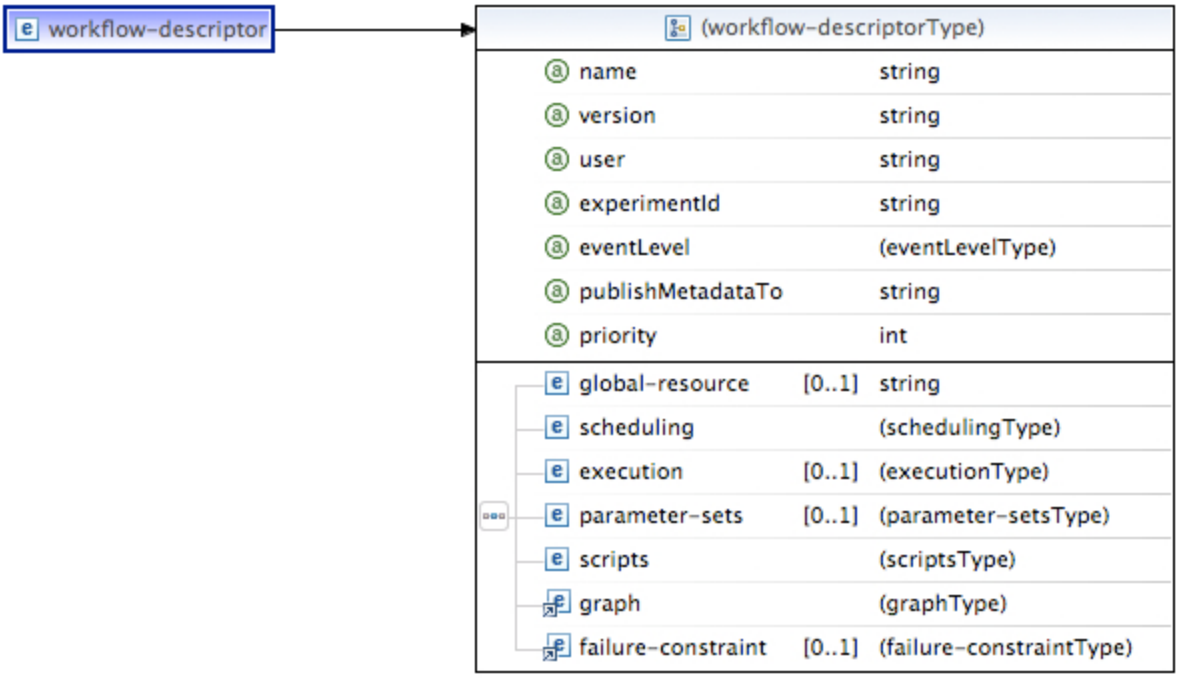
PTPFlow Workflow Descriptor

An annotated XSD for the workflow descriptor discussed below is available for download: workflow.xsd

Workflow Descriptor



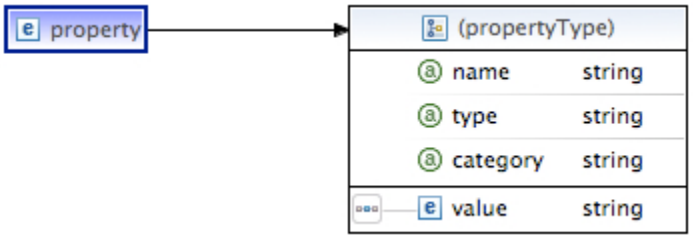
Represents a workflow description to be submitted to the Parameterized Workflow Engine, part of the PTPFlow service stack. The descriptor contains properties for scheduling the workflow, execute-time properties associated with the various executable nodes of the graph, parameterization descriptons, payload scripts, and the graph itself.

eventLevel	Tells the event-sending components involved in handling the workflow what level of information to publish as remote events (rather than or in addition to logging). The following list of options is in ascending order, such that successive levels include the preceding ones: ERROR, INFO, STATUS, PROGRESS, DEBUG.
publishMetadataTo	Indicates to a potential metadata agent that the events for this workflow should be processed for cataloguing. The value of the attribute should indicate the type of agent. If the attribute is missing, no metadata handling will take place.
priority	Expresses a global priority for this workflow. (Currently unused.)
global-resource	Automatic scheduling can be overridden by setting this element. A comma-separated list of DNs tells the scheduler to choose only from those resources (rather than all the resources it knows about, the behavior which occurs when no resource is indicated); a single DN tells the scheduling module to bypass the full scheduling routines. The selected resource is applied to all nodes in the expanded graph, unless the individual node carries a different resource indication (see also below).
failure-constraint	See further below. These are global constraints applying to the entire workflow graph; each node can also have its own set of constraints. Default behavior is that a single node failure makes the entire workflow fail.

