Java CoG Kit Karajan/Gridant Workflow Guide

Gregor von Laszewski and Mike Hategan

Software Version: 4_0_a1
Manual version: 02/18/05
Url: http://www.cogkit.org/release/4_0_a1/manual/workflow.pdf
Url: http://www.cogkit.org/release/4_0_a1/manual/workflow/workflow.html
Last update: April 12, 2005

Contents

1. About this Document

This document includes basic information about the different Java CoG Kit workflow options.

1.1. Reproduction

The material presented in this document can not be published, mirrored, electronically or otherwise reproduced without prior written consent. As you can link to this document, this should not pose much of a restriction.

1.2. Viewing

The best way to read this document is with Adobe Acrobat Reader. Please make sure you configure Adobe Acrobat Reader appropriately so you can follow hyperlinks. This is the case if you follow the default installation. Acrobat Reader is available at http://www.adobe.com/products/acrobat/readermain.html. Because the hyperlinks are not available in the printed form of this manual and we support saving our environment we strongly discourage printing this document.
We recommend that you save this manual locally on your machine and use Acrobat Reader. This has the advantage that you do not lose your anchor points while switching back and forth between different hyperlinks. An HTML version of this manual is planed, but not available yet.

1.3. Format

We have augmented the document with some comments at places where we found issues. Our intend is to address these issues in a future release. The comments are marked by the icon  and the name of the person that will work on the removal of the issue.

2. Registration

Please be a team player and support us indirectly by registering with us or reporting your use of the Java CoG Kit. Although this software is free, we still need to justify to our funders the usefulness of the projects. If you want to help us with our efforts please take a few seconds to complete this information. We do not use this information for other purposes. If you have special needs or concerns please contact gregor@mcs.anl.gov. The registration form can filled out in a variety of formats. The online form can be found at

http://www.cogkit.org/register

This form is available also as ASCII text at

http://www.cogkit.org/register/form.txt

which you can FAX to

Gregor von Laszewski, Fax: 630 252 1997

3. Introduction

4. Karajan Workflow Orchestration

Gregor: The ref and label seem missnatched

Gregor: explain the name — Gregor

Karajan is a Grid parallel task management language and an execution engine. It aims to provide the scientific community with an easy-to-use tool to define and manage complex jobs on computational Grids, while keeping scalability and offering some advanced features, such as failure handling, checkpointing, dynamic execution, and distributed execution. — Gregor
It supports a range of Grid back-end services, through the Java CoG Kit Core abstraction layer, making it easy to work with heterogeneous Grid environments.

5. Installation

5.1. Obtaining the Source Code

Karajan can be downloaded from the Java CoG Kit CVS archive. Instructions regarding the Java CoG Kit requirements and details on obtaining the Java CoG Kit sources are available in Section **??**.

5.2. Compiling Karajan

Change directory to *cog*/modules/karajan and type 'ant dist'. This will compile Karajan and all its dependencies. It will also create a *dist* directory containing the distribution of Karajan. Inside the *dist* directory, the *bin* directory will contain the necessary scripts that can be used to launch Karajan.

6. Using Karajan

There are two interfaces to Karajan:

1. The command line interface, accessible through *bin/karajan* provides a very simple interface, which is mainly non-interactive and does not provide feedback on the execution.
2. The graphical interface, which can be started through *bin/karajan-gui*, can display a graphical representation of the workflow and other progress information and statistics. It also allows a certain level of interaction.

6.1. Command Line Interface

The command line interface allows you to start the execution of a specification. The syntax is very simple:

```
>./karajan spec.xml
```

Karajan will then try to load, parse, and execute the indicated specification. Any resulting messages will be printed on the console.

6.2. Graphical Interface

The graphical interface allows for additional interaction with the execution engine. It can be started using *bin/karajan-gui*. The following command line options are supported:

-help Displays a brief usage summary

-load filename Can be used to load a script upon starting

-run Used in conjunction with *-load*, will immediately start the execution.

When the interface is started without any parameters, an empty view is presented (Figure 1).

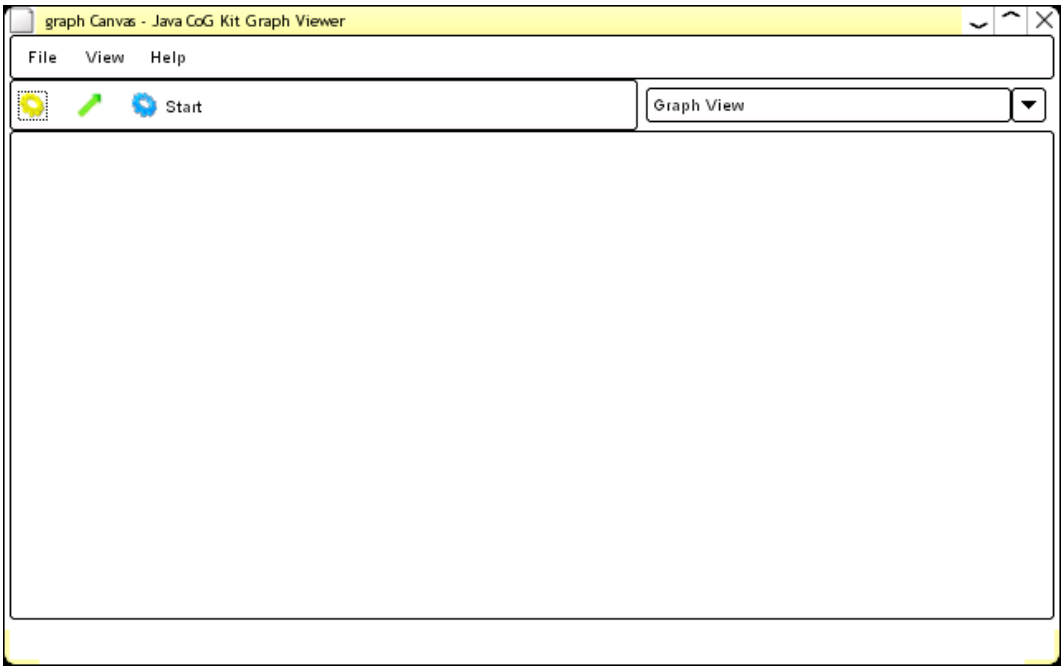


Figure 1: An empty Karajan desktop

The File->Open menu item can be used to load a script. After the script is loaded, a graph that represents the control flow of the loaded specification will be drawn. An example can be seen in Figure 2.
The execution can be started by pressing the *Start* button located on the toolbar. Once it is started, the status of each node will be visible as an overlayed image over the node icon. The following states exist:

None The node has not yet been executed

Running ( ) The node is being executed

Completed ( ) The node completed execution successfully

Failed ( ) Execution of the node failed

Breakpoint ( ) A breakpoint was set on the node

Paused ( ) The execution was paused at the current node, possibly because of a breakpoint being set on the node

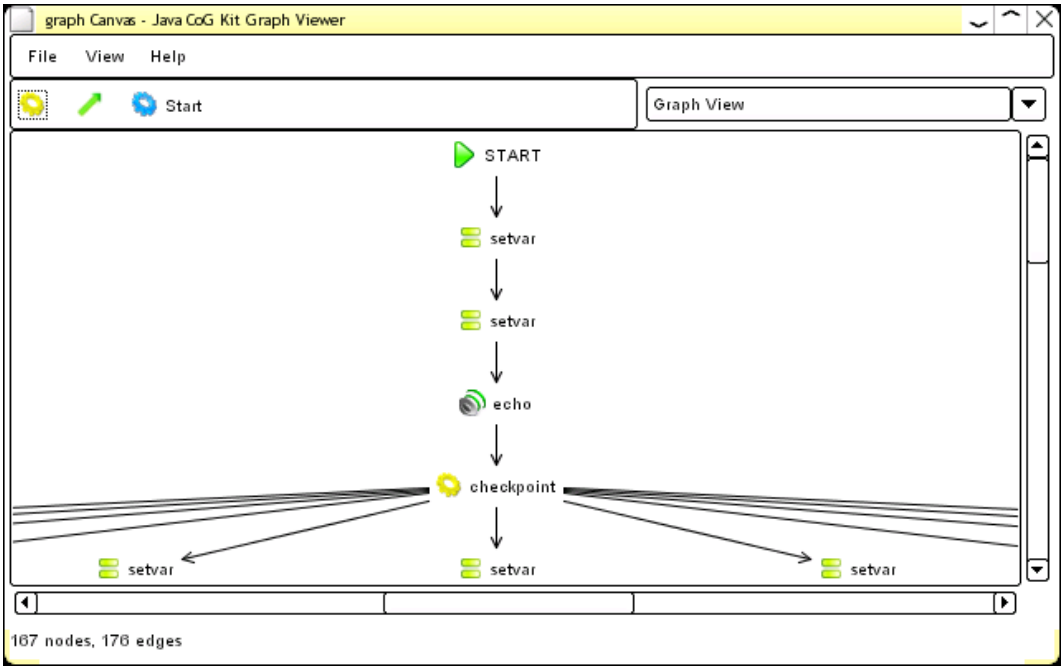


Figure 2: A script was loaded

6.2.1. Setting A Breakpoint

Breakpoints can be set by using a node's context menu. Clicking on a node with the right mouse button will pop up the menu, as it can be seen in Figure 3
Whenever the execution reaches the node where a breakpoint was set, a message dialog will pop up, and the execution of the specific thread/branch where the node is located will be suspended (see Figure 4).
The execution can then be resumed by using the context menu of the node (accessible by right-clicking on the node). In the case of a paused node, an item that will resume the execution will be present in the menu (see Figure 5).

6.2.2. Error Handling

Errors that may occur during the execution, which are not explicitly handled in the specification, will result in a dialog window that provides several options for dealing with the error. A sample error dialog is presented in Figure 6.
Each error option provided by the error dialog is described below:

Abort Passes the error to the execution engine, which will result in an error message dump on the console and the immediate termination of the execution.

Ignore Completely ignores the error as if it has never occurred.

Restart Restarts the failed node. You can also specify the number of times that the node will be restarted before the execution is aborted.

Apply to all errors of this type Whenever an identical error occurs on any node, the same action will be applied automatically.

