

JAVA COG KIT KARAJAN/GRIDANT WORKFLOW GUIDE

Gregor von Laszewski and Mike Hategan

Software Version: 4.1.2

Manual version: 02/18/05

Url: <http://www.cogkit.org/release/4.1.2/manual/workflow.pdf>

Url: <http://www.cogkit.org/release/4.1.2/manual/workflow/workflow.html>

Last update: July 12, 2005

CONTENTS

1	About this Document	2
2	Registration	2
3	The Karajan Language	3
4	Parameters and Return Values	6
5	Source Files	12
6	Libraries	12
	Appendix	39
A	Availability of the Document	39
B	Bugs	39
C	Administrative Contact	39

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The material presented in this document can not be published, mirrored, electronically or otherwise reproduced without prior written consent.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 planned, 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

◆ Gregor: explain the name

Karajan is a parallel scripting language. It uses a declarative concurrency approach to parallel programming. An interpreter is also provided, featuring the ability to checkpoint the state of the execution.

◆ Gregor

3. THE KARAJAN LANGUAGE

Karajan supports two syntax modes: a native syntax and XML. There is no difference on the semantic level between the two forms.

3.1. The Karajan Syntax

The following conventions are used:

(xy) are used to group x and y

$[x]$ indicates that x is optional

$x+$ denotes at least one occurrence of x

x^* denotes zero or more occurrences of x

$x|y$ means either x or y

' x ' is to be interpreted as the literal x

E represents the empty production

3.1.1. Elements

The Karajan semantics revolve around the notion of *elements*. An element is relatively similar to a function in that it has a name, can accept arguments, and may return values. The general syntax for an element is:

```
element ::= identifier '(' [arguments] ')'
```

Example:

```
print(1)
false()
```

3.1.2. Identifiers

An identifier can consist of alpha-numeric characters and certain symbols, but no whitespace. Symbols that cannot be used in an identifier are symbols that have other syntactic functions, such as brackets (all of them), commas, double quotes, and the equals sign. Identifiers are case insensitive.

```
identifier ::= (Letter | Digit | '!' | '@' | '#' | '$' | '%' | '^' |
               | '&' | '*' | '_' | '-' | '+' | ':' | ';' | '"' | '<' |
               | '>' | '.' | '?' | '/' | '\\ | '|' | "'" | '~')+
```

Example:

```
var, i, v123, -, +!@, a$, big_list, small-map, grid:task, file.list
```

A number cannot be an identifier. However, certain characters that are used to represent a number can be identifiers (such as '+', '-', '.')

Valid identifiers are also '==', '<=', '>=' (even though the general rule for identifiers is broken in their case).

3.1.3. Arguments

The arguments can either be other elements or values, such as numeric values or strings. Elements can be separated by commas or the new line character (or both):

```
arguments ::= argument [separator arguments] | E
separator ::= ',' | Newline
```

Example:

```
list(true(), false())

list(
  true()
  false()
)

list(true(),
      false())
```

Arguments come in two flavors. Named arguments and unnamed arguments:

```
argument ::= named_argument | unnamed_argument
```

3.1.4. Named Arguments

Named arguments provide a way of explicitly binding arguments to formal parameters:

```
named_argument ::= identifier '=' unnamed_argument
```

Example:

```
print(true(), nl = false())
```

3.1.5. Unnamed Arguments

Unnamed arguments can be either immediate values or elements. Immediate values can be numeric literals, string literals, variables, or quoted lists:

```
unnamed_argument ::= numeric_literal | string_literal | variable | quoted_list |
                  | element

numeric_literal ::= ['+'|-'] digit+ ['. ' digit+]

digit ::= '0'... '9'

string_literal ::= ''' any_characters_but_double_quotes '''

variable ::= identifier
```

Example:

```
list(1, 2.3, -4.56,
      +7.890, "A string", list("Another string value in a nested list", "*2"))
```

3.1.6. Quoted Lists

A quoted list is a special element that produces a list of identifiers. What is specific about a quoted list is that if its arguments are variables, the variables will not be evaluated. Instead their identifiers will be added to the list. Quoted lists are convenience syntax for expressing a list of formal arguments:

```
quoted_list ::= '[' arguments ']
```

However, quoted lists are not limited to expressing list of arguments. They can also be used to express lists of values. The only thing to remember is that variable evaluation will not take place for immediate arguments of a quoted list.