Scheduling & Execution Profiles

The **<scheduling>** and **<execution>** elements provide to the various nodes in the workflow static runtime properties they require either to be scheduled or to execute. Each list consists of a set of profiles, which in turn consist of a set of properties.

In general, execution profiles represent optional elements which group together run-time properties which may be required by one or more nodes in the execution graph. Scheduling properties are needed earlier in the workflow lifecycle within the engine, in order to determine where to run the nodes of the workflow graph. See below for guidelines as to whether a property should appear in one or the other categories.



The property defined here is used widely in our Java-based code; type denotes any of the serializable types which the codebase knows about, but in the current context, type is usually limited to strings or primitives. The value element can also be expressed as an attribute of the property element. Category is a catch-all attribute which allows us to tag the property for special purposes.

Typical scheduling properties

For all workflows, a submission type must be determined; default is interactive.

```
        <property name="submissionType" type="string" /><!-- batch / interactive -->
```

For batch submissions, the following are usually required, though some systems and/or queues have default settings:

```
        <property name="account" type="string" />
        <property name="maxWallTime" type="long" /> or
        <property name="minWallTime" type="long" />
```

The following are the most common ways to request processors or cores:

```
        <property name="ranksPerMember" type="int" />
        <property name="threadsPerRank" type="int" />
        <property name="maxCpus" type="int" />
        <property name="minCpus" type="int" />
```

Explanation:

If cpus are not set, they will (both) default to the number of cpus needed to satisfy all members (=1 for non-parametric workflow nodes); ranks is the MPI parallelism, on a member-by-member basis (default = 1); threads is number of threads/cpus assignable to each rank (default = 1).

For non-mpi workflow nodes to which you wish to assign k > 1 cpu (per member), either ranks or threads can be set to k (though it perhaps makes more sense to view this as a case of rank = 1, threads = k); in any case, on non-LoadLeveler systems, only the total cpus per member has any meaning.

The following are specialized for a "master-worker" node where an (ELF) script acts as the master and launches k workers through a scheduler (as with the IBM LL version of what we call "glide-in"); these specify the submission properties (as opposed to the total request given to the scheduler for reservation purposes), as we do not want to submit the master requesting all the reserved resources (else workers will not be able to run in the reservation); note this is different from a master submitting to a "glide-in" batch system, where the main script does request the full resources which it is responsible for distributing to the workers.

```
        <property name="masterRanks" type="int" />
           <property name="masterThreadsPerRank" type="int" />
```

If you wish to override the automatic computation of how many members there are in a glide-in partition (i.e., in order to grab extra work from the TupleSpace):

```
        <property name="maxMembers" type="int" />
```

If you wish to override the computation of how many nodes are necessary to run the job based on the number of cores per node, use the following:

```
        <property name="coreUsageRule" type="string" />
```

EXAMPLE: ABE:7,BLUEPRINT:16

where the name corresponds to HostInfo key for the machine, and the number is cores per node to be used.

The following defaults to true; it indicates that unused cores on a node allocated to this submission should be used if possible; i.e., if you are running 4 16-wide members on nodes with 32 cores, you'd ideally like to run on 2 nodes, not 4; set to false if you want to enforce only 16 per node (for LL):

```
        <property name="shareNodes" type="boolean"/>
```

To indicate to the ELF container that members should not have to run as long as the container needs to:

```
        <property name="maxWallTimePerMember" type="long" />
           <property name="minWallTimePerMember" type="long" />
```

These default to 'std[].log' in the initial directory:

```
        <property name="stdout" type="string" />
        <property name="stderr" type="string" />
```

Other optional properties:

```
        <property name="queue" type="string" />
        <property name="stdin" type="string" />
        <property name="maxTotalMemory" type="long" />
         <property name="minTotalMemory" type="long" />
        <property name="dplace-trace" type="boolean" />
        <property name="nodeAttributes" type="string" />
```

(The last one, if supported, allows you to specify a particular kind of machine node to run on.)