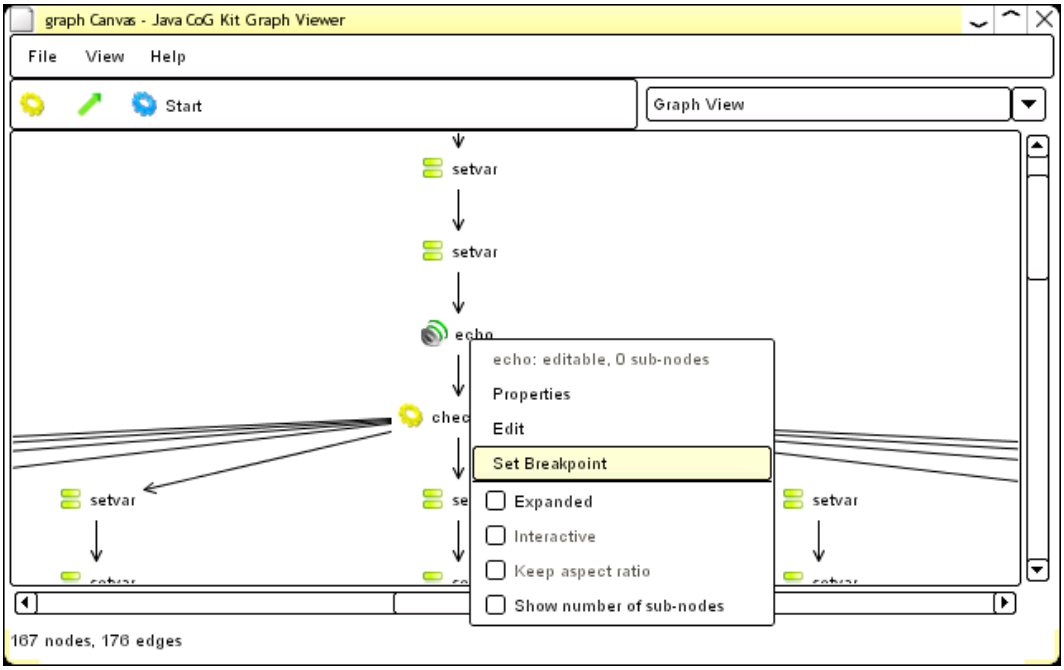


Figure 3: The node context menu

Apply to all errors for this element All other errors that occur on the node will automatically be treated with the same action.

6.3. The Karajan Service

Karajan also provides a web-service interface, which can be used to remotely control and monitor the execution. An instance of the Karajan service can be used to execute multiple specifications at one time.

6.3.1. Using the Karajan Service

The Karajan service uses the Axis (http://ws.apache.org/axis) Web-Service container. The Karajan service can be deployed inside a running instance of the Axis container by using the *service.xml* Ant build-file, which can be found in the `cog/modules/karajan` directory. In order to deploy the Karajan service, you will first need to edit the *service.properties* file, available in the same directory. After editing the *service.properties* file and making sure Axis is running, you can use the following command to deploy the service:

```
ant -f service.xml deploy
```

You can also undeploy the Karajan service with the following command:

```
ant -f service.xml undeploy
```

7. Language Specification

The Karajan specifications are written in an XML based language. Extensive information about XML is available from http://www.w3.org/XML. XML has the advantage of a very strict and well defined structure.

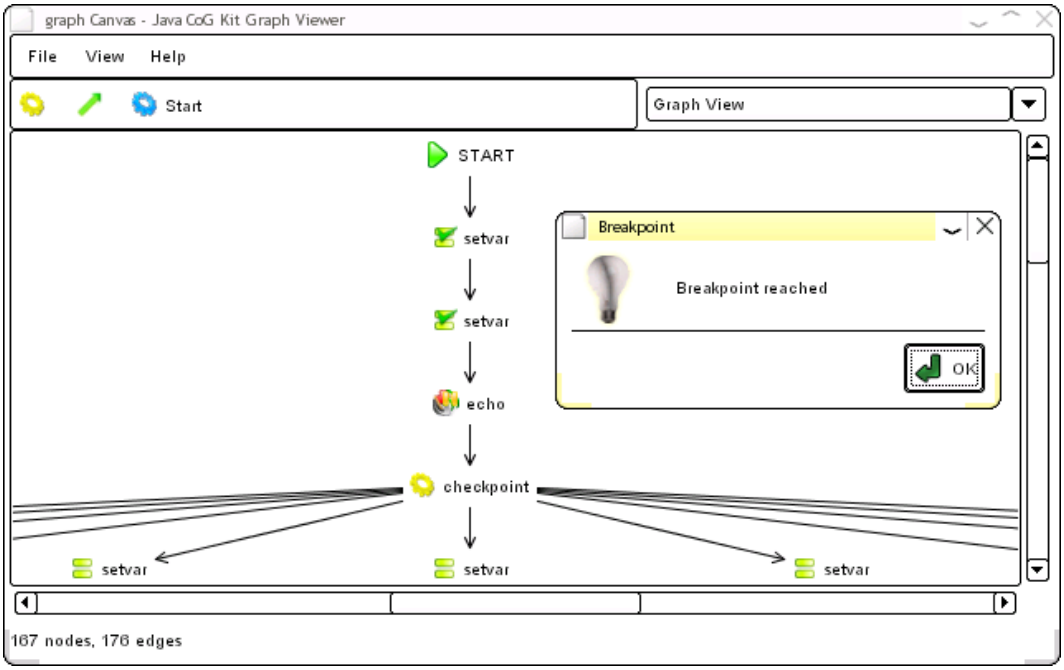


Figure 4: A Breakpoint Was Reached

7.1. Concepts

7.1.1. Elements

The building block of a Karajan specification is an XML element. The structure of Karajan specifications is very similar to that of structured languages (such as C, Java, or Pascal). Most elements can also act as containers for other elements. Each element performs a specific function, or describes how contained elements relate to each other.

7.1.2. Variables

Variables can be used in Karajan to store temporary values, values that can change and appear often in the specification, as counters for iterators, and so forth. Variable names are case-sensitive and can be any strings with the following exceptions:

- Numbers are reserved for indexed arguments (See Section7.6.6).
- Variable starting with a has symbol (#) are reserved for internal use.

Defining Variables Variables can be defined explicitly by using the *set* or *setvar* element, which takes two attributes: *name* and *value*. The following example assigns the value *blah* to the variable named *variable1*:

```
<set name="variable1" value="blah"/>
```

If the *value* attribute is not specified, *set* will use the value of the default return variable ($). This can be used for getting values from functions[1]:

```
<set name="variable2">
 <!-- read the contents of /tmp/exitcode -->
```

[1] more about functions in Section??

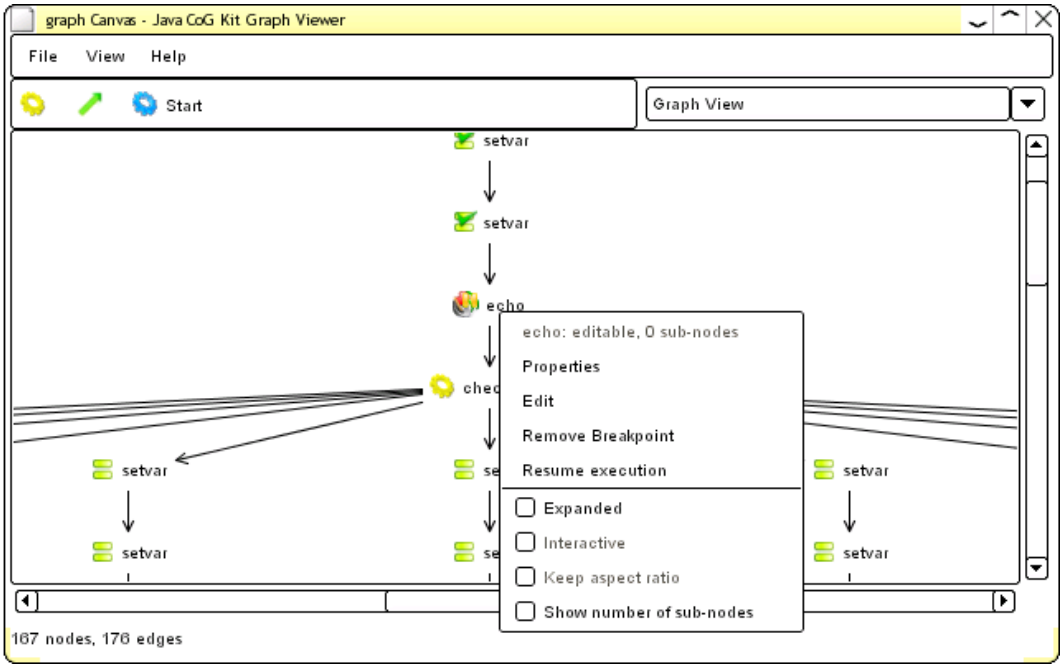


Figure 5: Resuming execution

```
  <readFile name="/tmp/exitcode" />
</set>
```

Variable Expansion Variables can be expanded inside element attributes by enclosing them inside curly brackets. Nested expansion is not possible[2].
Example:

```
<set name="variable1" value="blah" />
<echo message="variable1={variable1}" />
```

The Scope of Variables The scope of variables is limited to the element inside which they appear, unless they are shadowed in sub-elements. In such a case, the scope of the shadowed variable will be limited to the element in which the variable was defined and any sub-elements executed at runtime. In technical words, Karajan is a dynamically scoped language and uses shallow binding. The following example illustrates this:

```
<sequential>
  <!-- define the variable "var" -->
  <set name="var" value="one" />

  <!-- print its value on the console -->
  <echo message="{var}" />

  <!-- a container-->
  <sequential>
```

[2]It used to be possible to have nested expansion. The feature was though removed due to the fact that it was relatively difficult to quickly determine the exact the context in which nested expansions would take place

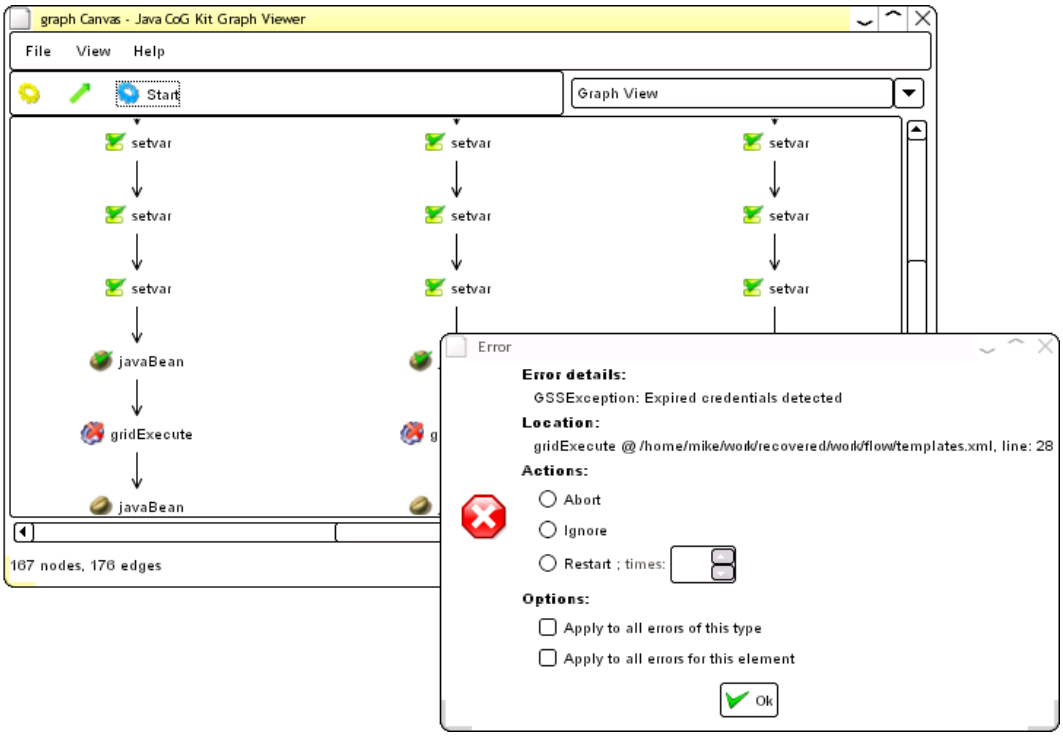


Figure 6: Error dialog

```
    <!-- override "var" -->
    <set name="var" value="two" />

    <!-- print the value on the console -->
    <echo message="{var}" />

  </sequential>

  <!-- at this point "var" will be "one" again-->
  <echo message="{var}" />

</sequential>

<!-- "var" does not exist here; an error will occur -->
<echo message="{var}" />
```

The example will produce the following output:

```
one
two
one
!!ERROR
```

In the last *echo* element, Karajan tries to expand *var*, but since it cannot be found, it will generate an error[3]

[3]Before version 0.45, if a variable was not found, the expansion would result in the literal "{variable-name}". This behavior proved to be error-prone, in particular in the cases where missing variables were passed as arguments to other elements, producing hard to trace errors.