Example:

```
list("A quoted list follows", [a, b, c])
```

3.1.7. Programs

A Karajan program is a list of arguments:

```
program ::= arguments
```

There exists an implicit root element that sits at the top of the element tree, and implements certain system functions.

3.1.8. Comments

And finally, Karajan uses C-style comments. Single-line comments begin with two forward slashes and end at the following new line character, while multi-line comments are delimited by `/*` and `*/`:

```
//This is a comment

print("This is not a comment")

/*This is
also a
comment
*/
```

3.2. The XML Syntax

Karajan also supports XML as its syntax. In the XML syntax, each XML element corresponds to a Karajan element. Arguments can be expressed either through XML attributes or nested elements.

3.2.1. Particularities of Using XML

One of the particular aspects of using XML with Karajan is that when using XML attributes for arguments, it is impossible to make a syntactic distinction between a numeric value and its string representation. In general, Karajan will try to use the context to figure out which one is desired, but there are instances when it is impossible to do so. Therefore, when using the XML syntax, the following elements can be used for the purpose of differentiating between numeric and string values: `number` and `string`

```
<list>
  <number>1</number>
  <string>1</string>
</list>
```

The equivalent Karajan construct would be:

```
list(1, "1")
```

Karajan can load and interpret arbitrary XML files, provided that definitions exist for the XML elements present in the file, but XML mixed content is not handled properly. The unfortunate aspect is that it is impossible to handle XML mixed content in a generic way. For example, it cannot be known whether whitespace between two XML elements is to be interpreted as content or not, without knowledge of the implementation of an element. Since Karajan is a dynamic language, the implementation of an element is not known statically, at the time the parsing takes place. Therefore, the following rule was adopted: An element will consider textual content content if and only if no nested elements exist. If nested elements exist, textual content will be ignored.

If processed, textual content will be mapped as a string argument. A consequence of the above rule is that textual content and multiple arguments are mutually exclusive.

Lastly, a well-formed XML document must always have a root element. While in the native Karajan syntax, the root element is implicit, in XML the `project` or `karajan` elements can be used as root elements.

4. PARAMETERS AND RETURN VALUES

An element can accept any number of arguments and can generate any number of return values, and that includes an infinite number of arguments and/or return values (at least in theory).

4.1. Parameters and Arguments

Arguments are divided into two major types: single value arguments and channels. As their name implies, single value arguments can have only one value. By contrast, channels can be used for any number of values.

4.1.1. Single Value Arguments

Single value arguments can be specified using the named argument form. For example, the `print` element has a `message` argument. Thus passing a string as the `message` argument to `print` element can be done in the following way:

```
print(message = "Some string")
```

Or in XML:

```
<print message = "Some string" />

<!-- OR -->

<print>
  <argument name = "message" value = "Some string" />
</print>
```

Single value arguments can be further divided into mandatory and optional arguments.

4.1.2. Channels

Channels can be used to pass multiple arguments to an element. Each channel has a name, except for the default channel. The default channel is similar to the notion of variable arguments in C. Passing arguments on the default channel is done implicitly when arguments are not passed as single value arguments:

```
list(1, "value")
```

```
<list>  
  <number>1</number>  
  <string>value</string>  
</list>
```

In the above case, 1 and "value" are both passed to the `list` element on the default channel. Elements define whether they do receive arguments on a specific channel or not. One possibly interesting aspect is that an element that does not process arguments on a channel, will automatically return all values received on that channel. It is therefore possible to use named channels to return values to elements other than the immediate parent. Assuming that `foo` is an element that does not take any arguments on any channels, the following will produce the same result:

```
list(1, 2, 3)
```

```
list(foo(1, 2, 3))
```

```
<list>  
  <number>1</number>  
  <number>2</number>  
  <number>3</number>  
</list>  
  
<list>  
  <foo>  
    <number>1</number>  
    <number>2</number>  
    <number>3</number>  
  </foo>  
</list>
```

Since `foo` does not process any arguments, all the arguments it receives on the default channel will be returned to the parent element.