Optional properties that are currently unimplemented/unsupported:

```
        <property name="maxCpuTime" type="long" />
           <property name="minCpuTime" type="long" />
           <property name="maxCpuTimePerMember" type="long" />
           <property name="minCpuTimePerMember" type="long" />
        <property name="maxMemoryPerNode" type="long" />
           <property name="minMemoryPerNode" type="long" />
           <property name="maxSwap" type="long" />
           <property name="minSwap" type="long" />
           <property name="maxDisk" type="long" />
           <property name="minDisk" type="long" />
           <property name="maxBandwidth" type="long" />
           <property name="minBandwidth" type="long" />
        <property name="maxOpenFiles" type="long" />
        <property name="maxStackSize" type="long" />
        <property name="maxDataSegmentSize" type="long" />
        <property name="maxCoreFileSize" type="long" />
        <property name="checkpointable" type="boolean" />
        <property name="suspendable" type="boolean" />
        <property name="restartable" type="boolean" />
        <property name="priority" type="int" />
```

Properties defined in the Host Information configurations:

If a Host Information configuration defines a property for a given host (as an "environment property"), and the name of that property is also included in a scheduling profile for a workflow, then that host will only be included among the potentially matching targets if the values for that property are the same. For instance, if

`SUPPORTS_GLIDE_IN="false"`

for hostA, and a scheduling profile contains

```
<property name="SUPPORTS_GLIDE_IN" type="boolean"><value>true</value></property>
```

then that property will be enforced on the match such that hostA will be excluded as a possible match for the node with that scheduling property.

Platform-dependent properties

In order to allow a single workflow to run on multiple independent resources, it may be necessary to define certain properties (such as paths) according to the targeted resource. This can be achieved by doing the following:

1. Define a profile containing a platform-specific configuration as a property included within an <execution> profile (note that the category must be "platform.configuration"):

```
<execution>
    <profile name="paths">
      <property name="paths-${HOST_KEY}" category="platform.configuration"/>
    </profile>
</execution>
```

The following variables are replaced on the basis of the target resource information contained in the Host Information configurations:

```
HOST_KEY:                e.g., "ABE"
NODE_NAME:                e.g., "grid-abe.ncsa.teragrid.org"
ARCHITECTURE:                e.g., "x86_64"
OS_NAME:               e.g., "Linux"
OS_VERSION:                e.g., "2.6.9-42.0.10.EL_lustre-1.4.10.1smp"
```

2. Configurations for this profile, based on the various possible resolutions of the variable, are then written to the TupleSpace service. For example,

```
<tspace-entry-builder id="1" owner="/C=US/O=National Center for Supercomputing Applications/OU=People/CN=Albert
L. Rossi" typeName0="platform.configuration" typeValue0="paths-ABE" name="tspace-entry-paths-ABE">
   <ranOn/>
   <payload payloadType="ncsa.tools.common.types.Configuration">
      <configuration>
        <property name="PATH_TO_EXECUTABLE" category="environment">
          <value>/u/ncsa/arossi/exec</value>
        </property>
      </configuration>
   </payload>
</tspace-entry-builder>
```

Presumably there would be one tuple of this sort for each resource the workflow is able to run on.

Some questions ...

1. Can platform-dependent properties appear in <scheduling> profiles?
    - ANSWER: No, only in <execution> profiles. If we think of <scheduling> properties as being used to determine what target resource to use, then obviously platform-dependent properties should not be included.
2. Then why, for instance, is "account" one of the <scheduling> properties?
    - ANSWER: A contradiction. It is included simply because it is specific to running on a resource. Note that if it were placed in an <execution> profile, the workflow would still complete successfully.
3. Which of the <scheduling> properties must appear in a <scheduling> profile?
    - ANSWER: Currently, the properties needed to make a scheduling request/reservation. These include values which affect the number of cores/nodes required and the wallclock time, along with submission type, and any properties that must match Host Information environment properties. All the others could actually appear either in <execution> profiles or platform.configuration tuples if so desired.