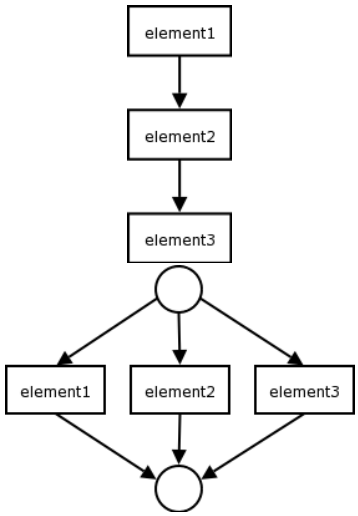
7.2. Parallelism

??

Karajan supports two basic containers through which parallelism can be manipulated, namely, *sequential* and *parallel* . Both containers are synchronous, which means that their execution will terminate when all sub-elements have finished execution. Unsynchronized execution can be acheived using the *unsynchronized* element[4]. The following examples illustrate the use of *sequential* and *parallel* containers, as well as synchronous and asynchronous execution. On the right side, an image showing the resulting control flow of the specifications on the left is shown:

Sequential execution

```
<sequential>
  <element1/>
  <element2/>
  <element3/>
</sequential>
```

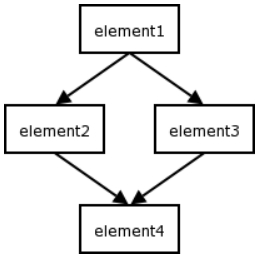


Parallel execution

```
<parallel>
  <element1/>
  <element2/>
  <element3/>
</parallel>
```

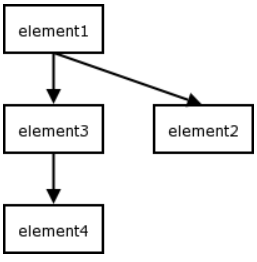

Mixed sequential/parallel execution

```
<sequential>
  <element1/>
  <parallel>
    <element2/>
    <element3/>
  </parallel>
  <element4/>
</sequential>
```



Sequential execution with asynchronous element

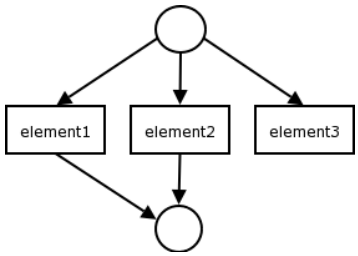
```
<sequential>
  <element1/>
  <unsynchronized>
    <element2/>
  </unsynchronized>
  <element3/>
  <element4/>
</sequential>
```



[4]Previous versions used the *sync="false"* attribute on an element to achieve the same behavior.

Parallel execution with asynchronous element

```
<parallel>
  <element1/>
  <element2/>
  <unsynchronized>
    <element3/>
  </unsynchronized>
</parallel>
```



7.3. Iterators

Iterators are used in Karajan to execute a sequence repetitively. All iterators can have either a sequential behavior (the default), in which an iteration begins execution only after the previous iteration has completed execution, or a parallel behavior (switched on using the *parallel="true"* attribute), in which all iterations are executed in parallel. The parallel behavior of an iterator does not apply to contained elements. If a parallel iterator has two sub-elements, the elements will execute in sequential order. This can be prevented by using an explicit *parallel* container inside the iterator.
Two iterators exist: *for* and *foreach* . Both have a mandatory attribute (*name*), which represents the name of the iteration variable.

7.3.1. for iterator

The *for* iterator is used for iterations across integer ranges and takes two numerical attributes, representing the first and last value. There are two equivalent ways of specifying the two attributes, as shown in the following examples:

```
<for name="iteration" from="1" to="4">
  <echo message="Iteration {iteration}"/>
</for>
```

and

```
<for name="iteration" range="1, 4">
  <echo message="Iteration {iteration}"/>
</for>
```

The result in both cases is as follows:

```
1
2
3
4
```

7.3.2. foreach iterator

The *foreach* iterator can be used to iterate across arbitrary values, or files in a directory (a feature that needs a little polishing). Iteration values can be specified by using the *in* attribute. To iterate on the files in a directory, you can use the *dir* attribute. When iterating over files in a directory, an additional attribute (*filter*) is available, which can be used to specify a filter in a regular expression form.
Examples:

```
<foreach name="iteration" in="one, two, three, four">
  <echo message="Iteration {iteration}"/>
</foreach>
```

would produce the following output:

```
one
two
three
four
```

while

```
<foreach name="file" dir="/home/johndoe" filter=".*\\.txt">
  <echo message="File {file}"/>
</foreach>
```

would list all the files ending in .txt contained in the home directory of user *johndoe*.

7.4. Conditional execution

Conditional execution is expressed using the *if* , *then* , *elseif* , and *else* elements. The format is as follows:

```
<if>
  <condition>
    <condition1/>
  </condition>
  <then>
    ...
  </then>
  <condition>
    <condition2/>
  </condition>
  <elseif>
    ...
  </elseif>
  ...
  <condition>
    <conditionN/>
  </condition>
  <elseif>
    ...
  </elseif>
  <else>
    ...
  </else>
</if>
```

The *condition1* through *conditionN* are elements that evaluate to something. Typically they are functions, such as *math:equals* , *math:le* , etc. Functionally, the *then* and *elseif* are the same. They both execute if the last specified condition can be evaluated to a boolean value of "true", or cause the enclosing *if* to terminate if the condition evaluates to a boolean value of "false".*Since version 0.25, the conditions must be enclosed inside a* condition *element*

7.5. Loops

The *while* element can be used to express a set of elements that are to be executed repeatedly. The elements contained by the *while* element are executed in sequence, repeatedly, as long as a condition is true. The condition is indicated by a *condition* element. The position of the *condition* element also determines the location at which the loop is exited:

```
<while>
  <condition>
    <false/>
  </condition>
  <echo message="You will never see this message" />
</while>
```

as opposed to:

```
<while>
  <echo message="You will see this message" />
  <condition>
    <false/>
  </condition>
</while>
```

If elements inside the loop must be executed in parallel, a *parallel* container can be used. The *condition* element can itself be executed in parallel with the rest of the elements. However, the termination condition will not be verified during the execution of any of its sub-elements:

```
<while>
  <parallel>
    <condition>
      <false/>
    </condition>
    <echo>
You will see this message because the condition
will be checked after the parallel container completes
    </echo>
  </parallel>
  <echo message="You will still not see this message" />
</while>
```

The execution of a while loop can also be terminated abruptly using the *break* element:

```
<while>
  <sequential>
    <if>
      <condition>
        <true/>
      </condition>
      <then>
        <break/>
      </then>
    </if>
    <echo message="You will not see this message" />
  </sequential>
  <echo message="You will not see this message either" />
</while>
```

Arguments in Karajan are generally expressed as XML attributes. However, XML attributes are limited. It is impossible to use attributes to express the result of executing other elements. The solution is to express such arguments as nested elements. There are four main types of argument named, variable unamed, defaukt, and typed default arguments.

7.6.1. Named Arguments

Named arguments are used to express arguments when there exist multiple arguments for an element and they differ from a semantic perspective. The *setvar* element for example, has a *name* and a *value* attribute, which cannot be commuted without changing the meaning of the program. Named arguments can be expressed both as XML attributes, and as nested elements with the *argumentName* attribute:

```
<set name="var" value="1" />
```

would produce the same result as:

```
<set value="1">
  <argument argumentName="name" value="var" />
</set>
```

7.6.2. Variable unnamed arguments

Variable unnamed arguments are used when arguments are equivalent. The *math:sum* element is such an example:

```
...
<math:sum>
  <argument value="1" />
  <argument value="2" />
  <argument value="3" />
</math:sum>
...
```

This does not necessary mean that changing the order of the arguments will not change the outcome of the execution of the element, as it can easily be seen in the following example:

```
...
<list:append>
  <argument value="1" />
  <argument value="2" />
  <argument value="3" />
</list:append>
...
```

7.6.3. default arguments

can be used for at most one of the arguments. A default argument can be expressed as an unnamed nested argument. No element can have both a default argument and multiple unnamed arguments. The *setvar* has the *value* argument as a default argument. The following code snippets lead to the same result:

```
<set name="var" value="1" />
```

```
<set name="var">
  <argument value="1"/>
</set>
```

7.6.4. Typed default arguments

Typed default arguments are an exception to some of the above rules. In some instances, an element may expect multiple arguments, each having a different type. In such cases there is no need to name the arguments, since their type will provide the required information that is needed to map the supplied arguments to the ones required. Since types exist only internally in Karajan, typed default arguments can only be used for internally defined elements.