4.1.3. Argument Mapping

It is not always convenient to use the named argument form to pass arguments to an element. Elements in Karajan will automatically map arguments received on the default channel to single value arguments. The mapping is done dynamically, in the order arguments are received. Suppose there is an element `foo` that takes three arguments, namely 'one', 'two' and 'three'. The following would then be equivalent:

```
foo(one = 1, two = 2, three = 3)
```

```
foo(one = 1, two = 2, 3)
```

```
foo(one = 1, 2, 3)
```

```
foo(1, 2, 3)
```

```
foo(1, 2, three = 3)
```

```
...
```

```

<foo one = "1" two = "2" three = "3" />

<foo one = "1" two = "2">
  <number>3</number>
</foo>

<foo one = "1">
  <number>2</number>
  <number>3</number>
</foo>

<foo>
  <number>1</number>
  <number>2</number>
  <number>3</number>
</foo>

<foo>
  <number>1</number>
  <number>2</number>
  <argument name="three" value="3" />
</foo>

...

```

4.1.4. Optional Arguments

The unfortunate side-effect of using automatic mapping of default channel arguments to single value arguments is that elements that would accept both single value arguments and arguments on the default channel (variable arguments) cannot avoid mapping of variable arguments to certain single value arguments unless different semantics are introduced: optional arguments. Optional arguments do not need to be specified. However, if specified, the named form must always be used. An example is the `print` element, which has an optional argument named `nl`. It can be set to `false` to indicate that no new-line character should be appended at the end of the `message` argument:

```
print(message = "Message", nl = false())
```

or

```
print("Message", nl = false())
```

The following however, is not valid:

```
print("Message", false())
```

4.2. Return values

Return values are a mirror image of the arguments concept. Whatever can be accepted as an argument by an element can also be returned by another. Thus, it is possible to define a single element that returns all arguments to any given element. The following example defines an element that returns both a message and the named form of the `nl` argument, suitable for the `print` element:

```

element(foo, []
  "Message", nl = false()
)

```

```
print(foo())
```

```
<element name="foo" arguments="">  
  <string>Message</string>  
  <argument name="nl">  
    <false/>  
  </argument>  
</element>  
  
<print>  
  <foo/>  
</print>
```

4.3. Argument Evaluation Order

There is no imposed order for evaluating arguments. The order is controlled by each element. Most elements, by default, evaluate their arguments in sequential order. However, it is very easy to override the default order by using elements that use a different execution order. For example, the `parallel` element evaluates all of its arguments in parallel, returning all the resulting values. Evaluating the arguments to an element in parallel then becomes as easy as surrounding them with a `parallel` element:

```
list(  
  parallel(  
    "Value1"  
    "Value2"  
  )  
)
```

```
<list>  
  <parallel>  
    <string>Value1</string>  
    <string>Value2</string>  
  </parallel>  
</list>
```

Furthermore, the way in which an element processes the arguments is also left to each element. For example, an element can choose to start executing after the evaluation of all arguments has been completed, or process arguments as they arrive. In other words, and in the most general case, arguments are both generated and processed asynchronously.

A concrete example is the `print` element, which simply returns the *message* argument on the *stdout* channel. When Karajan starts execution, an implicit root element is created that receives arguments on the *stdout* channel, and prints them to the console. Since the processing is done asynchronously, the appearance of `print` doing the actual work when executed is achieved. The advantage of such a mechanism is that, provided that an element does not produce any side-effects, its execution becomes equivalent to the totality of values returned (both single values, and channels).