Scheduling options

In addition to profiles, an <options> element can be included in the <scheduling> section. This element has one attribute and two sub-elements, and can appear with any combination of these defined.

1. ```
   <options algorithm="random"/>
   ```
   ```

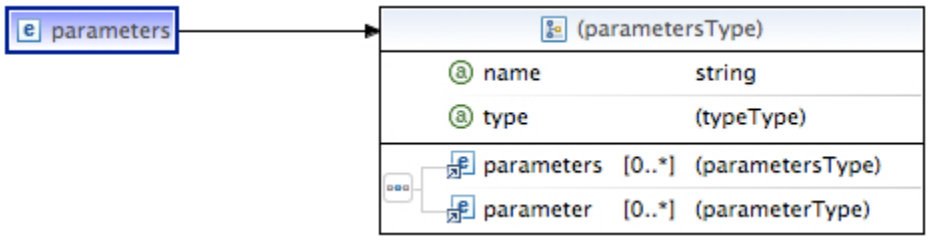
2. `<options><must-terminate-by>2009/09/03 11:30:00</must-terminate-by></options>`

3. `<options><rules>starttime=0:00:30;cpus=0.5,walltime=2.0;cpus=0.25,walltime=4.0</rules></options>`

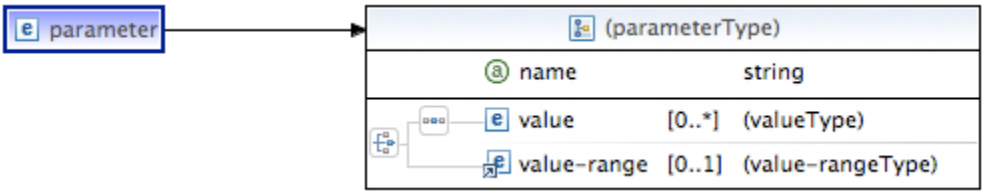
1. The algorithm attribute defines the method used to order potential matching target resources, defining the sequence in which they will be tried. There are currently two available algorithms, one which randomly orders the target names, and the other ("static-load", the default), which contacts the machine to determine something like a "load" number on the system.
2. Including this element indicates the workflow should be treated as "on-demand" (hard, time-based reservations determined all up front for the entire graph).
3. This element establishes a set of rules to apply, in order, to the resource request issued to each potentially matching target machine when the original request fails. Currently, there are three available modifiers: `starttime`, `cpus`, `walltime`; rules are separated by a semicolon, and clauses of the rule by commas; the predicate stands for a percentage alteration of the original value or, in the case of starttime, an increment. Thus the rules tell the scheduler to try 4 times; first with the original request; then by pushing forward the start-time; then by halving the number of cpus and doubling wall time; then finally, by taking one-fourth of the cpus and increasing wall time by 4.

Parameter Sets

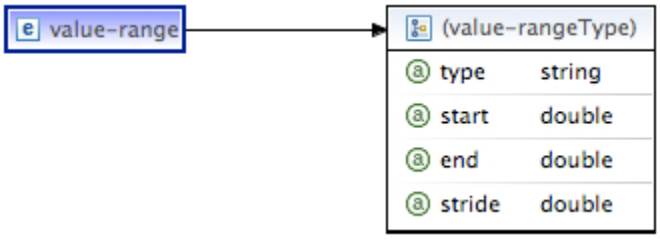
Optional named descriptions of how to parameterize nodes in the graph.



parameters	For this recursive object to be valid, it must bottom out in at least one simple parameter element; for an example, see below.
type	1. *product* signifies the Cartesian product obtained by crossing each of its members with the others. 2. *covariant* signifies the selection of one element from each of its members, in the order given. The "bin" represented by each member must have matching cardinality, or the parameterization will fail with an exception.



Simple parameter sets can be of two types: a list of typed values (the <value> element can have a *type* attribute), or a single value-range description.



The text of this element can be a comma-delimited list of values; alternately, the *start*, *end* and *stride* attributes can be used with numerical types.

