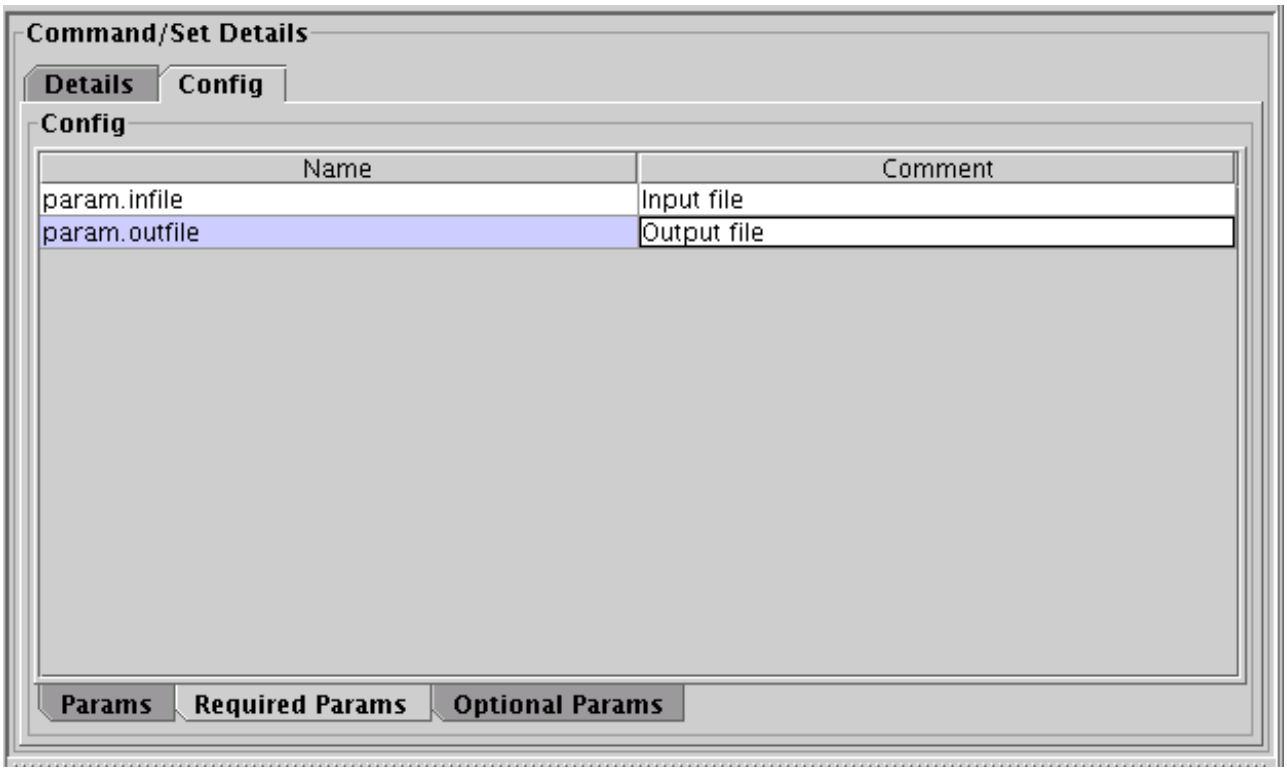


To edit the parameters of a command click on the **Params** tab and add the key value pairs to the list. For distributed commands the DCE specification values can be specified in the **DCE** tab. The arguments, and flags of a command command can be specified in the **Arguments** and **Flags** tabs.

7.1.4.3 Edit Command Config

To edit the command configuration, select the command in the Display Panel. This will display the details of the command in the details pane. To edit the command config click on the **Config** tab which will show the three types of config parameters that can be edited for commands. See Fig. 13 to see the summary fields of the command that can be edited. (Note: Currently workflow engine and builder ignores the command config element)

Fig 13: Template Editor - Command Config Editing

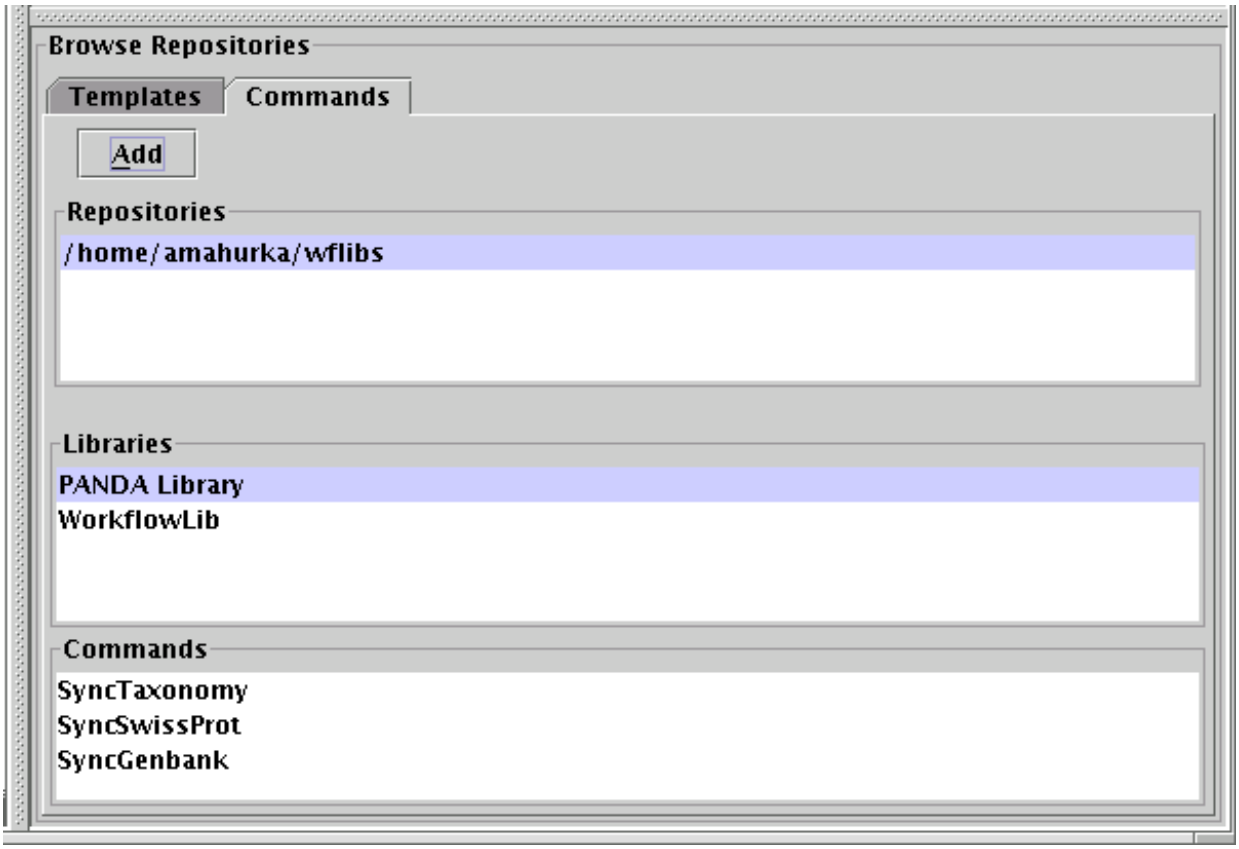


To edit the parameters of a command click on the **Params** tab and add the key value pairs to the list. The required params and optional params can be specified in the **Required Params** and **Optional Params** tabs.

7.1.4.4 Add Command from Commands Library

Another way of adding commands to a template is to choose a command from a pre-existing library of commands. A pre-existing library is a special template file marked as *library* that contains a series of command definitions. To use a library, select the **Commands** tab in the **Browse Repositories** pane of the editor and click on the **Add** button. This will bring up a file dialog which can be used to navigate and select the directory that contains the library files. The editor parses all the files in the specified directory to identify library files and adds the libraries to the library collection. The repository is listed in the **Repositories** list, the libraries in this repository are added to the **Libraries** list, and the commands in a selected library displayed in the **Commands** list. Fig. 14 shows the commands in the *PANDA Library* in the repository */home/amahurka/wflibs.*

Fig 14: Template Editor - Select Library Command



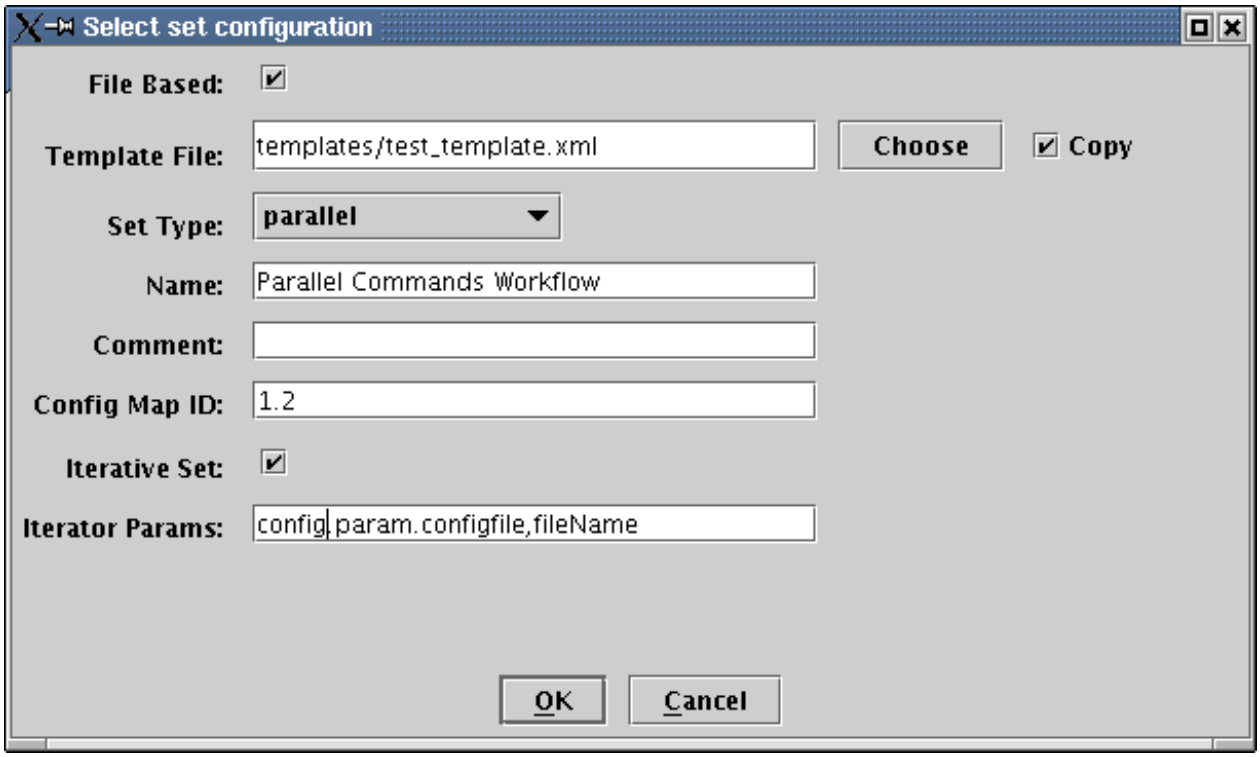
To add a command from the commands list, drag and drop one of the commands on to the *Display Pane* at the desired location. A copy of the selected command is made and the *Select Command Configuration Dialog* shown in Fig. 11 filled with values from the command is displayed. The user can either accept the values or make necessary changes and click on the **OK** button to add the command to the template.

7.1.4.5 Add Command Set

To add a new command set to the template drag and drop one of the following command set icons from the **Tool Bar** to the desired location in the display panel.