4.4. Variables and Scope

We will not insult the reader's intelligence by explaining what variables are. There is no explicit declaration of variables in Karajan. A variable is defined when it is assigned the first time.

The scope of a variable extends to the element that it was defined in, and is pseudo-lexical. By pseudo-lexical it is meant that internally, the scoping is dynamic, but provisions are made

to make it impossible to access variables outside the lexical scope. Therefore, Karajan does not support closures.

On a lexical level, it is possible to read the value of a variable defined in a parent element, but setting the value of the same variable will create a new scope. In other words, Karajan uses deep access and shallow binding. The following example should make things clearer:

```
...
set(v, 1) //v$ is $1$ on stack frame $n$

list( //A new stack frame is created: $n+1$

  v //v$ refers to the variable on frame $n$

  set(v, 2) //A new binding is made for $v$ on frame $n+1$
           //the new binding shadows the one from frame $n$

  v //v$ now refers to the binding on frame $n+1$

) //The returned list contains the values $1$ and $2$

print(v) //v$ refers again to the binding on frame $n$
         //Therefore the printed value will be $1$
...
```

```
...
<set name="v" value="1"/> <!-- v$ is $1$ on stack frame $n$ -->

<list>   <!-- A new stack frame is created: $n+1$ -->

  <variable>v</variable> <!-- v$ refers to the variable on frame $n$ -->

  <set name="v" value="2"/> <!-- A new binding is made for $v$ on frame $n+1$ -->
                          <!-- the new binding shadows the one from frame $n$ -->

  <variable>v</variable> <!-- v$ now refers to the binding on frame $n+1$ -->

</list>  <!-- The returned list contains the values $1$ and $2$ -->

<print>
  <variable>v</variable> <!-- v$ refers again to the binding on frame $n$ -->
</print> <!-- Therefore the printed value will be $1$ -->
...
```

In the above example, it would be impossible for the definition of `list` to access variable `v`, since the body of the definition of `list` does not fall within the lexical scope of the definition of `v`.

The reason for this kind of scoping is to reduce the ambiguity that could be introduced by not knowing the order in which child elements are executed. Please note that such ambiguity is not completely eliminated. The order in which the arguments to `list` are evaluated does matter, and can change the resulting list. However, the scope of the ambiguity is the same as the scope of the ambiguity in the order of evaluation of the elements. If `set`, `list`, and `print` are evaluated in sequence, no change in the way `list` evaluates its arguments can change the outcome of the execution of `print(v)`.¹

As mentioned before, closures are not supported. The following example will not work, because at the time `inner` is evaluated, `foo` would not be defined in the scope of `inner`.

¹I don't like this

```

    element(outer, [foo]
      element(inner, []
        print(foo)
      )
    )
  )
executeElement(outer("Foo"))

```

In fact, a very simple rule is that no variable can be accessed inside the definition of an element, unless it refers to an argument or a global variable.

4.4.1. Global Variables

Global variables are provided for conveniently defining settings that have a global scope. Please note that in the future, global variables will be single-assignment, this approaching more the notion of constants.

```

global(foo, "Foo")

element(boo, []
  print(foo)
)

boo()

```

4.5. Variable Expansion

Karajan offers convenient variable expansion constructs. All pairs of curly brackets inside strings are replaced by the value of the variable with the name of the identifier inside the brackets. If no such variable exists, the element trying to access the string will fail. If the '{' literal is needed inside a string, it must be used twice. There is no need to escape the closing curly bracket, since it cannot be part of an identifier. If a closing bracket is part of a variable expansion expression, it will mark its end. If not, it will be interpreted as the closing curly bracket literal:

```

set(a, 1)

print("A is {a}")

print("An opening curly bracket: {{")

print("A closing curly bracket: }")

```

```

<set name="a" value="1"/>

<print message="A is {a}"/>

<print message="An opening curly bracket: {{"/>

<print message="A closing curly bracket: }"/>

```

4.6. Futures

Futures are a mechanism of binding a variable to the results of a future computation. Until the value of the computation to which the future is bound to, the future exists in an unbound

state. Any attempt to use the value of an unbound future will cause the execution of the thread that tried to access the future to block until the future becomes bound.