Parameterization example

```
<parameter-sets>
     <parameters name="compute" type="covariant">
         <parameters type="product">
            <parameters type="covariant">
               <parameter name="conditioning-algorithm">
                  <value>file:/conditioning-0</value>
                  <value>file:/conditioning-1</value>
               </parameter>
               <parameter name="physics">
                  <value>file:/physicsP</value>
                  <value>file:/physicsQ</value>
               </parameter>
            </parameters>
            <parameter name="t">
               <value-range type="double" start="-1.0" end="1.0"
                  stride="0.5" />
            </parameter>
         </parameters>
         <parameters type="product">
            <parameter name="input">
               <value>file:/input-x4083</value>
               <value>file:/input-x63</value>
               <value>file:/input-z762</value>
               <value>file:/input-x111</value>
               <value>file:/input-b059</value>
               <value>file:/input-z4985</value>
               <value>file:/input-a3118</value>
               <value>file:/input-c5593</value>
               <value>file:/input-x2067</value>
               <value>file:/input-z4391</value>
            </parameter>
            <parameter name="logfile">
               <value>file:/log</value>
            </parameter>
         </parameters>
         <parameter name="case">
            <value-range type="int" start="0.0" end="9.0" />
         </parameter>
     </parameters>
</parameter-sets>
```

The top level parameter set, named 'compute', ends up with 10 members, formed by picking one element from each of three 'bins' in order:

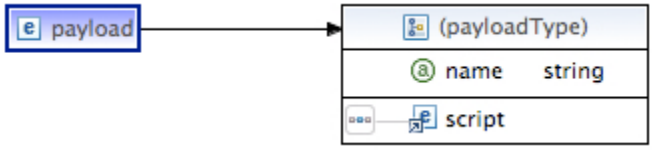
- bin 1: the Cartesian product formed by two parameter groups; (conditioning, physics) and t the first group is another covariant set of 2 values apiece the second group is a simple range which produces 5 values:
    - ((conditioning-0,physicsP),(conditioning-1, physicsQ)) X (-1.0, -0.5, 0, 0.5, 1.0);
- bin 2: the Cartesian product formed by two parameters: input and logfile input has 10 values, logfile a single value;
- bin 3: a value range with 10 integer values from 0 to 9.

The 10 cases explicitly:

conditioning-algorithm	physics	t	input	logfile	case
file:/conditioning-0	file:/physicsP	-1.0	file:/input-x4083	file:/log	0
file:/conditioning-0	file:/physicsP	-0.5	file:/input-x63	file:/log	1
file:/conditioning-0	file:/physicsP	0	file:/input-z762	file:/log	2
file:/conditioning-0	file:/physicsP	0.5	file:/input-x111	file:/log	3
file:/conditioning-0	file:/physicsP	1.0	file:/input-b059	file:/log	4
file:/conditioning-1	file:/physicsQ	-1.0	file:/input-z4985	file:/log	5
file:/conditioning-1	file:/physicsQ	-0.5	file:/input-a3118	file:/log	6
file:/conditioning-1	file:/physicsQ	0	file:/input-c5593	file:/log	7
file:/conditioning-1	file:/physicsQ	0.5	file:/input-x2067	file:/log	8
file:/conditioning-1	file:/physicsQ	1.0	file:/input-z4391	file:/log	9

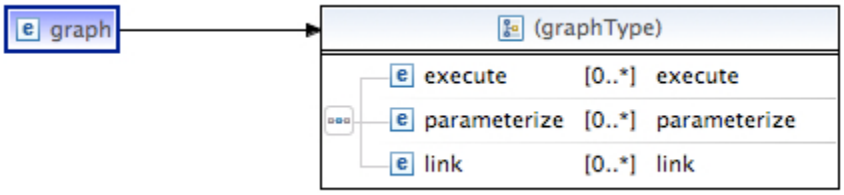
Payloads

The set of payload descriptions (i.e., executables) available for binding to the nodes in the graph.



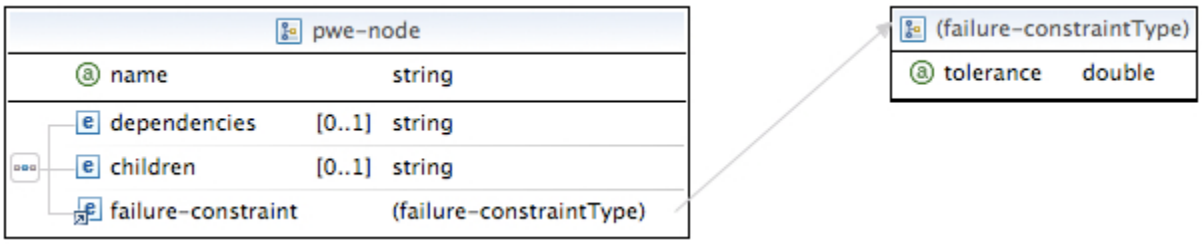
The **<script>** element is the actual script content. This is an open-ended type. Its child element could be an XML subtree, or its text could be a literal script (CDATA). Currently, the recognized element type names are `elf` and `csh`. Other types (e.g., `bash`, `perl`, `python`) may in the future be added, depending on need.