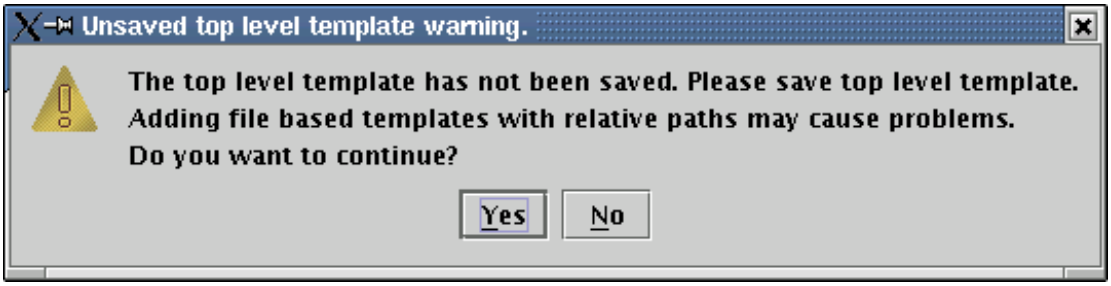
- Serial Command Set
- Parallel Command Set
- Distributed Serial Command Set
- Distributed Parallel Command Set
- Remote Serial Command Set
- Remote Parallel Command Set

Fig 15: Template Editor - Select Command Set Configuration Dialog



This brings up a *Select Set Configuration Dialog* shown in Fig. 15. If the new command set is a file based command set click on the **File Based** check box. This enables the **Template File** text field, **Choose** button**,** and **Copy** check box. If the top level template has not been saved before and the user attempts to add a file based subflow, a warning dialog box pops up to warn the user that relative path names cannot be used until the top level template is saved. (See Fig. 16) Although the user is given the option to continue (in which case fully qualified path names are used for subflows), it is highly recommended that the user save the top level template before attempting to add file based subflows as this enables relative path names making the workflows more portable.

Fig 16: Template Editor - Unsaved Top Level Template Dialog



The user can either choose an existing template file or create a new one by entering the name of the subflow file in the **Template File** field, or click on the **Choose** button to navigate the file system to select the template file. If the file is an existing file and resides in a separate directory the user can also check on the **Copy** check box to instruct that the template file be copied from the original location to the current location of the top level template file.

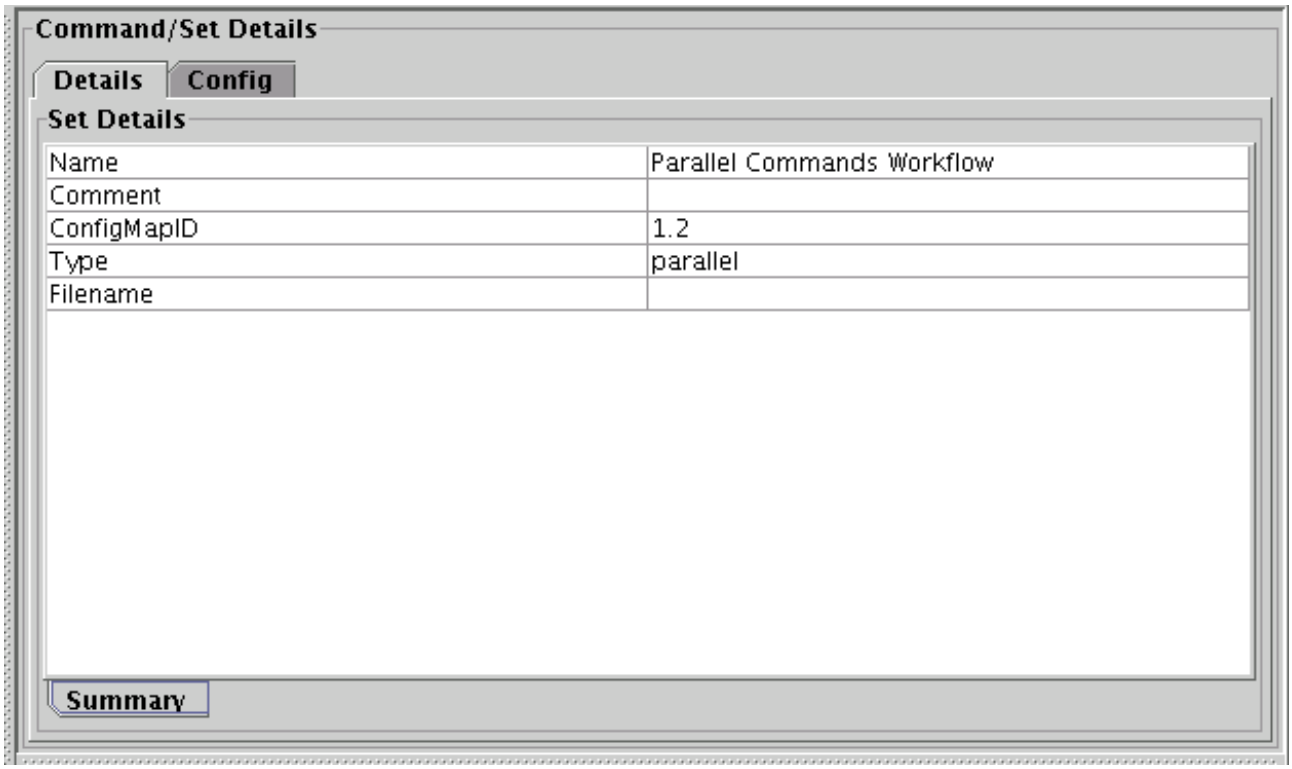
The user can select command set **Type** of command from the drop down list, enter a **Name**, the optional **Comment**, and the required **Config Map ID** fields in the dialog box**.** If this is an iterative command set the user can click on the **Iterative Set** check box which enables the **Iterator Params** text field where the user can enter a comma separated list if the iterator parameter names. Fig. 14 shows that the new set is an iterator set that has two iterator params, *config.param.configfile* and *fileName.* When the user clicks on the ***OK*** button, the new set is inserted at the specified

location.

7.1.4.6 Edit Set Details

To edit the details of a set select the set in the *Display Pane*. This will select the set and the details of the set are displayed in the *Details Pane*. Click on the **Summary** tab if it is not already selected. To edit the command summary click on the **Summary** tab and edit the appropriate fields. See Fig. 17 to see the summary fields of the set that can be edited.

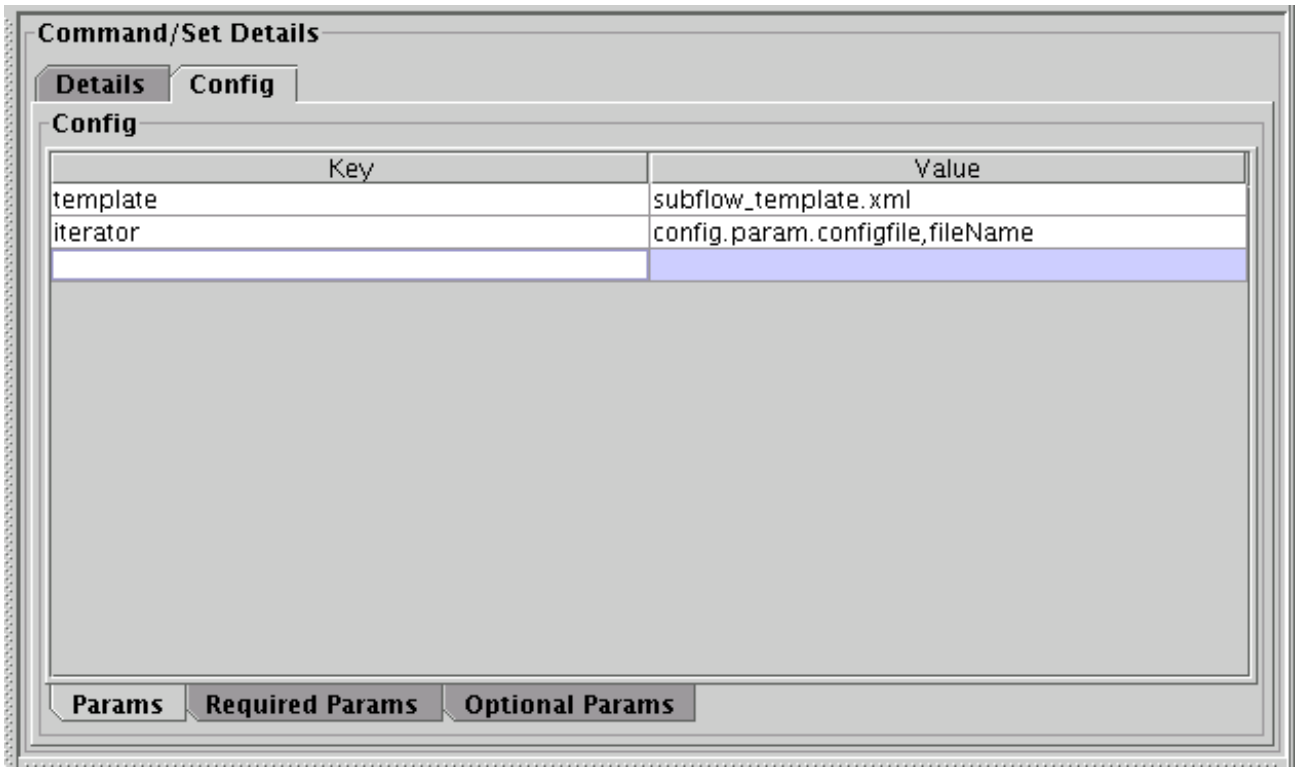
Fig 17: Template Editor - Command Set Summary Editing



7.1.4.7 Edit Set Config

To edit the set configuration select the set in the *Display Pane*. This will select the command and the details of the command are displayed in the *Details Pane*. To edit the set config click on the **Config** tab which will show the three types of config parameters that can be edited for commands. See Fig. 18 to see the summary fields of the command that can be edited.

Fig 18: Template Editor - Command Set Config Editing

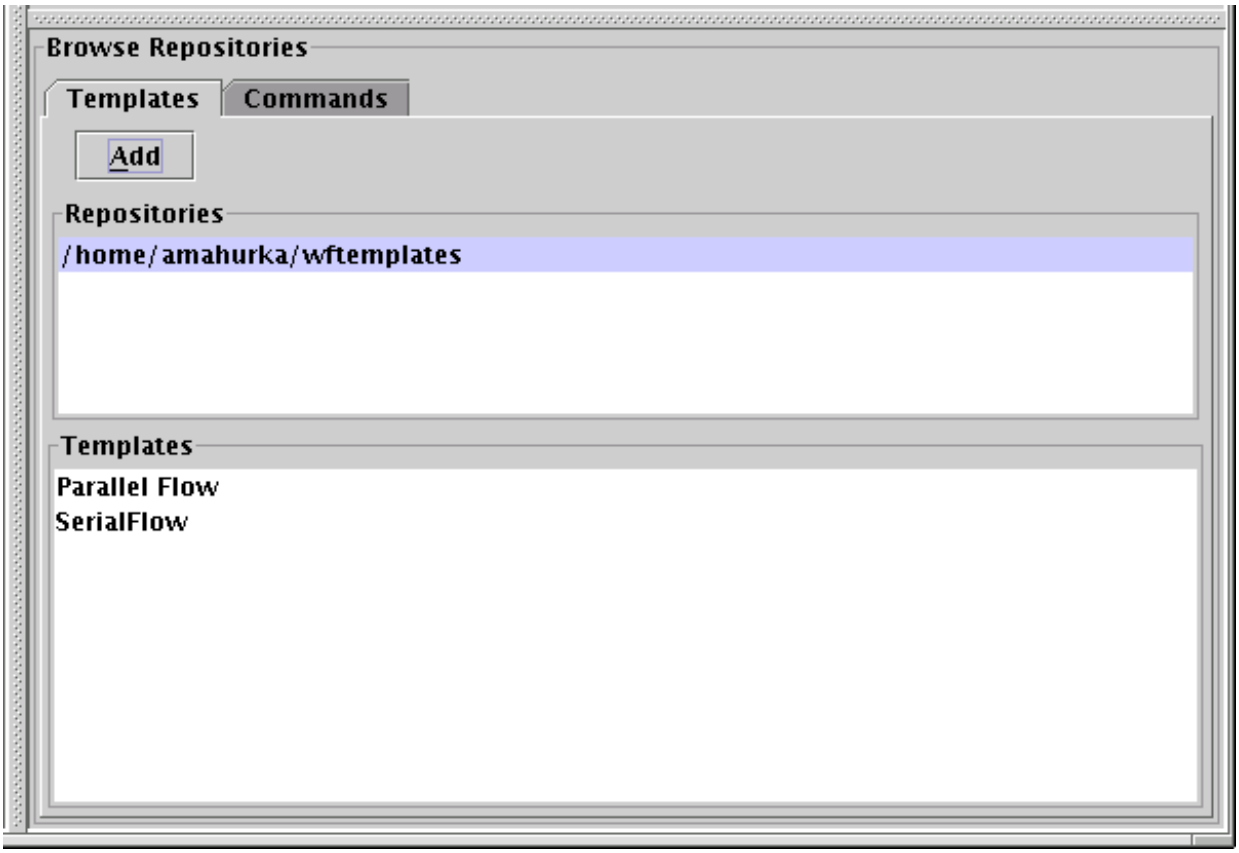


To edit the parameters of a command click on the **Params** tab and add the key value pairs to the list. The required params and optional params can be specified in the **Required Params** and **Optional Params** tabs.