In Karajan there are two types of futures:

single value futures are used to hold a single value of a future computation. They are defined using the `future` element.

future iterators can be used to hold multiple values. However, not all the values need to be generated before the future iterator can be used. Iterating over a future iterator will cause the iteration to use as many values as are available, then block waiting for more values to be added to the future iterator. Future iterators are defined using `futureIterator`

5. SOURCE FILES

As mentioned in Section ??, Karajan understands two syntaxes: the native Karajan syntax and the XML syntax. The distinction between them is made using the file extension. A file with the “.k” extension will be parsed using the native parser, while a file with the “.xml” syntax will be parsed using an XML parser.²

5.1. Libraries

Libraries are collections of elements grouped by the functionality they provide. A library is defined in a source file. Its functionality can be reused in other source files by using the `include` element:

```
include("sys.k")
```

It is possible to include XML libraries from native Karajan files. It is also possible to include native Karajan libraries from XML Karajan files. Consequently, the following are valid:

```
include("task.xml")
```

```
<include file="task.k"/>
```

5.2. Namespaces

Namespaces provide a way of distinguishing between elements with conflicting names in different libraries. Suppose a library “a” defines an element named `foo`, and a library “b” also defines an element named `foo`. Also, suppose that both libraries are included in a certain file. Namespaces make it possible to access both instances of the `foo` definition, without ambiguity, by prefixing the name with the namespace prefix in which the element was defined: `a:foo` and `b:foo`. Any reference to `foo` without a prefix will result in an error. Nonetheless, if only one of the libraries is used, the use of `foo` without a prefix will be allowed. Namespaces are defined using `namespace`.

6. LIBRARIES

This section lists the current libraries available in Karajan.

The following conventions are used:

Normal arguments are listed in *italics*: *argument*

Optional arguments are listed in *italics* and with an asterisk in front: **optional*

The default channel is symbolized by `...`

For channels the ***bold-italics*** font variant is used: *channel*

²The current implementation will first translate the native syntax to XML, than parse the resulting XML file

6.1. The Kernel

Files: *none*

Namespace prefix: *kernel*

The Karajan kernel contains a minimum set of elements that are required in order to get the rest of the system running. All kernel elements are automatically available in any program.

`project(stdout)`

Aliases: *karajan*

The root element of a Karajan program. Accepts arguments on the *stdout* channel and prints them immediately on the console.

`include(file)`

Parses and includes and executes a file inside the current file at the location where `include` is present. An included file will not be included a second time in any sub-scope of the parent of the current `include`:

```
foo(  
    include("file.k")  
  
    moo(  
        include("file.k") //will not be included again  
    )  
  
    include("file.k") //will not do anything either  
  
) //scope of foo() ends  
  
include("file.k") //will be included
```

`namespace(prefix)`

Allows the specification of a namespace prefix. Any elements defined in the scope of `namespace` will automatically have the prefix indicated by the *prefix* argument, unless another `namespace` is nested.

`elementdef(type, classname)`

Used by the current implementation to map element names to Java implementation classes.

`named(value, *name)`

Used internally by the named argument form, but can be used by the user equally well. The following are equivalent:

```
print("Test", nl = false())  
print("Test", kernel:named(name = nl, false()))
```

However, the fact that the **name* argument is optional makes it impossible to completely avoid the named form.

If the **name* argument is not present, it simply returns the value of the *value* argument on the default channel.

`number(value)`

Can be used with the XML syntax to represent a number. There is no distinction between integral numbers and floating point numbers in Karajan.

`string(value)`

Can be used with the XML syntax to represent a string value.

`variable(name)`

Used to represent the value of a variable.