Graph



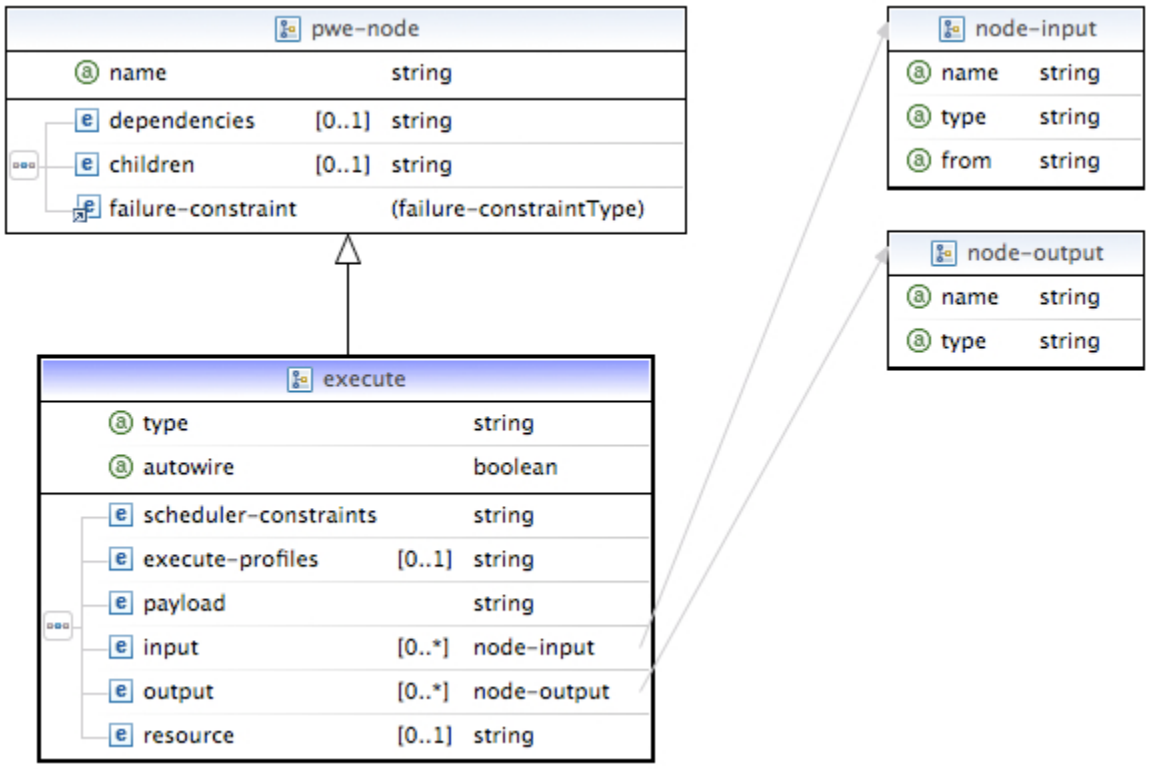
A directed, acyclic graph whose edges are untyped. In essence, this is a control-flow, not a data-flow, graph, but the (optional) use of the input/output elements on the execute nodes allows one to propagate data values (usually small values, not entire Fortran arrays!) downstream, establishing an implicit data-flow; see further on Node Input and Output below. There should be at least one node in the graph.

PWE Node



Parent type for the two concrete node types. Dependencies and children are comma-delimited lists of names referencing other nodes in the graph. See below for **<failure-constraint>**.