7.1.4.8 Add Set from Template Repositories

Another way of adding command sets to a template is to choose a pre-defined template. To use a pre-defined template, select the **Templates** tab in the **Browse Repositories** pane of the editor and click on the **Add** button to navigate and select the directory that contains template files. The editor parses all the files in the specified directory to identify template files and adds the templates to the editor. The repository is listed in the **Repositories** list, and the templates in a selected repository are listed in the **Templates** list. Fig. 19 shows the commands in the *PANDA Library* in the repository */home/amahurka/wftemplates.*

Fig 19: Template Editor - Select Command Set Configuration Dialog



To add a template from the list, drag and drop one of the templates on to the Display Pane at the desired location. The *Select Set Configuration Dialog* shown in Fig. 15 filled with values from the selected template is displayed. The user can either accept the values or make necessary changes and click on the **OK** button to add the subflow to the current template.

7.1.5 Command Line Options

7.1.5.1 Summary

Param	Brief Description	Required	Default	Valid values
logconf	Java logging configuration file	Optional	*log4j.properties*	
t	Template file	Optional		
version	Program version	Optional		
help	Program usage or help	Optional		

7.1.5.2 Details

--logconf : This parameter is used to specify a user specified logger configuration file.

Instead of using the workflow default configuration file the user can specify a custom configuration file using this parameter.

-t : This flag is used to specify the template file to open. This is an *optional* parameter.

If no template is specified the editor is opened without any tempalte file.

-version : This flag is used to print the version information.

--help : This option produces a short help / usage message before exiting the program.

7.1.6 Usage and Examples

Usage:

```
EditTemplate [--help] [--version]
               [--logconf=<logger config file>]
               -t=<template file> | template file
```

Examples:

```
EditTemplate -t egc_template.xml
```

7.2 *CreateWorkflow*

CreateWorkflow is used to create a workflow instance from a **template** and **configuration** files. A workflow template is a bare-bones XML file that specifies the structure and definition of the workflow and included all the required and invariant elements of the workflow, including the *name* and the *configMapID*, that remain the same across instances of a workflow. Refer to the Workflow Template section of this guide for more information on templates. A configuration file is an *INI*-style file that specifies parameters for a single instance of the workflow. Refer to the Configuration File section of this guide for more information.

Workflow instances can be created in compressed format by specifying the name with '.gz' extension. Only '.gz' extensions are supported at this time. If a workflow instance has file based subflows, the file names for the subflows also have to be specified with '.gz' extension, on order to create the subflows in compressed format. Compression is decided on a per-file basis, therefore if a parent has '.gz' extension but the child does not, the parent is saved in a compressed format while the child is not.

7.2.1 Command-Line Options

7.2.1.1 Summary

Param	Brief Description	Required	Default	Data Type	Valid values
t	Workflow template file	Required		String	
c	Configuration file	Required		String	
i	Workflow instance file	Required		String	
autobuild	Build file-based subworkflows	Optional	*true*	Boolean	true, false
paramfile	Command line parameters file	Optional		String	
logconf	Java logging configuration file	Optional	*log4j.properties*	String	
delete-old	Delete old instance files	Optional		Flag	

7.2.1.2 Details

-t : This flag is used to specify a Workflow template. This is a *required* parameter.

For example, to use a Workflow template *egc_template.xml* which is under */usr/local/projects/workflow/testing/templates*, specify ***-t****=/usr/local/projects/workflow/testing/templates/egc_template.xml* .

-c : This flag is used to specify the configuration file. This is a *required* parameter.

For example, to use a configuration file *sample.ini* in the directory *usr/local/projects/workflow/testing/config*, specify ***-c****=/usr/local/projects/workflow/testing/config/sample.ini*.

-i : This flag is used to specify the Workflow instance file name. This is a *required* parameter.

To create an instance file in compressed (*.gz) format, specify the file name with a ".gz" extension. Only "*.gz" extensions are supported at this time.

For example:
To create an instance file *egc.xml* in the current directory, specify ***-i****=egc.xml.*
To create an instance file *egc.xml* in compressed format, in the current directory, specify ***-i****=egc.xml.gz*

--autobuild : This flag is used to specify if file-based subflow instance files are created now or their creation postponed till execution.

When this value is set to *true*, if a workflow has a file-based subflows, the instance builder attempts to build all subflow instance files recursively. The program fails if any of the instance files cannot be created.

If the value is *false*, then the recursive subflow instance creation is delayed till the point of execution, in effect allowing rudimentary dynamic workflow creation and execution. Thus, for instances where one of the commands in a parent flow elaborates the child flow, the user should set this flag to *false* to allow the creation of subflow instance files on the fly.

--delete-old : This flag is used to specify that all old instance files be deleted and recreated, the default behavior is to create files only if they do not exist.

--paramfile : This parameter is used to specify a file containing the command line parameters.

Instead of typing the command line parameters for each invocation, the command line parameters can be used from the specified INI style file. Explicitly specified parameters supersede file based parameters.

--logconf : This parameter is used to specify a user specified logger configuration file.

Instead of using the workflow default configuration file the user can specify a custom configuration file using this parameter.

-help : This option produces a short help / usage message before exiting the program.

7.2.2 Usage and Examples

Usage:

```
CreateWorkflow -t = <TemplateFile> -c = <ConfigFile> -i = <Workflow name>
 [--autobuild<true|false> (default true)] [--paramfile = <param file>]
 [--logconf = <logger config file>] [--delete-old]
```

Examples:

```
CreateWorkflow -t=/export/workflow/testing/templates/egc_template.xml
 -c=/export/workflow/testing/config/sample.ini -i=egc.xml

To Create an instance file in compressed format:
        CreateWorkflow -t=/export/workflow/testing/templates/egc_template.xml
        -c=/export/workflow/testing/config/sample.ini -i=egc.xml.gz
```

7.3 *RunWorkflow*

RunWorkflow launches the workflow engine that executes commands in an instance file. When this tool is invoked, it opens the XML instance file and launches all incomplete, interrupted and previously unsuccessful commands in the specified order through the specialized command processors. A Workflow instance can be run in compressed format, if it was created in compressed format with '.gz' extension. Only '.gz' extension is supported at this time. For ease of use, this tool may also be used to build an instance before it is executed. If the user specifies a template and configuration file, then this program creates a instance before executing it. Otherwise, the program executes commands in an existing instance file.

As of release 3.0, RunWorkflow allows only one engine to execute a given instance file at a time. If the engine encounters another engine executing this workflow the engine will fail with an error message.

7.3.1 Command-Line Options

7.3.1.1 Summary

Param	Brief Description	Required	Default	Data Type	Valid values
i	Workflow instance file name	Required		String	
t	Workflow template name	Optional		String	
c	Configuration file	Optional		String	
m	Marshalling interval in minutes	Optional		Integer	
notify	e-mail address for notification	Optional		String	
observers	Java classes to which event notifications are sent	Optional		String	
jars	Additional jar files prepended to classpath	Optional		String	
scripts	Standalone scripts to which notifications are sent	Optional		String	
delaybuild	Delay building file-based subflow instances	Optional	*true*	Boolean	true, false

delete-old	Delete old instance files	Optional		Flag	
resume	Resume execution of existing instance file	Optional		Flag	
paramfile	Command line parameters file	Optional		String	
init-heap	Initial heap size in megabytes	Optional	100m	String	
max-heap	Maximum heap size in megabytes	Optional	250m	String	
thread-stack	Thread stack size in kilobytes	Optional	JVM default (128K)		
proc-lookup	Custom command processor lookup	Optional			
dist-lookup	Custom distributed command processor lookup	Optional			
logconf	Java logging configuration file	Optional	*log4j.properties*		
perl-logconf	Perl logging configuration file	Optional	*log4perl.properties*		

7.3.1.2 Details

-i : This flag is used to specify the Workflow instance file name. This is a *required* parameter.

For example, to execute an instance file *egc.xml* in the current directory, specify ***-i****=egc.xml*.

To create and run an instance file in compressed (*.gz) format, specify the file name with a ".gz" extension.
Only "*.gz" extensions are supported at this time.

For example:
To create an instance file *egc.xml* in compressed format, in the current directory, specify ***-i****=egc.xml.gz*

-t : This flag is used to specify a Workflow template. This is an *optional* parameter, but is required if a config file is specified.

For example, to use a Workflow template *egc_template.xml* which is under */usr/local/projects/workflow/testing/templates*, specify ***-t****=/usr/local/projects/workflow/testing/templates/egc_template.xml* .

-c : This flag is used to specify the configuration file. This is an *optional* parameter, but is required if a template file is specified.

For example, to use a configuration file *sample.ini* in the directory */usr/local/projects/workflow/testing/config*, specify ***-c****=/usr/local/projects/workflow/testing/config/sample.ini*.

-m : This flag is used to specify the marshalling interval, an interval at which the workflow instance files are updated by the WorkflowEngine, while the workflow is running. This is an *optional* parameter. If this flag is specified the engine will accumulate the state changes till this interval has elapsed before saving changes. If the engine were to die then some of these accumulated state changes may be lost.

For example, to update the workflow instance files only every 2 minutes, specify -m=2

The marshalling interval, when specified, will also be passed along to the remote subflows, if any.

--notify : This parameter is used to specify an e-mail address to which workflow completion message is sent.

The message includes the Workflow instance file name and the status of the workflow.

--observers : This parameter is used to specify java observer classes to which workflow progress events are sent.

For example, to receive workflow start and completion events, specify a java class which implements the *CommandSetLifetimeLI* listener interface, along with the optional properties files as follows:
--observers*=org.tigr.MyObserver:myobs.props*

This will register the class *'org.tigr.MyObserver'* to receive command set start and finish notification. If the Workflow instance has child command sets, then each of these will trigger start and finish events. The second part of the specification is the properties file name. If one is specified the name is passed to the class so that the class can use this file to lookup the value it needs.

To register multiple observers either use multiple instances of '*observers*' parameter, or specify the observers as a comma separated list as follows:
--observers*=org.tigr.MyObserver:observer.props,org.tigr.CommandObserver*
See *Observers* section for additional details.

--jars : This parameter is used to specify additional jar files that are prepended to the execution classpath.

To specify the jar that contains the observer class specify the jar file as follows: --jars=*myproj.jar.* This will cause the *myproj.jar* file to be the first file on the classpath. Typically when the user specifies observers, this parameter is used to specify the jar file containing the observer classes or for specifying the file that contains custom command processor.

--scripts : This parameter is used to specify standalone programs/scripts to which event notifications are sent.