Example:

```
<set name="n" value="5" />  
  
<list>  
  <number>10</number>  
  <string>10</string>  
  <variable>n</variable>  
</list>
```

will return the list [10, "10", 5].

`quotedlist(...)`

Used internally to represent a quoted list. The following two lines of code will produce the same result:

```
[a, b, c]  
quotedlist(a, b, c)
```

6.2. The System Library

Files: *sys.k*, *sys.xml*

Namespace prefix: `sys`

The system library contains general purpose elements that help implement the most common tasks in Karajan.

6.2.1. Flow Control Elements

`sequential()`

Executes all arguments in sequence.

`parallel()`

Executes all arguments in parallel.

`unsynchronized()`

Asynchronously executes all arguments in sequence. Does not return any value. If return values from asynchronous computations is required, use either [future](#) or [futureIterator](#)

`choice()`

Executes arguments in succession. Terminates when one of the arguments completes successfully. If an argument fails, the next argument will have access to the following variables:

element Contains a reference to the element that caused the initial failure.

error A textual message detailing the error that occurred

trace A textual representation of the Karajan stack trace.

exception Available if a Java exception caused the failure.

If no argument completes successfully **choice** will fail with the last failure encountered.

Choice exhibits a transactional behavior when it comes to return values. All single values and all channels are buffered until one argument completes successfully. If that happens then **choice** returns the buffered values. If an argument fails, all the buffered values produced by that argument are discarded.

`parallelChoice()`

Can be used to race a number of arguments. **ParallelChoice** will execute all its arguments in parallel, buffer their return values, and wait for the first one that completes. It will then return all the values that the winner generated. If an any argument fails before any other argument completes, then **parallelChoice** will fail.

`for(name, in)`

Can be used to iterate sequentially across a range of values. The *name* argument is an identifier that indicates the name of the variable that will be set to the successive values of the *in* argument, which Karajan will try to convert to an iterator before beginning the iteration process.

After evaluating the *name* and *in*, `for` will proceed and evaluate the rest of the arguments repeatedly, while setting the variable indicated by the *name* argument to each value produced by the iterator (*in*).

Example:

```
==(
  list(
    for(i, range(1, 5), i)
  )
  list(1, 2, 3, 4, 5)
)
```

will return *true*

`parallelFor(name, in)`

Will behave in a similar way to `for` with the exception that iterations will occur in parallel. Each iteration will occur in a separate scope. Therefore variables set in one of the iterations will not be visible in the others. Each scope will have the variable indicated by the *name* argument set to one of the values obtained from the *in* argument.

`while(condition)`

Will repeatedly execute its arguments in sequence until a value of *false* is received on the *condition* channel. A convenience element that returns a boolean argument on the *condition* channel is `condition` (or `?`). `While` will check for a condition every time an argument completes. It is therefore possible to exit the loop after the termination of any of the arguments.

Example:

```
list(
  while(
    1, 2, 3, ?(false())
  )
)

list(
  while(
    1, ?(false()), 2, 3
  )
)

list(
  while(
    ?(false()), 1, 2, 3
  )
)

list(
  while(
    sequential(
      ?(false())
      0
    )
  )
)
```

```

        /* zero will make it to the list
        * because while will only do the check
        * after sequential() completes
        */
    )
    1, 2, 3
)
)
)

```

will return the following lists:

[1, 2, 3]

[1]

[]

[0]

Or, in XML:

```

<list>
  <while>
    <number>1</number>
    <number>2</number>
    <number>3</number>
    <condition>
      <false/>
    </condition>
  </while>
</list>

<list>
  <while>
    <number>1</number>
    <condition>
      <false/>
    </condition>
    <number>2</number>
    <number>3</number>
  </while>
</list>

<list>
  <while>
    <condition>
      <false/>
    </condition>
    <number>1</number>
    <number>2</number>
    <number>3</number>
  </while>
</list>

<list>
  <while>
    <sequential>
      <condition>
        <false/>
      </condition>
      <number>0</number>
    </sequential>
  </while>
</list>