Execute Node



The principal type of node. It consists of a profile plus an executable script. *Scheduler-constraints* and *execute-profiles* are comma-delimited (ordered) lists of references to the workflow's respective elements; these are merged (with successive properties overwriting previous identical ones) into a single profile for this particular node. *Payload* similarly references the workflow elements. *Input* and *output* also indicate properties that belong to a node's profile. If input elements are present, dependency nodes will be searched for corresponding properties, and these will also be merged into this node's execution profile. If output elements are indicated, the job status message which returns via the update call from the ELF container will be searched for corresponding properties, and these will be added to the node's persistent profile so as to make them available to child nodes should they be required. In the case of parameterization, output values are suffixed with the parameterization index for the given member producing them; input values can specifically look for such suffixed values, or they can simply reference the prefix of the property's name, in which case all such values will be made available to the node as an array. Note that the return of output values is a feature of the ELF container; returned output is also possible when running raw shell scripts, provided the script itself writes an output file in the XML syntax required by ELF. For Ogrescript, the output name must correspond to a variable which remains in the global scope of the Ogrescript environment upon completion of its execution.

The *type* attribute is optional, and can be either "remote" (the default) or "local"; to make the node run local to the service, either set this attribute, or give it "localhost" as its resource.

Autowire refers to the generation of dependency edges. If set to *true* (the default), this will be done automatically for this node. If false, the node must be wired by providing explicit link elements (see below).

NOTE the same semantics apply to the **<resource>** element of individual execute nodes as was described above for **<global-resource>**; e.g.,

```
      <execute name="setR">
         ...
         <resource>cobalt.ncsa.uiuc.edu,tg-login.ncsa.teragrid.org</resource>
      </execute>
```

This setting would override the <global-resource> definition for this particular node in the workflow graph.

Resource variables

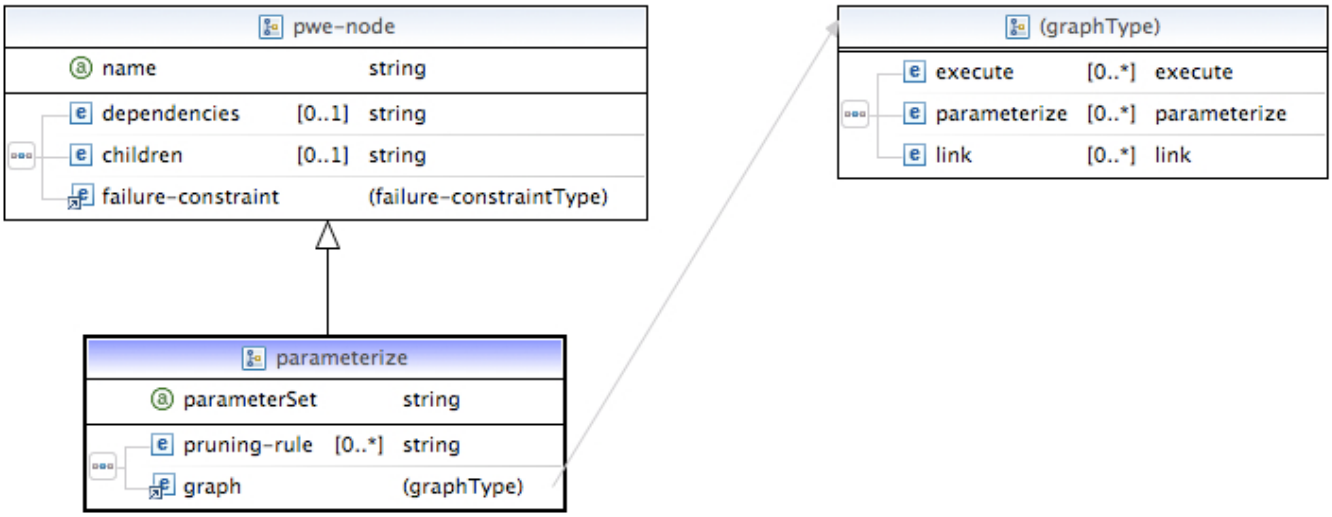
It is also possible to bind node assignments to each other using a "scheduling variable".

```
<execute name="setR" type="remote">
      <resource>@R</resource>
</execute>
...
<execute name="setS" type="remote">
      <resource>@R</resource>
</execute>
```

This variable (by convention these begin with '@' to distinguish them from actual machine node names) indicates that whatever resource has been assigned to "setR" must also be assigned to "setS".