For example, to receive workflow start and completion events, register the program of interest, the event type, along with the optional properties files as follows:
--scripts*=myscript:life:myscript.props.*
This will register the script *'myscript'* to receive command lifetime notifications that include execution start and finish for each of the commands in the instance file and its sub-workflows. To register multiple observers either use multiple instances of '*scripts*' parameter, or specify the observers as a comma separated list. See *Observer Scripts* section for details on script invocation and the contract that these scripts must

adhere to.

--delaybuild : This flag is used to specify if file-based subflow creation is delayed till execution.

This flag takes on different meaning if the workflow execution is invoked with an existing instance file, or if the user wants the top-level instance file to be created before execution begins. If the user invokes this command with an existing top-level workflow and the value of this parameter is *true*, the Workflow engine attempts to create non-existent file-based subflow instances just before execution. If however the value is *false*, then the engine makes no attempt to create the subflow instances and returns with an error if it cannot find the subflow instance file.

In the second instance where the engine is asked to first create the Workflow instance before execution, if the value of **delayedbuild** is *true*, the engine delays the creation of file-based subflows till the point of execution. If the flag is *false*, the engine tries to create all the subflows when the parent is created and fails if any of the subflows cannot be created.

--resume : This parameter is used to specify that the engine should resume executing the workflow if it exists.

When the user specifies the '-t' and '-c' flags the typical behavior is for the engine to try and create the instance file. Specifying 'resume' flag indicates to the workflow that if the specified instance exists then do not recreate the instance file. This flag will override the behavior of the delete-old flag, such that even if **delete-old** is specified along with resume the top level instance is reused if it exists.

--delete-old : This parameter is used to specify that all old instance files are deleted if they exist and new files created.

The default behavior is for workflow to reuse any instance files, this flag will ensure that before an instance file is executed, it is always recreated (see 'resume' flag description for one exception case).

--paramfile : This parameter is used to specify a file containing the command line parameters.

Instead of typing the command line parameters for each invocation, the command line parameters can be used from an INI style file.

--init-heap : Initial heap size allocated for the JVM (default is 100 MB).

This is the amount of memory reserved by the JVM on startup for creating objects. As more objects are created, the JVM increases the amount of heap size up to the maximum heap specified. If the maximum heap size is exceeded, the JVM will throw an exception and quit.

--max-heap : Maximum heap size allocated for the JVM (default is 200 MB).

This is the maximum amount of memory allowed for creating objects. The JVM stats out by allocating the initial heap, as more objects are created, the JVM increases the amount of memory up to the maximum heap specified. If the maximum heap size is

exceeded, the JVM will throw an exception and quit.

--thread-stack : Individual thread stack size (default is JVM default).

--proc-lookup : This parameter is used to specify a processor lookup file that is checked to resolve command processor names.

The workflow system resolves the command processor used for executing a command by matching the name or type in the default lookup file *command_proc_factory_lookup.prop*. To use custom names or command types to map to existing command processors, or user developed command processors, the user can use this flag to specify the lookup file. This file, a java properties style file, contains entries where each key in the lookup file is either name or type while the value associated with the key is the name of the java class which executes the specific command.

--dist-lookup : This parameter is used to specify a distributed processor lookup file that is check to resolve distributed command processor names.

The workflow system resolves the distributed command processor used for executing a command by matching the name or type in the default lookup file *distributable_command_lookup.prop*. To use custom names or command types to map to existing command processors, or user developed command processors, the user can use this flag to specify the additional lookup file. This file, a java properties style file, contains entries where each key in the lookup file is either name or type while the value associated with the key is the name of the java class which executes the specific command.

--logconf : This parameter is used to specify a user chosen logger configuration file.

Instead of using the workflow default configuration file the user can specify a custom configuration file using this parameter.

--perl-logconf : This parameter is used to specify a user chosen logger configuration file for the perl components of the Workflow system.

Instead of using the workflow default configuration file (Log4perl.properties) the user can specify a custom configuration file using this parameter.

-help : This option produces a short help / usage message before exiting the program.

7.3.2 Usage and Examples

Usage:
Run existing work flow:

```
RunWorkflow -i=<WorkflowInstanceFile>
    [--delaybuild=<true|false> (default true)] [--resume] [--delete-old]
    [--observers=<class:props>] [--scripts=<script:event:props>]
    [--paramfile<param file>] [--notify=<user@tigr.org>]
    [--init-heap=<initial heap size>] [--max-heap=<max heap size>]
    [--thread-stack=<thread stack size>]
```

```
    [--proc-lookup=<processor lookup>] [--dist-lookup=<distributed processor lookup>]
    [--observers=<class:props>] [--scripts=<script:event:props>]
    [--logconf=<logger config file>] [--perl-logconf=<perl logger config file>]
```

Create and run a workflow:

```
RunWorkflow -i=<WorkflowInstanceFile> -t=<TemplateFile> -c=<ConfigFile>
    [--delaybuild=<true|false> (default true)] [--resume] [--delete-old]
    [--observers=<class:props>] [--scripts=<script:event:props>]
    [--paramfile<param file>] [--notify=<user@tigr.org>]
    [--init-heap=<initial heap size>] [--max-heap=<max heap size>]
    [--thread-stack=<thread stack size>]
    [--proc-lookup=<processor lookup>] [--dist-lookup=<distributed processor lookup>]
    [--observers=<class:props>] [--scripts=<script:event:props>]
    [--logconf=<logger config file>] [--perl-logconf=<perl logger config file>]
```

Examples:

Run an existing workflow:

```
RunWorkflow -i=/export/workflow/source/xml/egc.xml
 --observers=org.tigr.MyObserver:myobs.props --jars=myproj.jar
 --scripts=myscript.pl:life:myscript.props
```

Run an existing workflow in compressed format:

```
RunWorkflow -i=/export/workflow/source/xml/egc.xml.gz
 --observers=org.tigr.MyObserver:myobs.props --jars=myproj.jar
  --scripts=myscript.pl:life:myscript.props
```

Create and run a workflow:

```
RunWorkflow  -c=/export/workflow/testing/config/sample.ini
               -t=/export/workflow/testing/templates/egc_tempalte.xml
                                                              -i=egc.xml
```

Create and run a workflow in compressed format:

```
RunWorkflow  -c=/export/workflow/testing/config/sample.ini
               -t=/export/workflow/testing/templates/egc_tempalte.xml
                                                        -i=egc.xml.gz
```

7.4 *KillWorkflow*

KillWorkflow allows the user to terminate a workflow gracefully. When invoked with the name of an instance file, it finds the corresponding workflow and kills it. It is not necessary to execute KillWorkflow on the same machine on which the workflow is running or even to know what that machine is. Alternately, KillWorkflow may be invoked without a filename, in which case it lists all the workflows running on the current machine but doesn't kill any of them. If the desired workflow is found, the user may kill it by reinvoking KillWorkflow with the filename. The process group of the selected process usually includes all descendant processes. At the end of the signal propagation, the workflow gets the *'interrupted*' state.

Note: The behavior of KillWorkflow has changed significantly as of release 3.0. Among other changes, it will be

sure all descendant processes have actually been terminated before returning. The old version of the tool may be invoked as OldKillWorkflow.

7.4.1 Command-Line Options

7.4.1.1 Summary

Param	Brief Description	Required	Default	Data Types	Valid values
i	Workflow instance file name	Required		String	
list	List all active workflow	Optional		Flag	

7.4.1.2 Details

-i: This flag is used to specify instance file name

Specify the instance file name whose engine needs to be terminated

--list: This flag is used to list all running workflows

Use this flag to list all the workflows currently running on this machine.

7.4.2 Usage and Examples

Usage:
To terminate an active workflow:

```
KillWorkflow -i=<WorkflowInstanceFile>
```

To list all active workflows:

```
KillWorkflow --list
```

Examples:

To terminate workflow:

```
KillWorkflow -i=/export/workflow/source/xml/egc.xml
```

7.5 *CheckWorkflow*

CheckWorkflow allows a user to determine whether a given workflow is currently running and, if so, where. It should be invoked with the name of an instance file. The filename may be specified with either an absolute or a relative path; in the latter case, it will look in the current directory only. It will report the workflow's current execution host, if applicable. Alternately, if no workflow based on that instance file is running, it will report that fact. The functionality of CheckWorkflow is also used internally by the Workflow system whenever RunWorkflow is invoked to ensure that multiple copies of the same workflow aren't executed simultaneously.

7.5.1 Command-Line Options

7.5.1.1 Summary

Param	Brief Description	Required	Default	Data Type	Valid values
i	Workflow instance file name	Required		String	

7.5.1.2 Details

-i: This flag is used to specify instance file name

Specify the instance file name whose execution status is to be checked.

7.5.2 Usage and Examples

Examples:

Running workflow:

```
CheckWorkflow -i my-workflow.xml
The workflow is running on aegan-lx.
```

Completed workflow:

```
CheckWorkflow -i my-workflow.xml
No workflow based on my-workflow.xml is running.
```

7.6 *ControlWorkflow*

ControlWorkflow can be used to alter the course of a running workflow by restarting one or more commands or command sets on a workflow that is currently running. The commands or command sets to be restarted may have failed or may have succeeded but produced unexpected results (for example, because of an error in an input file). ControlWorkflow should be invoked with the name of an instance file, and a single command or set or a comma-separated list of commands or sets to restart. The commands or command sets should be the IDs from the instance file. Note that when multiple commands or command sets are to be restarted, the restarts are performed sequentially in the order in which the IDs are given. Currently running commands and sets or those that were not executed before cannot be restarted at this time. Also, the top-level command set may not be restarted; to restart a workflow from the beginning, just use KillWorkflow and then run the workflow again.

When a command or set to be restarted is part of a serial command set, all the commands that follow this command and have finished execution will have their state rest and will be executed again after the specified command. When the restarted command or set is part of a parallel command set that has some commands still running, the state of the parallel flows will not be reset, but only that specific command and its successors will be reset and restarted.

7.6.1 Command-Line Options

7.6.1.1 Summary

Param	Brief Description	Required	Default	Data Type	Valid values
i	Workflow instance file name	Required		String	
restart	List of command IDs to restart	Required		String	

7.6.1.2 Details

-i: This flag is used to specify instance file name

Specify the instance file name whose engine needs to be terminated

--restart: This parameter is used to specify the list of commands to restart

Comma separated list of command IDs which are to be restarted.

7.6.2 Usage and Examples

Examples:
Restart one command:

```
ControlWorkflow -i my-workflow.xml --restart 1922
```

Restart three commands:

```
ControlWorkflow -i my-workflow.xml --restart 1922,6705,6702
```

7.7 *CleanWorkflowRegistry*

CleanWorkflowRegistry can be used to clean up the RMI registry that was left in an inconsistent state because of some catastrophic failure of the workflow engine. This tool may also be used to list the RMI servers registered on a particular machine. To clean up the RMI registry you must be logged into the machine whose registry you are attempting to clean. You can list the registry from any host.

7.7.1 Command-Line Options

7.7.1.1 Summary

Param	Brief Description	Required	Default	Data Type	Valid values
i	Workflow instance file name which is to be cleaned up	Required		String	
host	Host name	Required		String	
list	List of command IDs to restart	Required		Flag	

7.7.1.2 Details

-i: This flag is used to specify instance file name that is to be cleaned up

Specify the instance file name whose RMI registry is to be cleaned up

--host: This parameter is used to specify the host name

Host name whose registry is to be listed, local host if nothing is specified.