7.6.5. Ussage of Arguments

It is generally both acceptable and useful to use arguments inside elements that describe execution flow. There is a specific set of elements, namely *sequential* , *parallel* , ELfor, *foreach* , *then* , *elseif* , and *else* which do not affect the contained arguments. The following example shows how a the sum of the results of two parallel operations can be calculated:

```
...
<math:sum>
  <parallel>
    <operation1/>
    <operation2/>
  </parallel>
</math:sum>
...
```

It is also possible to use iterators for expressing arguments, both in parallel and non parallel mode:

```
...
<math:sum>
  <for name="var" from="1" to="10" parallel="true">
    <argument value="{var}"/>
  </for>
</math:sum>
...
```

7.6.6. Indexed Arguments

Another way of expressing named arguments is by using indexed arguments. Indexed arguments are similar to the positional arguments found typically in shell scripting languages (such as BASH), with the difference that they refer to arguments that will be evaluated at a further time. They represent an index into the variable arguments passed to an element. Example:

```
...
<equals value1="{1}" value2="{2}">
  <argument value="123"/>
  <argument value="abc"/>
</equals>
...
```

In the above case,1 will evaluate to "123", while 2 will evaluate to "abc". It may again be useful to use other containers to describe the fashion in which such arguments may be evaluated:

```
...
<math:equals value1="{1}" value2="{2}">
  <parallel>
    <argument value="123" />
    <argument value="abc" />
  </parallel>
</math:equals>
...
```

Another, more interesting example is given below:

```
...
<math:equals value1="{1}" value2="{2}">
  <parallel>
    <argument value="123" />
    <if>
      <condition>
        <false/>
      </condition>
      <then>
        <argument value="abc" />
      </then>
      <else>
        <argument value="def" />
      </else>
    </if>
  </parallel>
</math:equals>
...
```

In the above case, since the condition is false, the *if* element will evaluate to whatever the *else* element evaluates to. Consequently, 2 will be substituted with "def".

7.7. Templates

Templates can be used to define reusable code and are somewhat similar to procedures in other languages. Templates can accept named parameters.

Definition of templates can be done by using the *templateDef* element. The mandatory attribute *name* specifies the name of the template. The body of the template can consist of any Karajan elements. An additional element (*default*) can be used to designate default values for parameters. A simple template definition is shown below:

```
<templatedef name="sample">
  <default name="arg1" value="default1" />
  <default name="arg2" value="default2" />
  <echo message="arg1 is {arg1}" />
  <echo message="arg2 is {arg2}" />
  <echo message="arg3 is {arg3}" />
</templatedef>
```

Template invocations can be made via the *template* element, which accepts the *name* attribute, plus any number of other arguments that are passed to the template:

```
<template name="sample"
  arg1="value1"
  arg2="value2"
  arg3="value3" />
```

All templates are re-entrant as long as no external resources are involved. Variables defined or overridden inside templates are considered local.

7.8. In-line elements

Elements can also be defined similar to templates. Since version 0.22, this is preferable to templates.

In-line element definition is achieved using the *element* element. The *name* attribute specifies the name that the definition will be bound to. The *arguments* attribute is a comma separated list of argument names that can be passed to the definition. The *vargs* attribute, if set to *true*, indicates that the element should rather take a variable length list of unnamed arguments than a set of named arguments. The following element definition creates and element that sums all numbers from *min* to *max*:

```
<element name="sum" arguments="min,max" vargs="false">
  <math:sum>
    <for name="var" range="{min},{max}">
      <argument value="{var}" />
    </for>
  </math:sum>
</element>
```

The following element definition makes use of unnamed arguments, and calculates the root mean square of the supplied values. The arguments are available as a list that can be iterated in the *vargs* variable:

```
<element name="rms" vargs="true">
  <math:sqrt>
    <math:quotient>
      <math:sum argumentName="dividend">
        <foreach name="item" in="{vargs}">
          <math:square value="{item}" />
        </foreach>
      </math:sum>

      <list:size list="{vargs}" argumentName="divisor" />
    </math:quotient>
  </math:sqrt>
</element>
```

If a definition takes only one argument and has no variable unnamed arguments, that argument will become a default argument. If there are multiple named arguments (but no variable unnamed arguments), a default argument can be specified using the *defaultArgument* attribute.

In-line element invocation is performed simply by using the element name. The following examples will invoke the previously defined *sum* and *rms* elements and print the resulting values:

```
<set name="sum">
  <sum min="1" max="10" />
```

```
</set>
<echo message="The sum is {sum}" />
```

```
<set name="rms">
  <rms>
    <argument value="10" />
    <argument value="12" />
    <argument value="8" />
    <argument value="50" />
    <argument value="11" />
  </rms>
</set>
<echo message="The rms is {rms}" />
```

7.9. Recursion

Due to the fact that side effects in Karajan have a limited scope (an element cannot change the environment of its parent, except for strictly controlled circumstances), recursion is necessary to address a certain class of problems that cannot be solved otherwise. Recursion is possible using element definition element. The following definition computes the n-th value in a Fibonacci series:

```
<element name="fibonacci" arguments="n">
  <if>
    <condition>
        <math:le value1="{n}" value2="2" />
    </condition>
    <then>
      <argument value="1" />
    </then>
    <else>
      <math:sum>
        <fibonacci>
          <math:subtraction from="{n}" value="1" />
        </fibonacci>
        <fibonacci>
          <math:subtraction from="{n}" value="2" />
        </fibonacci>
      </math:sum>
    </else>
  </if>
</element>
```

7.10. Element caching

By default the results of executing user-defined elements are cached. Subsequent invocations of a particular element with identical arguments, will result in the result value(s) being retrieved from the cache. Certain elements can however produce different results during subsequent executions. Caching is not desirable in such cases, therefore the *cache="false"* attribute can be used for the *element* element. Caching will also be disabled if variable arguments are enabled for an element.

7.11. Delayed caching

Delayed caching is a feature complementary to element caching. Should delayed caching be enabled for an element, and the element be called multiple times, in parallel, with the same arguments, only the first call will be executed. All other calls made with the same arguments will be deferred until the first call completes, and the cached value resulting from that first call will be used for the deferred calls. Delayed caching can be enabled by using the *delayedcaching=true* attribute. Caching must also be enabled for delayed caching to work.

7.12. Anonymous elements

Anonymous elements can also be defined in Karajan. Anonymous elements denote elements which do not have a name, but rather a reference which can be used to execute them at a later time with the *executeElement* element. Anonymous elements can be defined using the *element* element, but without specifying a *name* attribute:

```
...
<set name="echo-message">
  <element>
    <echo message="Important message:" />
    ...
    <echo message="Important message ends" />
  </element>
</set>

<executeElement element="{echo-message}" />
...
```

Anonymous elements are not cached.

7.13. Grid-related Elements

Karajan contains a series of elements that are divided into two main categories: Grid resource description and configuration and Grid tasks.

7.13.1. Grid Resource Description and Configuration Elements

- *scheduler* is used to select a scheduler type and specify various parameters for it. Currently only one scheduler is available (named *default*). The arguments are:

  type The type of the scheduler desired. Only *default* is available at this time.

  jobsPerCpu Sets the maximum number of tasks that the scheduler will allocate for one CPU.

  maxSimultaneousJobs Sets the total maximum number of remote tasks that the scheduler will allow at any given time.

  showTaskList If set to *true* the scheduler will pop-up a window providing a lists of tasks that are being executed, and additional task and memory statistics.

  grid Specified as a nested typed argument through the *grid* element, defines the set of resources that will be used for scheduling.

  taskHandler Specified as a nested typed argument through the *taskHandler* element, defines the type(s) of Java CoG Kit Core task handler to be used by the scheduler. If there are multiple task handlers for a scheduler, they will all be used, with a priority set by the order in which they were specified, the highest priority going to

the first. The scheduler should also try to match the handlers against available resources and use a lower priority handler if resources are not available for a higher priority handler.

- *taskHandler* defines a type of Java CoG Kit Core task handler that is going to be used by the scheduler. The attributes are *type*, which selects the type of the handler, and *provider* which indicates the provider for the handler (for a list of supported handlers, consult the Section8). The type of the provider is one of "execution" (or "job-submission"), "file-transfer", and "file-operation" (or simply "file").
- *securityContext* is used to define a security context (passwords, private/public keys, proxy location, etc.) for a service. The mandatory attribute is the *provider* attribute. A series of nested elements can exist for the *securityContext* element. These nested elements are handler specific and are described in Section 8.
- *grid* encapsulates a set of resources that will be used by the scheduler. Accepts an optional *name* attribute. A series of nested *host* arguments can be specified.
- *host* designates a single contact point (a remote host). The mandatory *name* attribute denotes the hostname of the remote contact. A *cpus* attribute allows the specification of the number of CPUs the host has. This information may be used for scheduling purposes. Each host can have multiple nested *service* elements.
- *service* defines a host service. The *provider* type of the service. The *type* attribute specifies the service type. The current possible types are the same as the ones for the *taskHandler* element. The exact details of the service are expressed in the form of a URI (the *uri* attribute). The format and details of the service URIs differ from handler to handler. Section8 provides details of all supported handlers and their details. Each service has an associated *securityContext* indicated by the *securityContext* attribute. If the attribute is missing, a default security context will be used. Please not that some providers may not have a default security context (such as SSH).

  The following example illustrates the use of the above elements:

```
<grid name="default">
  <for name="index" from="1" to="20">
    <host name="node{index}.mars.sol.mw" cpus="1">
      <service
        povider="gt2"
        type="execution"
        uri="{host}:2119/jobmanager-fork" />
      <service
        provider="gt2"
        type="file"
        uri="gsiftp://{host}:2811" />
    </host>
  </for>
</grid>
```

7.13.2. Grid Tasks

Two types of Grid tasks are available: remote execution and transfer. For all tasks that require one or more host attributes, the hosts may need to exist in the Grid definition, such that the handlers can extract service contact information. In some cases, defaults may work.

- *gridExecute* can be used to submit a remote job. The attributes are[5] as follows:

[5] attributes followed by an asterisk are optional

host* Specifies the host to which the job will be submitted. The host can be either a name pointing to a host in the grid description, such that the handler can extract the correct service information, or a complete *host* specification. If the attribute is not specified, it is up to the scheduler to pick a host for this job.

executable The executable to be run. It must exist on the remote site. If it does not, it can be transfered beforehand using a transfer task.

args* The attributes to be passed to the executable.

stdin* If input redirection is desired, this attribute can be used to specify a remote file that will be redirected to the process' standard input.

stdout* Can be used to redirect the standard output of the job to a remote file.

stderr* Used optionally to redirect the standard error stream of the job to a remote file.

type Can be used with the value "mpi" to indicate an MPI-type job.

count The number of instances of the executable. This argument only applies if submission is done through a batch queuing system supporting this option (i.e. PBS).

- *gridTransfer* is used to transfer a file from one host to another. The accepted attributes are:

  srchost The source host. Use *localhost* for the local machine. Similar to the *host* attribute for the *gridExecute* element, it must be either a pointer to a host in the grid definition or a *host* element.

  srcdir The source directory where the file can be found.

  srcfile The name of the file that is to be transfered.

  desthost The destination host. Can also be *localhost* for the local machine. The rules for the *srchost* apply.

  destdir The directory on the destination host where the file will be placed.

  destfile* Can be used to rename the file during the transfer.