```

```

        <!-- zero will make it to the list
            because while will only do the check
            after sequential() completes -->
    </sequential>
    <number>1</number>
    <number>2</number>
    <number>3</number>
</while>
</list>

```

break()

Can be used to break out of a **while** loop. By contrast with using the *condition* channel, **break** will immediately exit the loop, not matter how deep the nesting level.

continue()

Can be used to skip the evaluation of remaining arguments in a **while** loop and jump to the next iteration.

if()

Executes its elements using the following scheme:

1. start with $k = 0$
2. evaluate argument $2 * k$;
3. if argument $2 * k$ is the last argument, then return its return values and complete
4. if no value is returned by argument $2 * k$, fail
5. if value returned by argument $2 * k$ is *true* then evaluates argument $2 * k + 1$ returning its return values, and completes
6. if value returned by argument $2 * k$ is *false* then continue with $k = k + 1$

then()

Aliases: *else*

Then and **else** are the same as **sequential**, but can be used to make constructs using **if** more intuitive:

```

if(
  ==(1, a)
  then(print("a is 1"))
  ==(2, a)
  then(print("a is 2"))
  else(print("a is not 1 nor 2"))
)

```

```

<if>
  <equals>
    <number>1</number>
    <variable>a</variable>
  </equals>
  <then>
    <print message = "a is 1" />
  </then>

  <equals>
    <number>2</number>
    <variable>a</variable>
  </equals>
  <then>
    <print message = "a is 2" />
  </then>

  <else>
    <print message = "a is not 1 nor 2" />
  </else>
</if>

```

6.2.2. Elements Dealing with Variables and Arguments

`set(names, ...)`

Sets a variable or more to a value (or more). `Set` tries to interpret the first argument as an identifier or a list of identifiers. If the first argument is an identifier, it is treated as being a list with one identifier. `Set` expects the number of arguments on the default channel to be the same as the number of identifiers. It is important that a quoted list be used to specify the list of identifiers in order to avoid evaluating the variables that the identifiers represent:

```

set(a, 1)

set([a], 1)

set([a, b, c], 1, 2, 3)

```

Differences in XML:

The XML variant of the `set` element uses a slightly different set of arguments. If a single variable is assigned, the *name* argument can be used. If multiple variables are assigned, the *names* argument must be used. As opposed to the native syntax, the *names* argument can be a string of comma separated identifiers which will be tokenized by `set`

```

<set name="a" value="1" />

<set names="a, b, c" >
  <number>1</number>
  <number>2</number>
  <number>3</number>
</set>

```

`default(name, value)`

Assigns a value to a variable if no binding of that variable can be accessed within the current scope. If an accessible variable with the indicated name is already defined, then the assignment does not take place:

```
default(a, 1) //a is assigned the value 1

set(b, 2)

default(b, 3) //b is not assigned the value of 3 because
              //is already accessible within the current scope
```

`global(name, value)`

Sets a global variable. A global variable is a variable that has a global scope, and thus can be accessed from anywhere within the program. While the use of global variables is discouraged, it can prove useful to create constant-like definitions.

Differences in XML:

The XML variant of `global` uses the same arguments as the XML variant of the `set` element.

`...()`

Aliases: *vargs*

Can be used inside an `element` definition to return all arguments received on the default channel.

Differences in XML:

In the XML syntax the `vargs` alias must be used, because “...” is not a valid XML element name.

`channel:to(name, ...)`

Returns all arguments received on the default channel on the specified channel.

`channel:from(name, jvariesj)`

Returns all arguments received on the specified channel on the default channel.

`isDefined(name)`

Determines whether a variable is accessible within the current scope. Returns *true* if it is; *false* otherwise.

`quoted(name)`

Returns an identifier without evaluating the variable it might point to.

`discard(...)`

Sometimes only the side-effect of an element is needed, while ignoring the return values of the element. `Discard` will evaluate its arguments, but avoid returning anything on the default channel.