--list: This flag is used to list the registry contents

7.7.2 Usage and Examples

Examples:
Clean up registry:

```
CleanWorkflowRegistry -i my-workflow.xml
```

List registry contents:

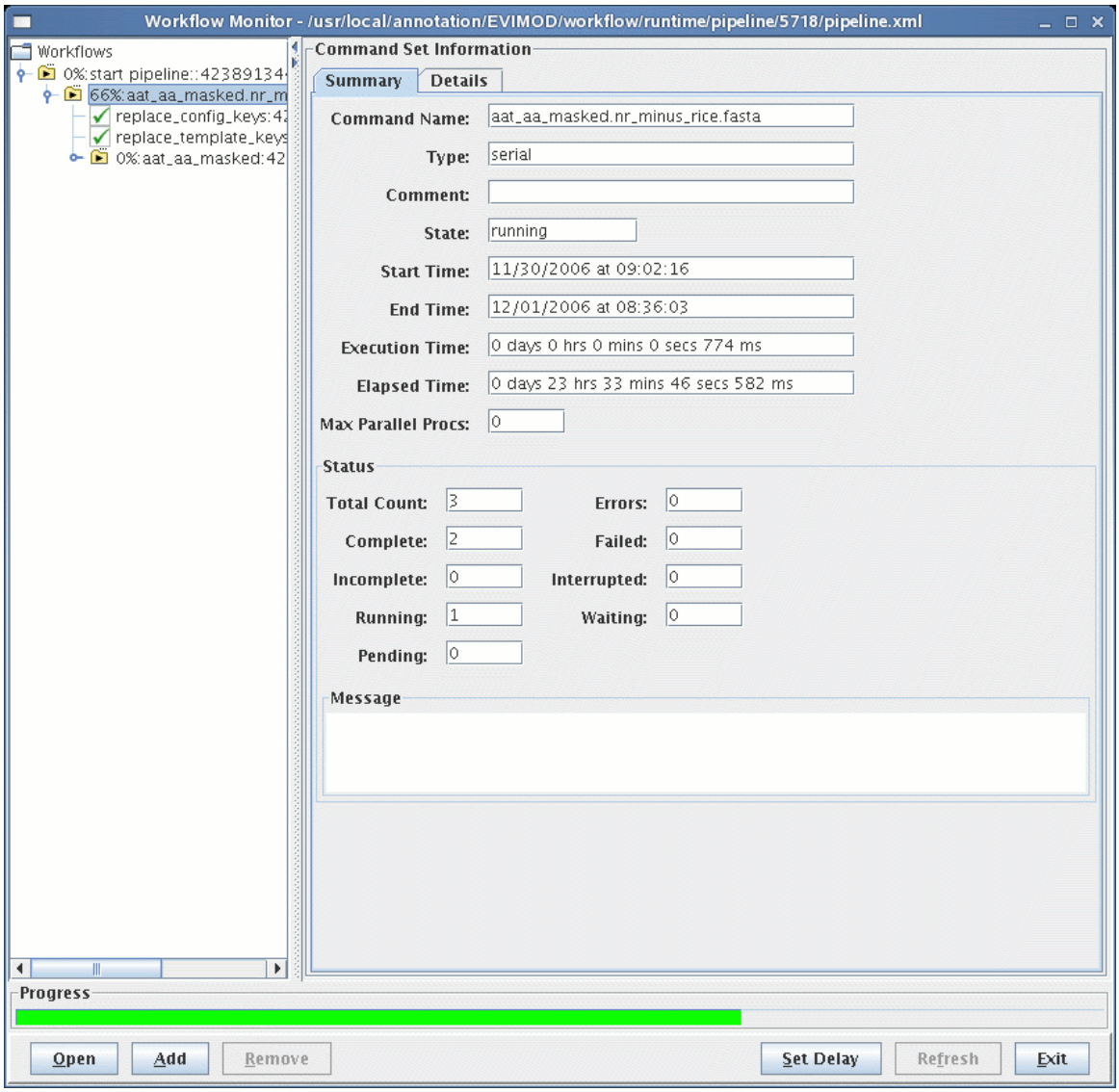
```
CleanWorkflowRegistry --host somehost --list
```

7.8 *Monitor Workflow*

This GUI tool allows the user to monitor the progress of one or more workflows by monitoring their instance files. The GUI checks all the instance files and their subflow files at the user specified frequency (default is 30 seconds) to see if any of the instance files have changed and updates the corresponding nodes automatically. The screen shot of the MonitorWorkflow screen shown in Fig 20 shows workflow being monitored.

The GUI has two main sections the left pane which shows the workflow instances as a tree, and the right pane which shows the details for the selected tree node. The details pane to the right shows the command set details corresponding to the workflow instance file *composite_out.xml*. The title of the screen shows the current top-level instance file being monitored. The tree nodes show the string built of the following elements for a command set, percentage complete, name, unique ID, and the instance file name for file-based sets. For commands the node string displayed includes the name and unique ID.

Fig 20: MonitorWorkflow - Command Set Summary



To see the details of an instance open the folder and click on the command or command set element. The right pane has a tabbed pane which includes various tabs that display the details of the selected node. In **Fig 20** the right pane shows the summary tab for the selected command set which includes the name, type, comment, state, start time, end time, elapsed time, the execute time, the status, and error messages for the command set. **Fig 21** shows the details tab for the command set and include the config path, and config parameters in addition to the above mentioned fields.

Fig 21: MonitorWorkflow - Command Set Details

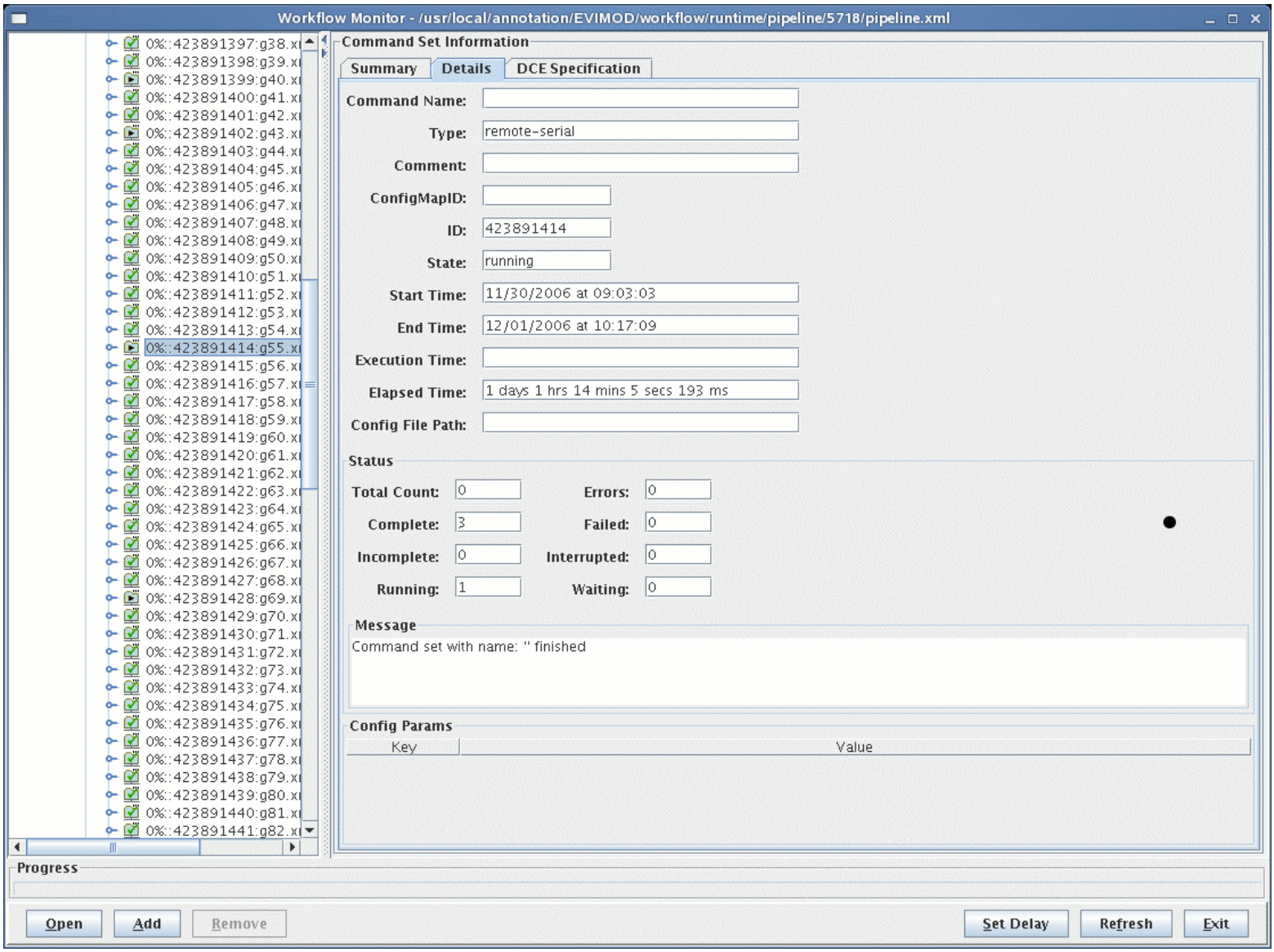


Fig 22 shows the summary tab for the selected command. The tab includes the name, type, comment, state, stdout, stderr redirects, start time, end time, execution time return value, message, and the command processor name or the simulated system command invocation string. The user can click on the **View Out** and **View Err** buttons to bring up the output or error redirected output if any. If the command is a distributed command the distributed computing environment (DCE) specifications are shown in a separate tab. See the next screen shot which shows the details of the command.

Fig 22: MonitorWorkflow - Command Summary

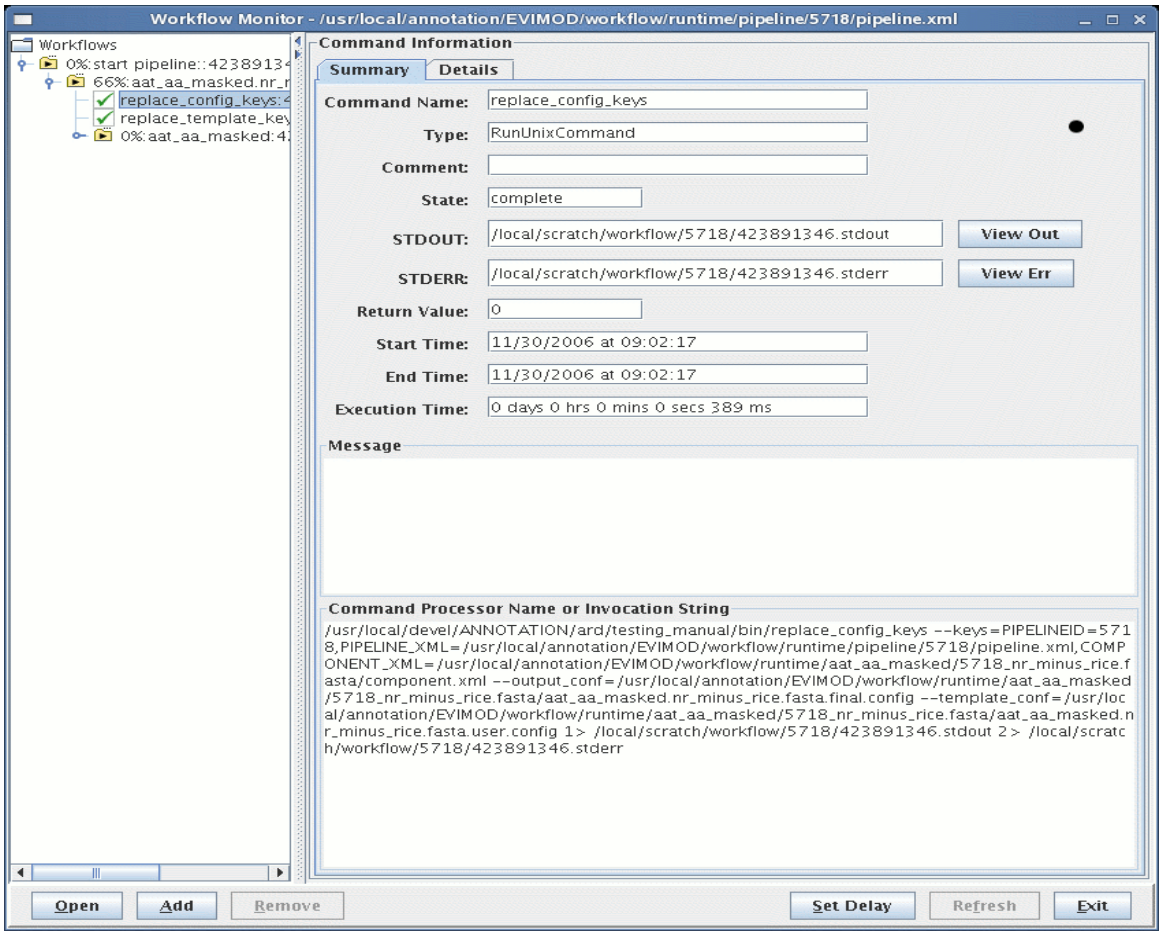


Fig 23 shows the details tab for the selected command and includes the retry count, retry attempts, the arguments, flags, and parameters for the command in addition to the fields displayed in the summary tab. If the command is a distributed command, a third tab, DCE Specification, displays the distributed computing environment specifications and execution information for the selected command.