- It may sometimes be necessary to execute a set of tasks on the same host. The *allocateHost* element can be used for this purpose. The *name* attribute specifies a variable that can be used inside the element by the various tasks whenever the remote host needs to be referenced. A simple example is provided below:

```
<allocateHost name="remote">
  <!-- transfer the input data-->
  <gridTransfer
    srchost="localhost"
    srcdir="/tmp"
    srcfile="in"
    desthost="{remote}"
    destdir="/tmp"/>

  <!-- do the heavy processing-->
  <gridExecute
    host="{remote}"
    executable="/usr/bin/tac"
    args=""
    stdin="/tmp/in"
```

```
    stdout="/tmp/out"/>

  <!-- transfer back the results-->
  <gridTransfer
    srchost="{remote}"
    srcdir="/tmp"
    srcfile="out"
    desthost="localhost"
    destdir="/tmp"/>
</allocateHost>
```

7.14. Explicit Error Handling

In certain cases, errors that appear in certain locations are known to have no impact on the overall execution. A typical example would be a cleanup process. In such cases, it may be preferable to be able to simply ignore errors. Other operations have particularly high rates of failure. However, subsequent re-executions of such operations may result in a successful result. The following elements deal with such cases:

- *ignoreErrors* has no attributes and any errors that occur on contained elements are ignored.
- *restartOnError* has a numeric mandatory attribute (*times*) that specifies the number of times the contained elements are restarted when an error occurs, before that error is reported.
- *generateError* will cause an error to be generated, with an associated message specified by the *message* attribute.

Gregor: Place this section at a better location. Communicate with Mike to get a specification to augment it in the manual — Gregor

Mike: This looks like the right place — Mike

Karajan includes also the ability to define customizable error handlers. You can (without java programming), intercept a "proxy expired" error, invoke the proxy initialization code, and then restart the failed element. Listing 1] shows an example on how to use a customized error handler.

Listing 1: Custom error handling in Karajan

```
<onError match="(.*Expired credentials detected.*)|(.*Proxy file.*not found.*)">
    <!-- pop up a proxy init window -->
    <echo message="Invalid GSI credentials detected. Executing proxy init..."/>
    <executeJava mainClass="org.globus.cog.karajan.util.ProxyInitWrapper"/>

    <!-- re-execute the element -->
    <echo message="Restarting failed element"/>
    <executeElement element="{element}"/>
</onError>
```

The following variables pertaining to the error are defined in an error handler:

- *element* Points to the element that caused the error.
- *error* The message of the error that occured
- *trace* A string containing a trace of the Karajan call stack

- *exception* A string with the Java exception if available, otherwise a generic message indicating that no exceptio was available

Error handlers are not reentrant. If a handler causes an error, the execution will abort immediately.

7.15. Java Elements

The Java elements allow limited interaction with the Java Virtual Machine.

- *newJavaObject* instantiates a new Java object. The class of the object is specified by using the *className* argument. A set of arguments to be passed to the constructor can also be specified through nested arguments.
- *invokeJavaMethod* invokes a method on an existing object indicated by the default argument *object*. The method name must be passed by the *method*argument. Nested arguments can be used to indicate arguments to be passed to the method.
- *executeJava* will synchronously execute a Java program. The *mainClass* attribute can be used to specify the fully qualified class name that contains the main method. The class must be present in the classpath of the instance of the Java Virtual Machine that is executing the specification. Arguments can be passed using nested elements:

```
<executeJava mainClass="org.com.net.foo.SomeClass">
    <argument value="-parameter" />
    <argument value="importantValue" />
</executeJava>
```

7.16. Arithmetic elements

- *math:sum* adds a set of numbers specified as unnamed arguments.
- *math:product* multiplies a set of numbers specified as unnamed arguments.
- *math:subtraction* subtracts two numbers specified by the *from* and *value*. Effectively calculates $from - value$.
- *math:quotient* divides *dividend* by *divisor* and returns the quotient.
- *math:remainder* computes the remainder of the division between *dividend* and *divisor*.
- *math:square* computes the square of the default *value* argument.
- *math:sqrt* computes the square root of the default *value* argument.
- *math:equals* tests if the two arguments are equal numerically. If arguments are lists, a recursive numeric comparison is performed.
- *math:gt* tests if *value1* is greater than *value2*.
- *math:lt* tests if *value1* is less than *value2*.
- *math:ge* tests if *value1* is greater than or equal to *value2*.
- *math:le* tests if *value1* is less than or equal to *value2*.

7.17. Logic elements

- *and* evaluates to the boolean *and* of the nested unnamed arguments.
- *or* calculates the boolean *or* value of the nested unnamed arguments.
- *not* negates the default *value* argument.
- *true* evaluates to a boolean value of true
- *false* evaluates to a boolean value of false

7.18. List elements

List elements can be used to construct and manipulate lists. Elements expecting a single list argument, use the default *list* argument.

- *list* see *list:append*
- *list:append* appends (concatenates) the supplied unnamed arguments in a list. The arguments can be lists. They are appended in the order they are supplied.
- *list:prepend* prepends the supplied unnamed arguments in reverse order.
- *list:concat* concatenates two or more lists
- *list:first* evaluates to the first element in the supplied list
- *list:last* evaluates to the first element in the supplied list
- *list:butFirst* evaluates to the original list with the first element removed
- *list:butLast* evaluates to the original list with the last element removed
- *list:size* evaluates to the size of the default *list* argument. If the argument is not a list, the size will be 1.
- *list:isEmpty* return true if the list is empty.

7.19. Map Elements

Map elements can be used to create and manipulate maps (hash-tables). Elements expecting a map argument use the default *map* argument.

- *map* constructs a map with the supplied map entries. If a *map* argument is specified, the entries will be added to the existing map
- *map:put* see *map*
- *map:entry* Constructs a map entry. A map entry is composed of a *key* argument and a *value* argument.
- *map:get* Retrieves from the indicated map the value corresponding to the key specified by the *key* attribute
- *map:contains* Evaluates to true if the key specified by the *key* attribute has a corresponding value in the specified map
- *map:delete* Deletes an entry from the map. The *key* attribute must be present
- *map:size* Returns the number of entries in the map

Example:

```
...
<set name="map">
  <map>
    <map:entry key="name" value="John"/>
    <map:entry key="age" value="99"/>
  </map>
</set>
<echo message="Name: {1}, age: {2}">
  <map:get map="{map}" key="name"/>
  <map:get map="{map}" key="age"/>
</echo>
...
```

- *project* is the main container of a Karajan specification. Any specification that can be executed by Karajan must have *project* as the root element.
- *echo* echoes a message on the console. The message can be specified either by using the *message* attribute or by inlining the text inside the *echo* element. The new line at the end of the message can be suppressed by setting the *nl* attribute to "false".
- *include* can be used to include reusable libraries inside a main specification. The *include* element acts during the parsing process, before the actual execution begins. The *file* attribute specifies a file, relative to the main specification file, that will be substituted for the *include* element. The included file will have its root element ignored. Section9 provides details about the include search path.
- *executeFile* loads and executes the Karajan specification indicated by the *file* attribute. The search algorithm used to find the file is identical to the one used by the *include* element. The difference between *executeFile* and *include* is that the former will perform the operations dynamically, at run-time. By contrast, *include* will load the file during the parsing sequence. With *executeFile* run-time variables can be used for the *file* attribute.
- *wait* produces a delay in the execution. One of the *delay* or *until* attributes must be set. The *delay* attribute indicates wait period in milliseconds, while the *until* attribute specifies an absolute date in a format accepted by the `java.util.Date` class.
- *equals* tests if *value1* is identic to *value2*. Unlike *math:equals* , this is not a numeric comparison. Comparing "1.0" and "1" will lead to a result of "false".
- *argument* evaluates to the value supplied by the *value* argument.
- *contains* determines whether a file contains a specific sequence of characters. The *file* attribute points to the file to be checked, while the *value* attribute specifies the value to be searched.
- *numberFormat* allows the formatting of a decimal number. The *pattern* attribute indicates the pattern to be used for formatting (as used by the `java.text.DecimalFormat` class). The *value* attribute holds the decimal value that is to be formatted.
- *readFile* reads the contents of a file, pointed to by the *name* attribute. This is intended for short text files that may possibly hold things like error messages or exit codes. The file is completely read into memory; therefore this function would not be suitable for manipulation of large files.
- *UID* generates a unique ID. The *prefix* and *suffix* arguments can be used to specify a prefix and a suffix respectively.

8. Supported Providers

Karajan supports any provider that the Java CoG Kit Core supports. However, some providers may require particular security settings, which must be known, or things will not work. The following is quick list of providers. For a complete reference, plase consult the **??**

?: We need list with providers and capabilities in core

gt2 supports execution and file operations (through gridftp)

gridftp supports gridftp services/operations

ftp supports file operations on FTP servers

?: What are the security context attributes?

ssh supports execution and file operations. The following attributes can be used by the ssh security context: *ssh-username*, *ssh-password*, *ssh-private-key*, and *ssh-passphrase*:

```
<securityContex name="ssh-doe">
  <property
    name="ssh-username"
    value="johndoe" />
  <property
    name="ssh-private-key"
    value="/home/johndoe/.ssh/identity" />
  <property
    name="ssh-pass-phrase"
    value="guessme" />

  <!-- "ssh-password" could also be used instead of-->
  <!-- the ssh-private-key/ssh-pass-phrase pair -->
</securityContext>
```

local a sample provider that supports execution and file operations on the local host

gt3.0.2 an execution provider for Globus Toolkit version 3.0.2 services (OGSA).

gt3.2.0 supports execution on Globus Toolkit version 3.2.0 services

gt3.2.1 supports execution on Globus Toolkit version 3.2.1 services

webdav a file provider for webdav services

9. Include Search Path

When the *include* element is used, the specified file is first searched in the directory where the main specification file is located. If the requested file is not found, the include search path is iterated until the file is found. The include search path is defined in *etc/karajan.properties*. The list of directories is separated by colons. A special token, *@classpath*, indicates Karajan should try to find the file in the JVM class path.
By default, Karajan starts with a very bare set of elements defined. To access most of the elements described in this manual, you should include the *sysdefaults.xml* file in the beginning of the specification:

```
<project name="myproject">
  <include file="sysdefaults.xml" />
  ...
</project>
```

10. Architecture

This section explains the main architectural characteristics of Karajan.

10.1. The Loading Process

Only basic structural and syntactic validation is being performed at load time. Semantic validation is performed individually at execution time by each execution element.
A one-to-one mapping of the XML document elements and flow elements is done by using an element map which provides the correspondence between XML element names and fully qualified flow element class names. An exception applies to the *include* element, which immediately after being loaded instructs the loader to parse the included file, the contents of which is in-lined in the current element tree.