NOTE: suppose setR gets assigned to 'cobalt.ncsa.uiuc.edu', runs, and then an attempt is made to assign setS to that node, but it fails. In this case, "setS" goes into the ATTEMPTED state, and is retried later.

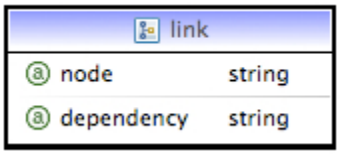
Parameterize Node



The indicated parameter set (referencing the workflow's parameter-set elements) will be used to expand the node by applying the parameters to the entire sub-graph. The latter is identical in type to the principal graph of the workflow; hence parameterization can be to an arbitrary depth.

The **<pruning-rule>** element is a constraint on the parameterization. The rule is expressed over the parameter names in the parameter set associated with this node. For instance, if the parameters include 'u' and 't', (${u} - ${t}) % 17 != 0 would be a valid rule, indicating that members should not be generated for the instances in which the rule is true. The rules are scoped by any nesting of the parameterization.

Link

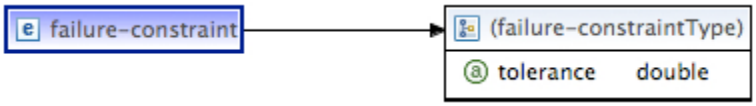


An explicit edge should be generated between these nodes (cf. autowire, above). The attributes can contain a number of wild-card expressions. Some examples:

```
   <link node="J-1" dependency="K-2-3" />       : a single link
   <link node="J-1" dependency="K-2-*" />       : J-1 to K-2-0, K-2-1, ...
   <link node="J-1" dependency="K-*-1" />       : J-1 to K-0-1, K-2-1, ...
   <link node="J-1" dependency="K-*-=" />       : J-1 to K-0-0, K-1-1, ...
   <link node="J-1" dependency="K-*-*" />       : J-1 to all K
   <link node="J-1" dependency="K-0,2-2:5" />   : J-1 to K-0-2, K-0-3, K-0-4, K-2-2, K-2-3, K-2-4
   <link node="J-0,2" dependency="K-0,2-2:5" /> : both J-0 and J-2 to the K nodes above
   <link node="J-0,2" dependency="K-@-2:5" />   : J-0 to K-0-2, K-0-3, K-0-4; J-2 to K-2-2, K-2-3, K-2-4
   <link node="J-*" dependency="K-@-2:5" />     : J-0 to K-0-2, K-0-3, K-0-4; J-1 to K-1-2, K-1-3, ... etc.
   <link node="J-*" dependency="K-@-=" />       : J-0 to K-0-0, J-1 to K-1-1, etc. [redundant if autowired]
   <link node="J-*" dependency="K-@-*" />       : J-0 to K-0-0, ..., J-1 to K-1-0 ..., etc. [redundant if
autowired]
   <link node="J-*" dependency="K-*-=" />       : J-0 to K-0-0, K-1-1 ...; J-1 to K-0-0, K-1-1, etc.
   <link node="M-*-=" dependency="N-*" />       : M-0-0 to N-0, N-1 ...; M-1-1 to N-0, N-1; etc.
   <link node="M-*-=" dependency="N-@" />       : M-0-0 to N-0, M-1-1 to N-0; etc. [redundant if autowired]
   <link node="M-1-[0-2]" dependency="N-@1"/>   : M-1-0 to N-0, M-1-1 to N-1; etc.
```

@ alone is positional; @i means index i of the child node; **WARNING**: in cases like the latter, the user must be sure that the indices referenced by @i are within the boundaries for the dependency's index.

Failure Constraint



The contents of this element, if expressed, is a comma-delimited list of dependencies which must successfully complete for this node to run. The default behavior is that all of its dependencies must be satisfied.

An alternative to expressing these constraints is to set the "tolerance" attribute; this indicates the percentage of dependencies for which failure will be allowed for this node to run. Default = 0.0. The latter is useful for programming a node which acts as a barrier on a large group of parameterized nodes, all results of which may not be necessary for the node to execute or for valid results to be obtained from the workflow. When expressed on a workflow, the tolerance constraint is applied to its leaf (that is, final, or childless) nodes.