Fig 23: MonitorWorkflow - Command Details

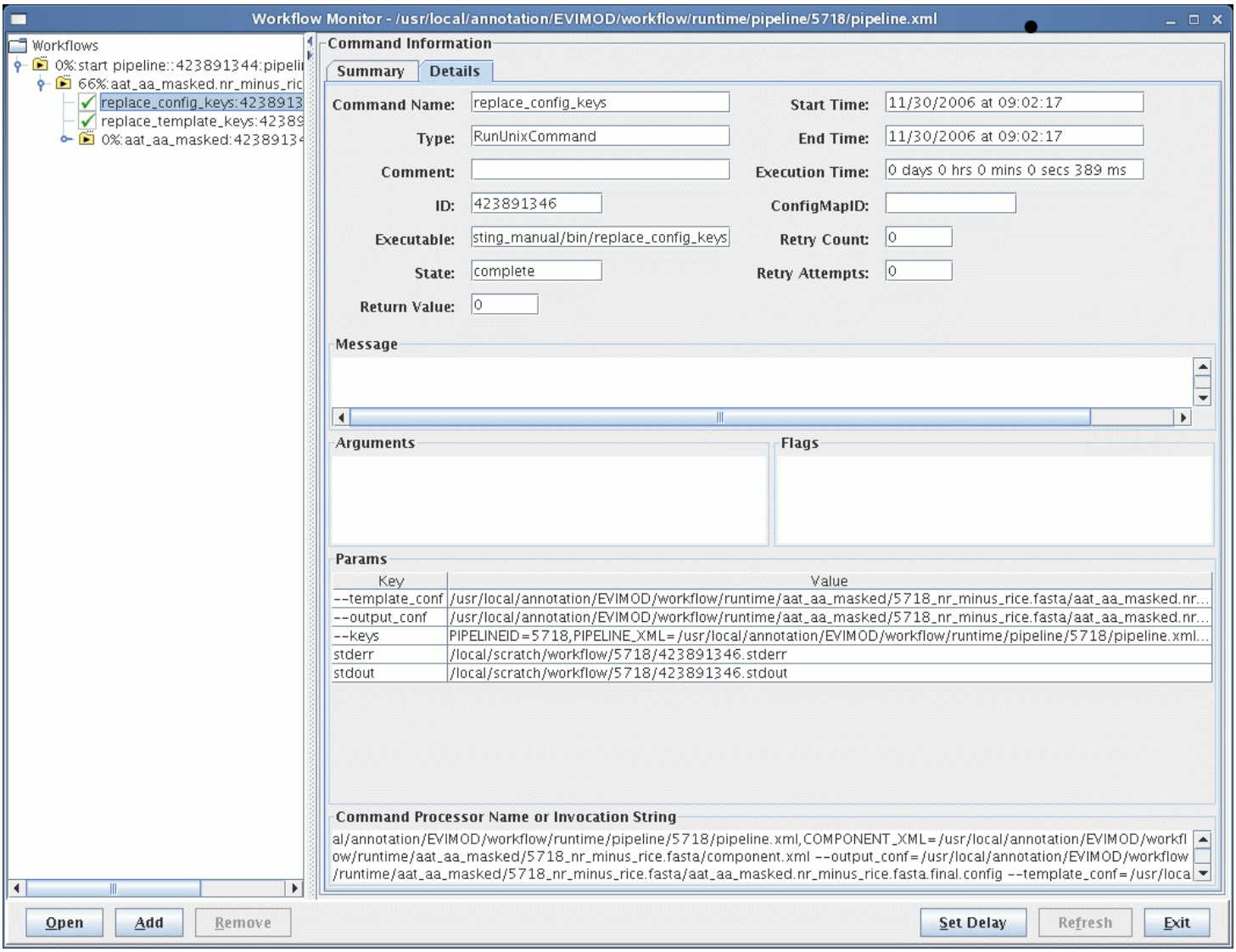
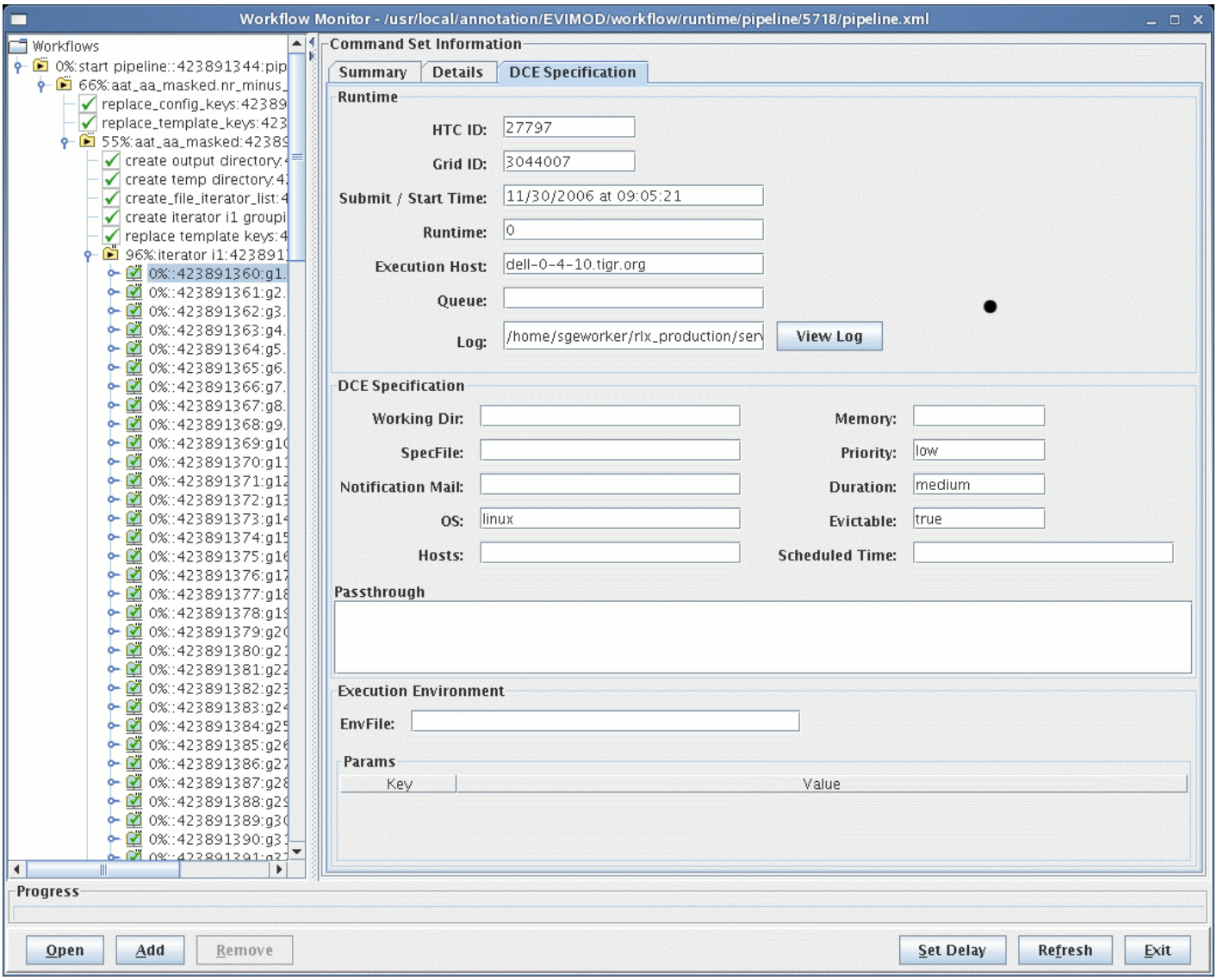


Fig 24 shows the DCE specification tab for the selected command and includes the HTC ID, Grid ID, the execution host, DCE log file, optional DCE specification file, the notification address, the OS, memory, queue, priority, and duration requirements, the submission and actual execution time, the passthrough string, the execution environment specification file and environment variables. The underlying DCE sets the values for the JobID, host name, and log file when the execution begins. It uses the requirements specified to find a matching computer that then executes the particular command.

Fig 24: MonitorWorkflow - Command DCE Specification Details



The GUI provides a mechanism to perform the operations such as adding or removing workflows, forcing a refresh, or setting the refresh delay. This section examines these operations.

7.8.1 Open Workflow

To monitor a different workflow click on the ***Open*** button. This will display a *Open File Dialog Box* which allows the user to choose the workflow instance file to open. When the 'open' function is used, all existing workflows are removed and the new instance file added to the monitor.

7.8.2 Add Workflow

To add a new workflow to the monitor click on the ***Add*** button. This will display a *Open File Dialog Box* which allows the user to choose the workflow instance file to add. When the add function is used, the new workflow is appended to the list of workflows in the monitor.

7.8.3 Remove Workflow

To remove a workflow instance select the folder representing the workflow instance. This will enable the ***Remove*** button. Click on the button to remove the workflow from the monitor. After a workflow is removed the monitor will select the next workflow if one is available, else the previous workflow is selected. If no workflows are present then the root is selected.

7.8.4 Set Delay

To change the frequency at which the monitor examines the instance files for change click on the **Set Delay** button. This will open a dialog box where the user can specify the the refresh delay in milliseconds.

7.8.5 Refresh

By default the monitor will recursively check all the instance files and their subflow files for file modification at the refresh frequency specified by the user. If any of the underlying files have been modified the corresponding node will be automatically updated. A user can force a refresh at any time by selecting a workflow instance, which will enable the ***Refresh*** button, and clicking on the button. This will force a refresh of the node representing the selected instance file.

7.8.6 Command Line Options

7.8.6.1 Summary

Param	Brief Description	Required	Default	Data Type	Valid values
i	Workflow instance file name	Optional		String	
proc-lookup	Custom command processor lookup	Optional		String	
dist-lookup	Custom distributed command processor lookup	Optional		String	
refresh	Refresh delay in seconds	Optional	*30 seconds*	Long	
logconf	Java logging configuration file	Optional	*log4j.properties*	String	

7.8.6.2 Details

-i : This flag is used to specify the Workflow instance file name.

--proc-lookup : This parameter is used to specify a processor lookup file that is checked to resolve command processor names.

By default the workflow system resolves the command processor used for executing a command by matching the name or type in the default lookup file *command_proc_factory_lookup.prop*. To use custom names or command types to map to existing command processors, or user developed command processors, the user can use this flag to specify the lookup file. This file, a java properties style file, contains entries where each key in the lookup file is either name or type while the value associated with the key is the name of the java class which executes the specific command.

--dist-lookup : This parameter is used to specify a distributed processor lookup file that is check to resolve distributed command processor names.