There are two important notions in Karajan. One is the execution element (or flow element), which (as outlined in the previous subsection) is constructed from XML elements in the specification. The second one is the event. Events are used either to notify elements about the status of the execution of other elements, or to instruct elements to perform certain actions (such as start or restart execution). Events also encapsulate the state of the overall execution through a variable stack. The stack contains the complete run-time state for a specific thread of execution. There should be no deterministic difference in the execution of two different instances of the same type of element, with the same attribute values, that receive equivalent events (with identical states).
Elements are static as far as the execution is concerned. Their internal state may change during the execution, but the execution state must not be influenced by the internal state of the elements. They react to events and can use the stack passed through the events to manipulate the state of the execution. Each element that is being executed can add a frame to the variable stack. The frame can be used to store variables that can represent the state of the element. These variables are also accessible to contained elements. When the execution of an element ends, it destroys the frame that it created, and together with it the variables that it contained. This behavior is not enforced, but it is recommended. It is however mandatory that the number of frames is the same before an element begins execution and after the same element has completed execution.
In the case of parallel containers, each parallel thread will start with a copy of the stack. The stack copies will internally share frames that are not write accessible to the threads. A diagram detailing the stack model can be seen in Figure 7. The conventional representation *pop - (a b – a)* for a stack indicates that *a* and *b* were present on the stack before the execution of *pop*, and only *a* was left afterwards.
The above may in some cases be insufficient. Certain variables need to be made accessible to the parent frames such that they can be used by subsequent elements that are not descendants of the element which defined those variables. This is still possible only in a sequential context. There is no way to propagate information from one thread to the other. While this characterizes the workflow execution, the applications themselves can still use inter-thread or interprocess communication or messaging as needed.
A distinction exists between Karajan threads and Java or OS threads. The Karajan threads differ from each other only by the variable stacks they receive. No assumption can be made about the Java or OS thread in which a Karajan thread executes. The events that are passed between elements are managed by an event dispatcher (which may use more than one Java thread). The appearance of parallelism is achieved through the fact that elements either take a short time to execute or they make use of their own Java threads if known to take a longer time to execute. The result is the ability to execute a large number of Karajan threads, without the overhead required by Java/OS threads. As an example, each Java thread requires a minimum of 96 kilobytes of memory just for the thread stack.
Karajan defines six event types:

START tells an element it should start execution.

RESTART tells an element which has not completed execution yet that it should restart its execution.

EXECUTION_STARTED sent by an element immediately after it has started execution.

EXECUTION_COMPLETED sent by an element after it completes execution. This event is sent as a result of receiving the END element and after cleanup is done.

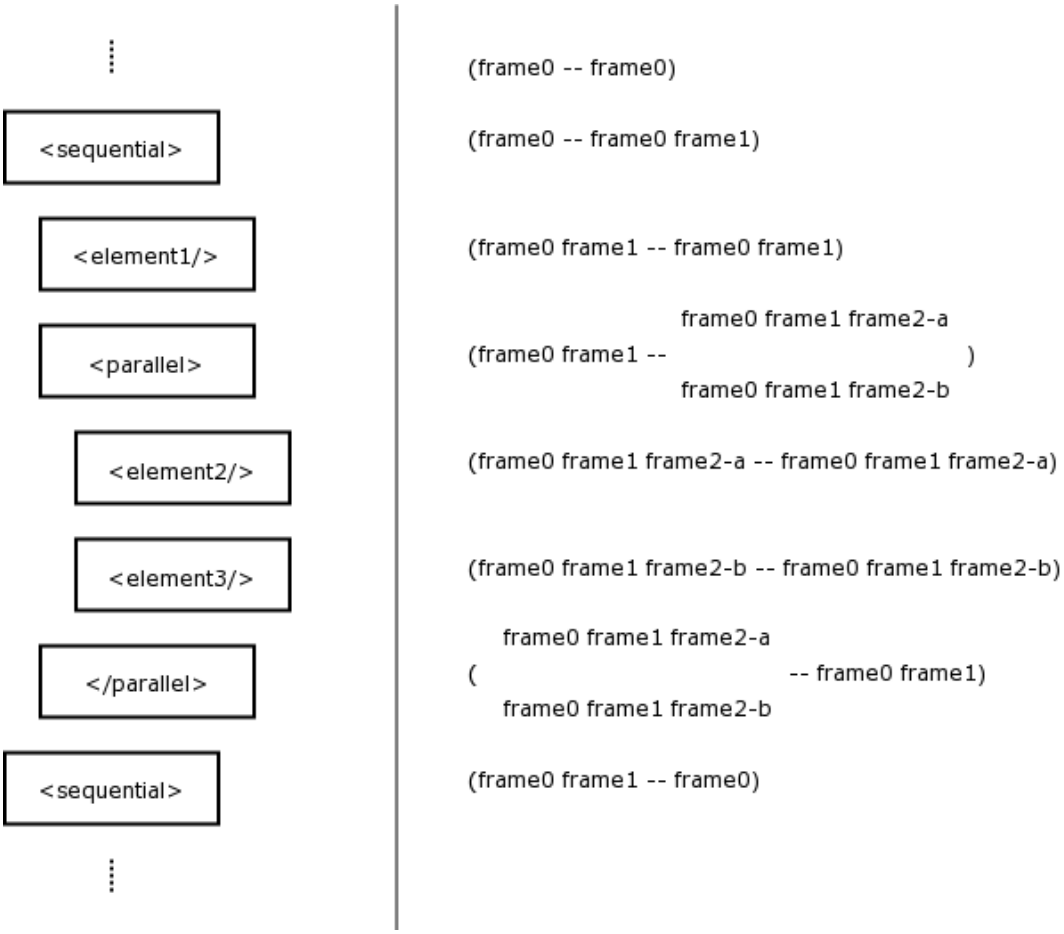


Figure 7: The stack model

EXECUTION_FAILED generated by an element when the execution failed. The frame created by the element should be popped from the stack before the event is sent.

EXECUTION_RESTARTED generated after receiving a restart event.

An example of the execution model for a sequential and a parallel container can be seen in Figure 8, and Figure 9, respectively. [6]

10.3. Task Scheduling

Task scheduling on Grid resources is done by using a scheduler that in turn uses the Java CoG Kit Core Grid abstraction layer. The *gridExecute* and *gridTransfer* elements submit the requests for execution to the scheduler which enqueues these requests and executes them as resources become available. It is up the the scheduler to manage both local and remote resources in order to ensure that these resources are not overloaded. However, certain parameters can be passed to the scheduler (using the *scheduler* element) that can alter the way in which the resources are allocated from the defined pool (the *grid* element). The scheduler also chooses the proper handlers and services for a given task.

[6] For space and readability considerations, the EXECUTION_COMPLETED event type was shortened to COMPLETED in the images

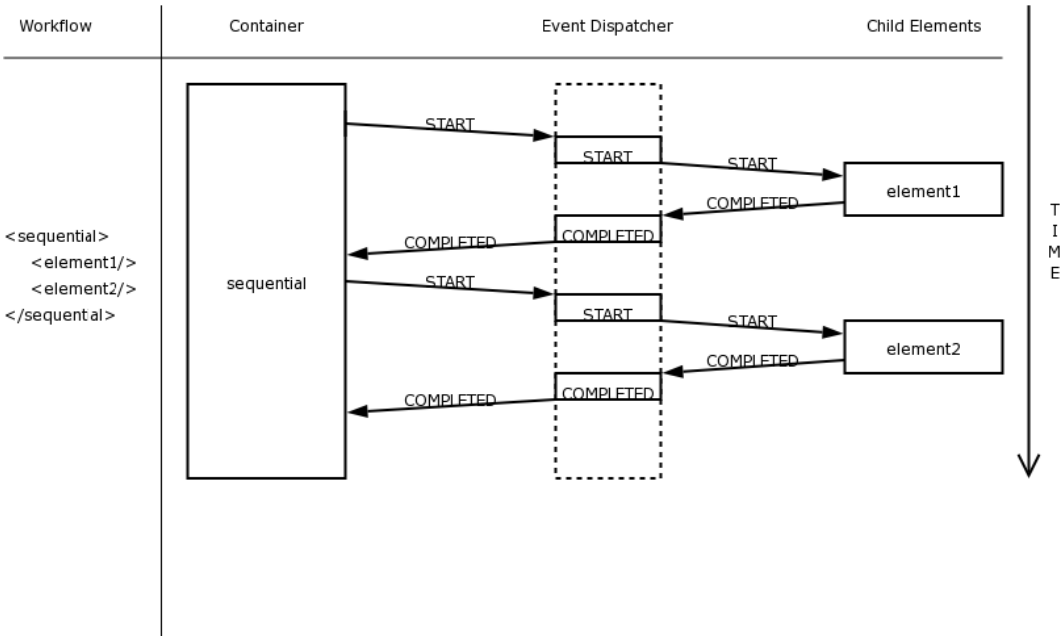


Figure 8: Execution of sequential elements

Tasks may or may not have certain constraints associated with them. Some tasks may have predefined resources or handlers that they require. For example, a certain job submission may have a predefined resource that it needs to run on. In such a case, the scheduler should not attempt to find another resource for the task. However, when a task does not specify such constraints, the scheduler must fill the missing parts required to execute the task. The scheduler must also take care of task encapsulation. This refers to the case when certain tasks must be executed on the same resource.
When trying to submit a task, the default scheduler cycles through the list of available resources and uses the first one that it finds suitable for the given task. The resource search for the next task begins with the resource immediately following the last used resource in the list. If the end of the list is reached, the search continues from the beginning of the list. If after one complete cycle through all the elements in the list, nothing suitable is found, the execution is postponed for a later time, when some of the resources may become free.
The scheduler does not take care of dependencies between tasks or the order of the execution of tasks. It is up to the workflow engine to do so. For the default scheduler, once a set of tasks is queued in the scheduler, there is no way to know anything about the order in which the execution of these tasks will begin, nor about the order in which they will complete their execution. Of course, other schedulers for which such things are known can be written, but the scheduler interface does not define explicit ways for enforcing execution order, nor it is required by the workflow engine from the scheduler that such order be known or be specifiable.

11. Checkpointing

Checkpointing is still an experimental feature in Karajan. This section describes the basic workings of checkpointing in Karajan. Checkpointing parameters can be adjusted by using the *checkpoint* element. Checkpointing here refers to Karajan checkpointing. Only the state of the workflow is saved in a checkpoint. The state of grid tasks is not included in the

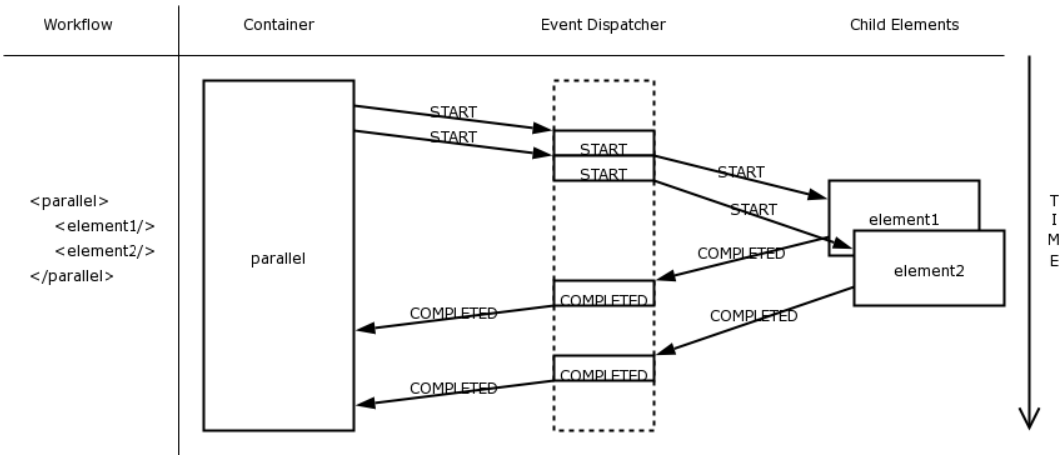


Figure 9: Execution of parallel elements

checkpoint. Imagine the following scenario (in chronological order): a Karajan specification creates certain files on a remote resource, the execution is checkpointed and interrupted, the files then are deleted, and the execution state is restored from the checkpoint. If the files are further needed and referenced in the specification, an error will eventually occur.