> By default the workflow system resolves the distributed command processor used for executing a command by matching the name or type in the default lookup file *distributable_command_lookup.prop*. To use custom names or command types to map to existing command processors, or user developed command processors, the user can use this flag to specify the lookup file. This file, a java properties style file, contains entries where each key in the lookup file is either name or type while the value associated with the key is the name of the java class which executes the specific command.

--refresh : This parameter is used to specify the refresh delay in seconds.

--logconf : This parameter is used to specify a user specified logger configuration file.

> Instead of using the workflow default configuration file the user can specify a custom configuration file using this parameter.

--help : This option produces a short help / usage message before exiting the program.

7.8.7 Usage and Examples

Usage:

```
MonitorWorkflow -i = instance file
    [--proc-lookup=<processor lookup>]
    [--dist-lookup=<distributed processor lookup>]
    [--logconf=<logger config file>]
```

Examples:

```
MonitorWorkflow --proc-lookup=egc_commands.props --logconf=custom_logger.conf egc.xml
```

8 Command Processors

The workflow system comes with a set of basic command processors that include the **SystemCommandProcessor**, a processor to execute other programs and scripts, **DistributedProcessor**, a processor to launch the executable on a remote machine via the DCE, **WaitProcessor**, a processor that waits for a predefined amount of time, and **WaitForFileCreationProcessor**, a processor that waits for the creation of a file. The details of these processors are discussed in this section.

8.1 SystemCommandProcessor

This specialized command processor written in Java is used to run system commands and can therefore be used to create a workflow made of other standalone utilities or applications. When the workflow engine encounters a command that uses this processor it invokes a system command to execute the specified program. The workflow engine looks for the program name by searching for the element with the name *executable*, in case it cannot find this element it looks for a parameter with the key *command* for the program name (Note: executable is a new element and is the preferred way of specifying the program name). The arguments, flags, and parameters to the system command are specified by a list of *param*, *arg*, and *flag* elements of the command.

So, for instance, to execute the following system command:

```
ls -al --width=100 --color=always /home/someuser /home/someotheruser
```

the major command elements of the workflow would be defined as shown below:

```
<command>
  <executable>ls</executable>
  <param>
    <key>--color</key>
    <value>always</value>
  </param>
  <param>
    <key>--width</key>
    <value>100</value>
  </param>
  <flag>-a</flag>
  <flag>-l</flag>
  <arg>/home/someuser</arg>
  <arg>/home/someotheruser</arg>
</command>
```

Note: In earlier versions of workflow if the parameter or flag did not have a leading '-' or '--', we attempted to guess the appropriate one based one on the key length. That method of specifying parameters is now deprecated, all parameters specified will be interpreted as is.

8.1.1 Redirection

To redirect stdout, stderr or stdin for a system command the user can either specify redirects as special parameters to the command or build the redirection string and specify it as one of the arguments. If the same file name is specified for stdout and stderr, the script builds an 'sh'-style string redirecting stderr to stdout. If no files are specified for redirecting *stderr* and *stdout*, the workflow system redirects the output to */dev/null*. The following redirection parameters of a command are interpreted as special redirection parameters by the engine:

- *stdout*
- *stdoutappend*
- *stderr*
- *stderrappend*
- *stdin*

To specify that stdin should be read from a file 'myin' and the stdout and stderr are to be redirected and appended to a file 'myout', the last command argument can be '1 >> myout 2>&1 <myin', or the command section might include the following parameters:

```
<command>
  <executable>somecommand</executable>
  <param>
    <key>stdin</key>
    <value>myin</value>
  </param>
  <param>
    <key>stdout</key>
    <value>myout</value>
  </param>
  <param>
    <key>stderr</key>
    <value>myout</value>
  </param>
  <param>
    <key>stdoutappend</key>
    <value>1</value>
  </param>
</command>
```

The complete command string used for the above example would be:

```
somecommand >> myout 2>&1 < myin
```

8.1.2 Command Elements

8.1.2.1 Summary

Element	Brief Description	Required	Default	Data Type	Valid values
executable	The system command to execute	Required		String	
param.command	An alternate mechanism for specifying system command	Required		String	
param.stdout	The file to redirect standard out	Optional		String	
param.stdoutappend	Indicate if the standard out should append or overwrite	Optional		Boolean	0, 1
param.stderr	The file to redirect standard error	Optional		String	
param.stderrappend	Indicate if the standard out should append or overwrite	Optional		Boolean	0, 1
param.stdin	The file from which to read standard in	Optional		String	

8.2 DistributedProcessor

This specialized command processor, written in Java, is used to run a command on a remote machine in a DCE. When the workflow engine encounters a command that uses this processor, the system uses the TIGR's High-Throughput Computing (HTC) API to submit the command to the underlying DCE. The same mechanism and command elements described for the **SystemCommandProcessor** are used by the distributed processor.

When a command is executed on a remote machine, the user can specify a series of requirements that the execution host must meet. The requirements available depend on the underlying DCE environment. The common requirements that can be specified include *memory*, *host name*, and *operating system*. The workflow engine will convey these to the underlying DCE. In addition to these, the user can specify a requirements string that is passed without interpretation to the underlying DCE through the *passthrough* parameter. See the DCESpecification schema for other requirements that can be specified.

8.3 WaitProcessor

This specialized command processor written in Java is used to wait for a predefined amount of time before completion. The processor by default waits for 5 seconds, this duration can be overridden by specifying a parameter with the key *duration* which specifies the duration to wait.

8.3.1 Command Elements

8.3.1.1 Summary

Element	Brief Description	Required	Default	Data Type	Valid values
param.duration	The durationthis processor should wait in seconds	Required	5	Long	
flag.create-file	Create a file to indicate that this command finished	Optional	Command name	Flag	
param.file	The name of the file to create after execution	Optional		String	
param.outfile	File to which a message is logged indicating that the processor started	Optional		String	

8.4 WaitForFileCreationProcessor

This specialized command processor written in Java is used to wait for a the creation of a file before proceeding. This is a quick way to add pauses to the workflow for the completion of some external event before a particular step can begin. For instance there may be a situation where we need to wait for an email notification about some request before the workflow can proceed. In such a situation the user can pause on a file and create this file when the notification is receieved.

The processor by default waits indefinitely for the creation of the file specified with the parameter with key *file* watching for the file creation every 10 seconds. The duration for which this processor waits as well as the watch frequency can be controlled by specifying additional parameters with the keys *watch-frequency*, and *max-duration* with values specified in seconds. If a maximum watch duration is specified and the file cannot be found the processor exits with an error code.

8.4.1 Command Elements

8.4.1.1 Summary

Element	Brief Description	Required	Default	Data Type	Valid values
param.file	The name of the file for which this processor is waiting	Required		String	
param.max-duration	The maximum duration this processor should wait in seconds	Optional	Indefinite	Long	
param.watch-frequency	The frequency in seconds at which the processor checks	Optional	10 seconds	Long	

9 Command Dispatchers

Workflow engine uses command dispatchers to execute the contents of a command set. The engine comes with three dispatchers, *LocalDispatcher, DistributedDispatcher, and RemoteDispatcher* to process local, distributed, and remote command sets. This section discusses these dispatchers and their behavior.

9.1 LocalDispatcher

This specialized command dispatcher is used to process commands in a command set that are executed on the local host where the engine is running. This dispatcher processes both serial as well as parallel command sets. In the present implementation, each command is executed in a separate thread. The total number of active threads executing command or sets for a given instance is limited by the maximum number of threads set in the configuration file. See the section on thread regulation for more details. In the case of a serial command set the maximum simultaneous processors is limited to one causing the dispatcher to behave like a serial dispatcher.

When this dispatcher is processing a parallel set and if one of the commands fails, the dispatcher will continue to launch other commands until all the commands in the set have been launched. While processing serial sets if one of the command fails, the execution is aborted.

9.2 DistributedDispatcher

This command dispatcher is used to process a command set made up of commands that should be executed on remote machines using the underlying DCE. This dispatcher processes both serial as well as parallel command sets. In the present implementation, a limit on the number of DCE requests that can be submitted by a single Workflow engine instance is specified in the workflow configuration file. See the section on thread regulation for more details.

When this dispatcher is processing a parallel set and if one of the commands fails, the dispatcher will continue to launch other commands until all the commands in the set have been launched. While processing serial sets if one of the command fails, the execution is aborted.

9.3 RemoteDispatcher

This command dispatcher is used to process a file-based subflow on a remote machine using the underlying DCE environment. This will cause the subflow to be executed in a separate engine running on the remote machine.

10 Processor and Dispatcher Lookup and Customization

The workflow system has been designed to be used out of the box using a standard set of processors SystemCommandProcessor, and *DistributedProcessor,* and dispatchers *LocalDispatcher, DistributedDispatcher, RemoteDispatcher*. However, the user is free to add other custom processors and dispatchers written in Java to perform pipeline tasks. This section discusses the mechanism by which the workflow system identifies these processors and how the users can use this mechanism to add custom processors and dispatchers.

10.1 Processor Lookup and Custom Processors

Custom processors can be built by extending the base command processor class *org.tigr.workflow.common.CommandProcessor*. The custom processor must then implement the necessary constructors and the *process* method which will be invoked by the engine to perform the actual task. For more information on adding a custom processor, see the programmer's guide.

The workflow system uses a factory class to instantiate a processor to process a specific command. This factory class uses the *name* or the *type* of the command to lookup the specific class used to execute a command in Java-properties-style lookup files. This mechanism also allows users to specify their own lookups that map meaningful names to specific command processors.

The system first attempts to map the command *name* from the lookup files, failing which, it attempts to use the command *type* to map. The user can specify a custom lookup file for workflow tools such as the engine and monitor with the *--proc-lookup* command-line argument. The system first attempts to lookup names in a user specified lookup file, if no file is specified, or no match can be found, it uses the default file *command_proc_factory_lookup.prop* found in the *properties* directory of the application deployment area.

```
The following are a few example entries from the file:

       WaitForFile = org.tigr.workflow.processors.WaitForFileCreationProcessor
       Sleep = org.tigr.workflow.processors.WaitProcessor
       Wait = org.tigr.workflow.processors.WaitProcessor
       RunPerlUnixCommand = org.tigr.workflow.common.SystemCommandProcessor
       RunUnixCommand = org.tigr.workflow.common.JavaSystemCommandProcessor
```

For instance, in the above file the command *type* **RunUnixCommand** maps to the class org.tigr.workflow.common.SystemCommandProcessor, which is used to process a command in a distributed environment. Similarly, the command processor org.tigr.workflow.processors.WaitForFileCreationProcessor is a generalized processor class that can be used to pause workflow execution till a file is created.

When commands are defined in *distributed* command set the workflow engine lookup is performed in a separate properties file called *distributable_command_lookup.prop,* which resides under the *properties* directory of the application deployment area.

The following are example entries from the file:

```
       RunDistributedCommand = org.tigr.workflow.common.DistributedProcessor
       DummyDistributedCommand=org.tigr.workflow.common.DistributedProcessor
```

10.2 Dispatcher Lookup and Custom Dispatchers

To accommodate custom dispatchers, the workflow engine uses a Java-properties-style file to keep a list of command set types and the corresponding Java classes used to execute them. This file dispatcher_factory_lookup.prop can be found in the *properties* directory of the application deployment area. Custom dispatchers must extend the class *org.tigr.workflow.common.CommandDispatcher*. For more information on adding a custom dispatcher, see the programmer's guide.

The following are example entries from the file:

```
serial = org.tigr.workflow.common.LocalDispatcher
parallel = org.tigr.workflow.common.LocalDispatcher
distributed-serial = org.tigr.workflow.common.DistributedDispatcher
distributed-parallel = org.tigr.workflow.common.DistributedDispatcher
remote-serial = org.tigr.workflow.common.RemoteDispatcher
remote-parallel = org.tigr.workflow.common.RemoteDispatcher
```

11 Observers

When a Workflow engine is created, the user can register one or more observers or listeners to receive notifications as various events occur in the workflow execution. Observers are Java classes that implement one or more of the Workflow event-listener interfaces. The Workflow system supports three kinds of event listeners, lifetime, status, and runtime, that provide the ability to monitor events at different levels of details. Lifetime observers are sent command or command-set start and finish notifications. Status observers are sent command or command-set start, finish, interruption and failure notifications. Runtime observers, which are only applicable to commands, are sent command submit, suspend and resume event notifications. When a user registers an observer, the engine identifies all the Workflow event-listener interfaces implemented by the observer and registers the observer to receive appropriate event notifications as workflow execution proceeds.

All observers must implement one of the two possible constructors shown below:

public void Observer();
public void Observer(String pPropsFile);

The zero-argument constructor is used if the observer does not need any additional information to process event notifications. The constructor which takes the properties file as an argument should be used when the observer needs information such as database-connection details or other variable values to perform its functions.

The following is a list of workflow related event-listener interfaces and their method signatures. See the javadoc for more detailed descriptions.

11.1 Command Set Interfaces

11.1.1 CommandSetLifetimeLI

This interface defines the command-set lifetime event notifications and includes set-execution start and finish events.

```
public void commandSetStarted(StartEvent p_event);
public void commandSetFinished(FinishEvent p_event);
```

11.1.2 CommandSetStatusLI

This interface defines the command-set status event notifications and includes set-execution start, resume, finish, interrupt and failure events.

```
public void commandSetStarted(StartEvent p_event);
public void commandSetFinished(FinishEvent p_event);
public void commandSetResumed(ResumeEvent p_event);
public void commandSetInterrupted(InterruptEvent p_event);
public void commandSetFailed(FailureEvent p_event);
```

11.2 Command Interfaces

11.2.1 CommandLifetimeLI

This interface defines the command lifetime event notifications and includes command-execution start and finish events.

```
public void commandStarted(StartEvent p_event);
public void commandFinished(FinishEvent p_event);
```

11.2.2 CommandStatusLI

This interface defines the command status event notifications and includes command-execution start, restart, finish, interrupt and failure events.

```
public void commandStarted(StartEvent p_event);
public void commandFinished(FinishEvent p_event);
public void commandRestarted(StartEvent p_event);
public void commandInterrupted(InterruptEvent p_event);
public void commandFailed(FailureEvent p_event);
```

11.2.3 CommandRuntimeLI

This interface defines the command set status event notifications and includes command submit, resume and suspend events.

```
public void commandSubmitted(SubmitEvent p_event);
public void commandResumed(ResumeEvent p_event);
public void commandSuspended(SuspendEvent p_event);
```

12 Observer Scripts

When a Workflow engine is created, the user can also register one or more observer scripts or programs to receive notifications as various events occur during the execution of the workflow. Observer scripts, unlike the observer Java classes in the previous section, are stand-alone programs or scripts that are invoked as execution proceeds. When an observer script is registered, the user must specify the event group type and may optionally specify a properties filename, which is passed to the observer script.

The following event group types are supported and map to the four listener interfaces discussed in the Observers section:

Event Type	Listener Interface	Events	Command or Command Set
setlife	CommandSetLifetimeLI	start, finish	Command Set
setstatus	CommandSetStatusLI	start, resume, finish, interrupt, failure	Command Set
life	CommandLifetimeLI	start, finish	Command
status	CommandStatusLI	start, restart, finish, interrupt, failure	Command

When an event notification is sent to the observer script, the following command-line parameters are passed to the script. The observer script must understand these options and process them as needed. In addition, the observer script must return with a zero value in the case of successful execution or a non-zero value to indicate an error in the execution of the script.

```
ObserverScript --name=<name> --id=<unique ID> --time=<time of event>
    --event=<start|finish|resume|failure|interrupt|restart>
    --file=<file name of enclosing instance file>
    [--props=<props file>] - Optional properties file
    [--message=<message>] - Optional message for failure, and interrupt events
    [--retval=<return value>] - Optional return value for finish events
```

As the Workflow engine is actually invoking these scripts, we strongly recommend that you keep their execution times short, particularly in instances where workflows are executed under a distributed computing environment.

13 Thread Regulation

Each Command or CommandSet of a workflow in a workflow engine is executed either by a CommandProcessor or a CommandDispatcher in a separate thread. When executing large workflows this will lead to the creation of a large number of threads posing a risk of slowing down or crashing the workflow engine or the host system. To avoid this risk, the workflow engine employs a mechanism to enforce a maximum limit on the number of processors or dispatchers that can run concurrently within an engine instance. Further, it also restricts the number of concurrent DCE jobs that can be submitted to the grid by the engine instance. These are configurable parameters in the *workflow.config*file found in the application deployment area and can be modified by the workflow administrator based upon your environment.

Following are the various limits that are enforced in the current release:

- **Command Processor Limit:** This limit represents the maximum number of concurrent CommandProcessors that can run inside a Workflow engine. The current limit is 400. This means that at any given instant, only a maximum of 400 commands can be executed with in the scope of a Workflow engine.

- **Command Dispatcher Limit: This limit represents the maximum number of concurrent CommandDispatchers that can run within the scope of a command set. The current limit is 10. This means that if a command set contains more than five command sets, only a maximum of five can be executed at once. There is no upper limit on the number of CommandDispatchers that can run simultaneously within a Workflow engine.**
- **DCE Jobs Limit: This limit represents the the maximum number of concurrent DCE jobs that can be submitted to the grid. The current limit is 50. This means that at given instance there can only be a maximum of 700 jobs submitted to the grid by a Workflow engine.**

14 Logging

The workflow system is written primarily in **Java**, but there are specialized processors that use **Perl** to run system commands. To allow logging through both these languages as well as to support custom processors written in other languages in the future, the system uses the log4xxx logging libraries. The workflow system has a built-in logging system that is used to log and monitor the application flow. This logging system is based on the Log4j logging system and uses the log levels listed in the table below to log messages about the application. This section describes the seven log levels and the types of messages logged for each of these levels. For more information about Log4j, see the short manual at the Log4j web site. The logging system is configured to monitor for changes in the configuration file so that the log levels can be changed on the fly during the workflow execution.

The Log4j logging system uses a set of log appenders that are configured through a configuration file that specifies the different appenders and their effective log levels. This file can be modified at runtime to change the logging behavior. When the user chooses a particular log level for an appender, messages intended for all levels above the selected level (that is, with lower numbers) are also logged. For instance, when the log level is set to DEBUG, then all messages of the levels DEBUG, INFO, WARN, ERROR, and FATAL will be logged. The table below summarizes the types of messages logged at each level. The Log Levels section discusses these levels in greater detail.

Note: Even though, for backward compatibility, the tools still accept -v (log level) and -l (log file) flags to enable logging, the preferred way to enable logging is through configuration files, these options will be depreacted in a future release.

Log Level	Param Value	Messages
Fatal	0	Exceptional conditions leading to failure
Error	1	Non-fatal conditions
Warn	2	Non-critical conditions
Info	3	Messages informing the user of the major steps
Debug	4	High-level trace of application flow
Finer	5	Detailed trace of application flow
Finest	6	Verbose mode of application execution

14.1 Default Java configuration file

By default the workflow system looks for a logging configuration file, *log4j.properties*, in the current working directory. If that cannot be found, it uses the default file included in the workflow system. The user can replace this default logging configuration by specifying a custom logging configuration file through the --logconf parameter to all the workflow tools or creating an file in the current working directory.

The following are the contents of the default log4j.properties file in the Workflow installation area:

```
instance=${wf.instance.file}
file=${instance}.log
log4j.rootLogger=INFO#org.tigr.antware.shared.util.Finest, FILE
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${file}
log4j.appender.FILE.Append=false
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=%p %d{HH:mm:ss:SSS} [Name: %X{Name}] %C{1} %M:%L %m%n
```

```
# Console configuration
log4j.appender.CON=org.apache.log4j.ConsoleAppender
log4j.appender.CON.Threshold=WARN
log4j.appender.CON.layout=org.apache.log4j.PatternLayout
log4j.appender.CON.layout.ConversionPattern=%p %d{HH:mm:ss:SSS} [Thread: %t] %C{1} %M:%L %m%n
```

The file creates a single file appender which creates a log file with the extension '<instance file>.xml[.<tool>].log'. When executing programs for which there is no explicitly specified instance file, such as the MonitorWorkflow or EditTemplate , the process ID is used as the file prefix of the log file. This runtime file name resolution is accomplished by setting the wf.instance.file java runtime property in the launch script. The tool extension is added for all other tools except CreateWorkflow and RunWorkflow. Therefore when checking a workflow with the name **pipeline.xml** the log file would be named **pipeline.xml.check.log.**

The above configuration file specifies that the default log level is **INFO** and the class ***org.tigr.antware.shared.util.Finest*** is used to define the log levels. New messages are appended to the file, if one exists, through the *Append* pragma. The pattern layout is used to lay out the log message with the specified conversion pattern, which prints the log level (%p), date and time (%d), the message diagnostic context 'Name' (%X{Name}), the class name (%C{1}), the method and line number (%M:%L), and finally the message (%m).

14.2 Log Levels

14.2.1 Fatal

Messages logged at this level are deemed to be important to the workflow administrator and the end user. Fatal errors usually require the application to be aborted. Messages logged at this level should provide adequate information to the end user or the administrator to determine the reason for the error and its potential location in the application flow. A fatal message could be a detailed one, possibly with a proper stack trace of the error or exception. Example causes of fatal errors: *missing files, invalid configuration or template files, invalid working directories, incorrect parameters, command failure in a serial workflow,* etc.

14.2.2 Error

Messages logged at this level are deemed to be important to the workflow administrator and the end user. These errors will not require the application to be aborted but will affect the application flow and the end result. A typical example of such an error is *command failure in a parallel workflow.*

14.2.3 Warn

Messages logged at this level may be of importance to the workflow administrator and the end user. These errors, while of concern, are not expected to affect the application flow significantly. A typical example of such a condition is *usage of deprecated format for config and template files.*

14.2.4 Info

Messages logged at this level may interest both the workflow programmer and the end user. All major events that provide the progress of the application flow are typically logged at this level. All the configuration information such as *workflow parameters, database profile, user profile, server / machine profile* and the *invocation strings of system and distributed Commands* must be logged at this level. A log file that is created at this level may provide a medium level of application monitoring.

14.2.5 Debug

Messages logged at this level may be of importance to both Workflow programmers and the workflow administrator. Application-flow monitoring at a detailed level is made possible by logging messages at this level. Typical debug messages include messages about *the parsed fields of configuration files, Command and CommandSet elements and attributes, event notifications, marshalling and unmarshalling of Command or CommandSet objects in xml forma*t *and SQL statements*. Logging at this level may produce a great deal of output, so users should be cautious. Users may also enable or disable logging for specific packages/modules of the application by providing custom configuration files for Log4J or Log4perl.

14.2.6 Finer

Messages logged at this level are of importance primarily to Workflow programmers. Application-flow monitoring at an even more detailed level is made possible by logging messages at this level. Typical **'finer'** level messages include messages about *the entry and exit of each routine along with the parameters or parameter summaries in case of lists, return values, and summaries in case of lists, etc., and other detailed information about the application progress.* Logging at this level will produce very verbose output, so users should be cautious. Users may also enable or disable logging for specific packages/modules of the application by providing custom configuration files for Log4J.

14.2.7 Finest

Messages logged at this level are of importance primarily to Workflow programmers. Application-flow monitoring at the most detailed level is made possible by logging messages at this level. Typical **'finest'** level messages include messages about *values of iteration parameters and other information intended to assist programmers in stepping through the code.* Logging at this level will produce very verbose output, so users should be cautious. Users may also enable or disable logging for specific packages/modules of the application by providing custom configuration files for Log4J.

15 Reporting Problems

Please report all problems, concerns and comments about the Workflow system by sending mail to antware@tigr.org.