11.1. Checkpoint Creation

Checkpointing works by dumping the specification and the state to a file. The specification consists of the element tree and is similar to the initial source after all the *include* elements have been processed. The state of of the execution is composed of two main areas: the set of events that are waiting to be delivered, and the state of elements that have begun execution but have not yet completed it.

When a checkpoint is requested, the checkpoint manager first locks the event dispatcher in order to guarantee that the state of remains consistent during the checkpointing process. While the event dispatcher is locked, it does not deliver events, nor does it accept new events. Threads that are trying to post events to the dispatcher are suspended during this time. The checkpoint manager also keeps track of elements that have been started but were not completed and also keeps a reference to the stack of those elements. Since the checkpoint manager does not make a full copy of the element stack (for performance considerations), it may sometimes be the case that an element can at specific moments modify the stack and leave it in an inconsistent state. A special locking mechanism that allows an element to group operations on the stack that should be atomic is provided. The checkpoint manager will therefore wait until all elements have completed the execution of blocks that need to be atomic relative to the stack, before making the actual checkpoint. It can be easily seen that posting an event to the dispatcher inside a atomic block could cause a deadlock. It is thus prohibited to do so.

After the checkpoint manager has ensured that the overall state of the execution is in a consistent state, it begins writing the specification, events, and list of currently executing elements to the checkpoint file. Each event and running element has an associated stack, which will also be serialized. It is mandatory that all elements put only Java Beans on the stack; otherwise, variables on the stack will not be saved, leading to an incomplete checkpoint.

11.2. Restoring from a Checkpoint

When invoked with a checkpoint file from the command line, Karajan will automatically detect the checkpoint and restore the state of the execution at the time the checkpoint was taken. A checkpoint file is self-contained and does not require the original specification.
The restoration process is done by first loading the specification from the checkpoint file. Afterwards, pending events are deserialized and posted to the event dispatcher. Elements that were executing at the time the checkpoint was taken are also sent a RESTART event using the associated stack that was saved during checkpointing. This will effectively put the execution in the state it was at the time the checkpointing was done.

12. New additions that need to be integrated

Mike: There are a few things to be integrated. In the next release.

Mike

12.1. Workflow Example

It can go in any directory.

```
<project>
    <include file="sysdefaults.xml" />

    <global name="SSHAUTH">
        <SSHSecurityContext>
            <publicKeyAuthentication
                username="neuman"
                privatekey="/home/neuman/.ssh/identity"
                passphrase="******" />
        </SSHSecurityContext>
    </global>

    <!-- definitions for a, b, c, and d -->
    <element name="a">
        <gridTransfer
                srcfile="flow.xml"
                desthost="hot.mcs.anl.gov"
                provider="ssh"
                securityContext="{SSHAUTH}" />
    </element>

    <element name="b">
        <gridExecute
                executable="/usr/bin/tac"
                arguments="flow.xml"
                stdout="flow.reversed"
                host="hot.mcs.anl.gov"
                provider="ssh"
                securityContext="{SSHAUTH}" />
    </element>

    <element name="c">
        <gridExecute
                executable="/usr/bin/wc"
                arguments="flow.xml"
                stdout="flow.count"
```

```
                host="hot.mcs.anl.gov"
                provider="ssh"
                securityContext="{SSHAUTH}" />
    </element>

    <element name="d">
        <parallel>
            <gridTransfer
                srchost="hot.mcs.anl.gov"
                srcfile="flow.reversed"
                desthost="localhost"
                provider="ssh"
                securityContext="{SSHAUTH}" />
            <gridTransfer
                srchost="hot.mcs.anl.gov"
                srcfile="flow.count"
                desthost="localhost"
                provider="ssh"
                securityContext="{SSHAUTH}" />
        </parallel>
    </element>


    <!-- the actual workflow -->
    <a/>
      <parallel>
            <b/>
            <c/>
      </parallel>
    <d/>
  </project>
```

Needs location in CVS

12.2. Run the workflow

```
karajan flow.xml
```

(assuming that src/cog/dist/cog-4_0_a1/bin is in the PATH)
or use

```
karajan-gui -load flow.xml
```

12.3. sheduler.xml

```
<lib name="scheduler">
  <include file="sysdefaults.xml" />

  <set name="ssh-credentials">
        <SSHSecurityContext>
            <publicKeyAuthentication
        username="neuman"
        privatekey="/home/neuman/.ssh/identity"
        passphrase="***" />
        </SSHSecurityContext>
```

```
  </set>

  <scheduler type="default" jobsPerCpu="10">
    <taskHandler type="execution" provider="ssh" />
    <taskHandler type="file" provider="ssh" />
    <grid name="default">
      <host name="hot.mcs.anl.gov">
        <service type="execution"
         provider="ssh"
         url="{host}"
         securityContext="{ssh-credentials}" />
        <service type="file"
                 provider="ssh"
                 url="{host}"
         securityContext="{ssh-credentials}" />
      </host>
    </grid>
  </scheduler>
</lib>
```

References

[1] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java Commodity Grid Kit," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8-9, pp. 643–662, 2001. [Online]. Available: http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--cog-cpe-final.pdf

[2] "Java CoG Kit Wiki," 2004. [Online]. Available: http://www.cogkit.org/wiki

[3] "Java CoG Kit Registration," 2004. [Online]. Available: http://www.cogkit.org/register

Additional publications about the Java CoG Kit can be found as part of the vita of Gregor von Laszewski `http://www-unix.mcs.anl.gov/~laszewsk/vita.pdf`. Most documents are avalable online if you follow the links. In future we intend to provide this information without Gregors vita data.
If you need to cite the Java CoG Kit, please use [1].

Appendix

A. Quick Element Reference

Default arguments are indicated by a plus sign (+). Optional arguments are indicated by an asterisk (*).

A.1. allocateHost

allocateHost Defines a token that can be used to guarantee that a set of tasks will be executed on the same resource.

Arguments

name The name of the variable that should be set with the value of the token. The token can then be used by the *gridExecute* and *gridTransfer* elements as a host attribute.

A.2. and

and Performs a boolean *and* operation.

Arguments

Nested arguments: The terms.

A.3. argument

argument Evaluates to the value supplied as argument.

Arguments

value+ The argument.

A.4. break

break Can be used to abruptly terminate the enclosing *while* loop.

A.5. checkpoint

checkpoint Sets checkpointing parameters or forces the immediate creation of a checkpoint.

Arguments

fileName The name of the file to which the checkpoint will be written.

interval Sets the interval at which regular checkpoints will be performed. The interval is specified in seconds.

now If set to *true* causes the immediate creation of a checkpoint. This is merely a debugging feature. The recommended method is to set a regular interval for checkpointing.

A.6. condition

condition Encloses a condition used by the *if* and *while* elements.

A.7. contains

contains Tests whether a file contains a certain text.

Arguments

file The file to be searched

value The value to be searched

A.8. default

default Typically used to define the default value for an argument in a template. It sets the value of the specified variable if it is not already defined.

Arguments

name The name of the variable to be defined

value The value of the variable

A.9. echo

echo Echoes a message on the console

Arguments

message The message to be echoed

<***inline text***> Can be used instead of the *message* attribute for larger chunks of text

A.10. element

element Used to specify a user defined element. Consult Section7.8

Arguments

name The name of the element that is being defined.

arguments A list of named arguments that the element accepts

vargs Set to true if the element accepts variable unnamed arguments. These will be available through the *vargs* variable.

A.11. elementDef

elementDef Defines a new element using a Java class

Arguments

className The fully qualified Java class name of the element

nodeType The XML element name to be defined

A.12. elementList

elementList Returns a list with the contained elements. This is a convenience element. The following are equivalent:

```
<elementList>
  <argument value="1"/>
  <argument value="2"/>
</elementList>
```

and

```
<list>
  <element>
    <argument value="1"/>
  </element>
  <element>
    <argument value="2"/>
  </element>
</list>
```

A.13. else

else Encapsulates a set of elements that will be executed if no other condition evaluates to true inside an *if* element. Details about conditional execution can be found in Section7.4

A.14. elseif

elseif Encapsulates a series of elements that will be executed if the condition specified prior to the *elseif* element evaluates to true. Further details about conditional execution are available in Section7.4

A.15. equals

equals Performs an object comparison on two objects. It can for string comparisons in particular.

Arguments

value1 The first argument.

value2 The second argument.

A.16. executeElement

executeElement Starts a Karajan anonymous element**??** and waits for its completion

Arguments

element The element to execute

<***varies***> Arguments to pass to the element

A.17. executeFile

executeFile Loads, parses and executes a Karajan specification

Arguments

file A file to execute.

A.18. executeJava

executeJava Executes a Java application in a separate thread. The element completes execution when the application completes execution.

Arguments

mainClass The fully qualified name of the class that contains the main method.
Nested arguments: Arguments to be passed to the constructor

A.19. false

false Represents a false boolean value

A.20. for

for Iterates across a range of integer values

Arguments

from Used in conjunction with the *to* attribute indicates the first value of the iteration

name The name of the variable that is set with the current iteration value

parallel If set to *true* the iterations will be executed in parallel, otherwise they will be executed sequentially

range A range of the form *n, m* describing all integers between n and m (inclusive)

to Used together with the *from* attribute, indicates the last value of the iteration

A.21. foreach

foreach Iterates across a sequence of discrete values

Arguments

dir Points to a directory. The iteration will be performed by using the files in the specified directory.

filter A regular expression that can be used to filter files in a directory

in A comma separated list of strings that will be used as iteration values

parallel If set to *true* the iterations will be executed in parallel, otherwise they will be executed sequentially

A.22. generateError

generateError Causes an error to be generated

Arguments

message Sets the message associated with the error

A.23. grid

grid Encapsulates a collection of Grid resources that are used by the scheduler to schedule tasks.

A.24. gridExecute

gridExecute Executes a job on the Grid.

Arguments

args The arguments to be passed to the executable

directory The directory on the remote resource to execute the job in

executable A path to an executable on the remote resource

host A resource on which the job will be executed. If left empty, the scheduler will choose a resource. If a resource token (see *allocateHost*) is used, the job will be executed on the resource that the token resolves to.

stderr A path to a file on the remote resource to which the standard error stream of the executable is to be redirected

stdin A path to a file on the remote resource that will be redirected to the standard input of the executable

stdout A path to a file on the remote resource that will be used to redirect the standard output stream of the job.

type Use *type="mpi"* for an MPI job.

count The number of instances of the executable to be started.

A.25. gridTransfer

gridTransfer Used to transfer a file on the Grid

Arguments

destdir The destination directory

destfile The name of the destination file

desthost The destination resource

srcdir The source directory

srcfile The source file

srchost The source resource

A.26. host

host Describes one resource in the Grid definition

Arguments

cpus The number of CPUs that the resource has

name The host name of the resource

A.27. if

if Allows conditional execution. Section 7.4 provides additional details.
See also: *condition then elseif else*

A.28. ignoreErrors

ignoreErrors Causes any errors generated by contained elements to be ignored.

A.29. include

include Parses the contents of a file inserting the elements after the position of the *include* element.

Arguments

file The file to be included

A.30. invokeJavaMethod

invokeJavaMethod Invokes a method on a Java object.

Arguments

object+ An object instance to invoke the method on. The *newJavaObject* element can be used to create such an instance.

method The name of the method to invoke.

Nested arguments: Arguments to pass to the method.

A.31. list

list See: *list:append*

A.32. list:append

list:append Appends items to a list.

Arguments

list* The list to append to

Nested arguments: Lists or items to be appended

A.33. list:butFirst

list:butFirst Returns a list consisting of all items from the original list, but the first

Arguments

list+ The original list

A.34. list:butLast

list:butLast Returns a list consisting of all items from the original list, but the last

Arguments

list+ The original list

A.35. list:concat

list:concat Concatenates two or more lists

Arguments

Nested arguments: The lists to concatenate

A.36. list:first

list:first Returns the first element in a list

Arguments

list+ The list

A.37. list:isEmpty

list:isEmpty Checks if a list is empty

Arguments

list+ The list to be checked

A.38. list:last

list:last Returns the last element in a list

Arguments

list+ The list

A.39. list:prepend

list:prepend Appends items in reverse order.

Arguments

list* The list to prepend to

Nested arguments: Lists or items to be prepended

A.40. list:size

list:size Evaluates to the size of the list argument.

Arguments

list+ A list or an item. In the later case, the size will be 1.

A.41. map

map Constructs a map (hash-table)

Arguments

map* A map to attach the entries to. If not specified, an initially empty map will be used

Nested arguments: A set of*map:entry* items

A.42. map:contains

map:contains Verifies if a map contains a value associated with a key

Arguments

map+ The map to work on

key The key to test

A.43. map:delete

map:delete Deletes an entry from a map

Arguments

map+ The map to work on

key The key of the entry that is to be deleted

A.44. map:entry

map:entry Represents an entry in a map

Arguments

key The key of the entry

value+ The value corresponding to the key

A.45. map:get

map:get Retrieves a value from a map

Arguments

map+ The map to retrieve the value from

key The key whose associated value is to be retrieved

A.46. map:put

map:put See: *map*

A.47. map:size

map:size Returns the number of entries in a map

Arguments

map+ A map

A.48. math:equals

math:equals Performs a numeric comparison of the arguments.

Arguments

value1 The first argument.

value2 The second argument.

A.49. math:ge

math:ge Tests if the first argument is greater than or equal to the second argument.

Arguments

value1 The first argument.

value2 The second argument.

A.50. math:gt

math:gt Tests if the first argument is greater than the second argument.

Arguments

value1 The first argument.

value2 The second argument.

A.51. math:le

math:le Tests if the first argument is lower than or equal to the second argument.

Arguments

value1 The first argument.

value2 The second argument.

A.52. math:lt

math:lt Tests if the first argument is lower than the second argument.

Arguments

value1 The first argument.

value2 The second argument.

A.53. math:product

math:product Multiplies a set of numbers.

Arguments

Nested arguments: The values to be multiplied

A.54. math:quotient

math:quotient Divides two real numbers.

Arguments

dividend The dividend

divisor The divisor

A.55. math:remainder

math:remainder Evaluates the remainder of a division operation.

Arguments

dividend The dividend

divisor The divisor

A.56. math:sqrt

math:sqrt Computes the square root of a number.

Arguments

value+ The number for which the root square is to be calculated.

A.57. math:square

math:square Computes the square of a number.

Arguments

value+ The number to be squared.

A.58. math:subtraction

math:subtraction Subtracts two numbers (evaluates from − value).

Arguments

from The minuend

value The subtrahend

A.59. math:sum

math:sum Adds a set of numbers.

Arguments

Nested arguments: The values to be summed

A.60. newJavaObject

newJavaObject instantiates a new Java object.

Arguments

className the class name of the object to be instantiated.

A.61. nonCheckpointable

nonCheckpointable Has no functional purpose. It is generated inside serialized versions of events in the locations where non checkpoint-able elements are found. An example of such an element is *include* which serves its purpose during the parsing process and has no further function afterwards.

A.62. not

not Performs a boolean negation.

Arguments

value+ The argument

A.63. numberFormat

numberFormat Formats a number according to the specified pattern.

Arguments

pattern The pattern according to which the number is formatted. The pattern has the syntax used by `java.text.DecimalFormat`

value The value to be formatted

See also:http://java.sun.com/j2se/1.4.2/docs/api/java/text/DecimalFormat.html

A.64. or

or Performs a boolean *or* operation.

Arguments

Nested arguments: The arguments.

A.65. parallel

parallel Executes the contained elements in parallel

A.66. project

project The root container of a main workflow file.

Arguments

name The name of the project

A.67. readFile

readFile Reads a file and stores the contents into a variable. This function is intended for small files.

Arguments

file The file to be read

A.68. restartOnError

restartOnError Restarts the execution of the contained elements if an error is generated

Arguments

times Indicates the maximum number of times the contained elements should be restarted in case of an error before the error is reported.

A.69. scheduler

scheduler Specifies the type and parameters for the scheduler that is going to be used to schedule Grid tasks.

Arguments

type Indicates the type of the scheduler. Details about available schedulers can be found in Subsection 10.3.

<***varies***> Attributes to be passed to the scheduler.

A.70. service

service Defines a service for a resource

Arguments

type The type of the service. Currently the accepted values are *job-submission* and *file-transfer*.

uri A URI indicating the location of the service.

provider A provider that is matched against the provider attribute of the defined task handler(s).

A.71. securityContext

securityContext Used as a nested element of *taskHandler* to define a security context for the handler.

Arguments

provider Indicates the provider of the security context. For details consult Section8

A.72. securityContextProperty

securityContextProperty Defines a property for a security context.*Deprecated. Use* property

Arguments

name The name of the property

value The value of the property

A.73. sequential

sequential Executes the contained elements in sequential order

A.74. set

set Sets the value of a variable

Arguments

name The name of the variable

value The value of the variable

A.75. setvar

setvar See: *set*

A.76. taskHandler

taskHandler Defines a task handler that can be used by the scheduler to execute tasks.

Arguments

type The type of the handler. Valid types are described in Section8

provider A Java CoG Kit Core provider

A.77. template

template Invoked a template that was previously defined using*templateDef*

Arguments

name The name of the template to be invoked

<***varies***> Arguments to be passed to the template

A.78. templateDef

templateDef Defines a template

Arguments

name The name of the template to be defined

A.79. then

then Encapsulates a series of elements that will be executed if the condition specified prior to the *then* element evaluates to true. Further details about conditional execution are available in Section7.4

A.80. true

true Represents a true boolean value

A.81. UID

UID Generates a unique numeric ID. A prefix and suffix can be specified using the *prefix* and *suffix* arguments.

A.82. unsynchronized

unsynchronized An container that returns immediately, but continues the execution of the contained elements in background.

A.83. wait

wait Delays the execution for a period of time or until a specific time

Arguments

delay The delay in milliseconds to wait

until A string representing a date. The format of the date is any format accepted by the `java.util.Date` class

See also: `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Date.html`

A.84. while

while Can be used to execute an iterative loop as long as a *condition* is true.
See also: *break*

B. Java CoG Kit Guides

Short Title	Description	Format
Guide	A guide to help you finding out what guides have been written	[PDF] [HTML]
Install	A guide to the different ways of installing the Java CoG Kit	[PDF] [HTML]
Commands	A guide to the command line tools of the Java CoG Kit	[PDF] [HTML]
Workflow	A guide to the Gridant/Karajan Workflow	[PDF] [HTML]
JavaDoc	The Java API documentation to the Java CoG Kit	[HTML]
Coding	A guide to the Coding rules for the Java CoG Kit	[PDF] [HTML]

C. Java CoG Kit Guides Under Construction

More guides are under development. The following guides are not yet completed, but are listed here to help us improving these guides. Please, explore them and send us e-mail about improvement suggestions. If you like to contribute a guide yourself, please contact gregor@mcs.anl.gov.

Short Title	Description	Format
MPI	A preliminary guide to execute MPI programs on the TeraGrid and alike	[PDF] [HTML]

D. Available Downloads

First time users of the Java CoG Kit should read the "Guide to Installing the Java CoG Kit" [PDF] [HTML]. We hope that you will find this guide useful to decide which bundles you need. For the more experienced user, we provide the following table.

Binary Distributions

- Complete (all providers) [tar.gz][zip]
  Installation Guide [HTML] [PDF]
- Separate providers
  Installation Guide [HTML] [PDF]
  - Main package (include GT2 providers) [tar.gz][zip]
  - Common GT 3.x.x package [tar.gz][zip] (required for all GT 3.x.x providers)
  - GT 3.0.2 provider [tar.gz][zip]
  - GT 3.2.0 provider [tar.gz][zip]
  - GT 3.2.1 provider [tar.gz][zip]
  - SSH provider [tar.gz][zip]
  - WebDAV provider [tar.gz][zip]
  - Local provider [tar.gz][zip]

Source Distributions

- Complete source distribution
  Installation Guide [HTML] [PDF]
  Source Distribution [tar.gz][zip]

API Documentation

- Complete API [tar.gz][zip]

CVS

- Please consult the Installation Guide [HTML] [PDF]

E. Availability of the Document

The newest version of this document can be downloaded by the developers from

1. cvs -d:pserver:anonymous@cvs.cogkit.org:/cvs/cogkit checkout manual/guide

F. Bugs

This guide is constantly improved and your input is highly appreciated. Please report suggestion, errors, changes, and new sections or chapters through our Bugzilla system at http://www-unix.globus.org/cog/contact/bugs/

G. Administrative Contact

The Java CoG Kit project has been initiated and is managed by Gregor von Laszewski. To contact him, please use the information below.

Gregor von Laszewski
Argonne National Laboratory
Mathematics and Computer Science Division
9700 South Cass Avenue
Argonne, IL 60439
Phone:(630) 252 0472
Fax: (630) 252 1997
gregor@mcs.anl.gov