`future(...)`

Evaluates the arguments asynchronously. Returns a future representing the first return value generated by the arguments. All other arguments received on the default channel are ignored.

`futureIterator(...)`

Evaluates its arguments asynchronously. Returns a future iterator representing all values received on the default channel. The particular aspect of a future iterator is that it can only be iterated over once. Every time a value is used from a future iterator, that value is removed. If access to the iterator values is needed more than once, a list can safely be created with the values. However, the process of creating the list will force synchronization with the thread that produced the values since the iterator is only closed when that thread completes.

6.2.3. Element Definition Elements

`element(name, arguments)`

Allows the definition of an element. Evaluates the *name* and *arguments* arguments. It expects the *name* argument to be an identifier, and *arguments* to be a list of identifiers. The scope of the definition is the same as the scope of a variable that could be defined instead of the element. The arguments are a list of mandatory arguments. Optional arguments can also be specified using the `optional` element. If the element accepts arguments on the default channel, the `...` identifier can be used in the argument list. Other channels can be specified using the `channel` element. The rest of the arguments are not evaluated when the definition takes place, but will be evaluated whenever the element is invoked.

The following example defines an element `foo`, which takes no arguments, and prints 'foo' on the console:

```
element(foo, []
    print("foo")
)

foo()
```

In the following example, `foo` takes two arguments and prints them both on the screen:

```
element(foo, [one, two]
    print(one)
    print(two)
)

foo(1, 2)
```

Arguments on the default channel can be accessed using the `...` identifier:

```
element(foo, [one, ...]
  print(one)
  for(i, ...
    print(i)
  )
)
foo("one", 1, 2, 3, 4)
```

Other channels can be used in a similar way:

```
element(foo, [one, ..., channel(channelOne)]
  print(one)
  for(i, ..., print(i))
  for(i, channelOne, print(i))
)
foo("one", 1, 2, 3, 4, channel:to(channelOne, 5, 6, 7, 8))
```

Optional arguments can be assigned a default value using `default`:

```
element(foo, [one, optional(two)]
  default(two, 2)
  print(one)
  print(two)
)
foo("one")
foo("one", two = "two")
```

`Element` can also be used to define anonymous elements. If the first argument evaluates to a list of identifiers instead of an identifier, `element` considers that an anonymous element was instead desired, defines the element, and returns the definition, which can later be used through `executeElement`:

```
set(foo,
  element([
    print("Foo")
  ])
)
executeElement(foo)
```

Differences in XML:

The arguments list is a string of comma separated identifiers.

The XML version of `element` uses a different set of arguments. If an element accepts arguments on the default channel, the `args="true"` attribute must be used. The arguments received on the default channel will then be available in the body of the definition through the `args` identifier.

Optional arguments are indicated using the `optargs` attribute. The value must be a comma separated list of identifiers.

In a similar way, the channels are specified using a comma separated list of identifiers and the `channels` attribute.

```
<element name = "foo" arguments = "one"
  args = "true" channels = "channelOne" >
  <print message = "{one}" />
```

```

<for name = "i" in = "{vargs}" >
  <print message = "{i}" />
</for>

<for name = "i" in = "{channelOne}" >
  <print message = "{i}" />
</for>
</element>

<foo one = "one" >
  <number>1</number>
  <number>2</number>
  <number>3</number>
  <number>4</number>

  <channel:to name="channelOne" >
    <number>5</number>
    <number>6</number>
    <number>7</number>
    <number>8</number>
  </channel:to>
</foo>

```

```

<element name = "foo" arguments = "one" optargs = "two" >
  <default name = "two" value = "2" />
  <print message = "{one}" />
  <print message = "{two}" />
</element>

<foo one = "one" />
<foo one = "one" two = "two" />

```

`parallelElement(name, arguments)`

Like `element`, `parallelElement` also defines an element. However, elements defined using `parallelElement` will evaluate their arguments in parallel with their bodies. All single value arguments will automatically be futures, and all channels will automatically be future iterators. `ParallelElement` can be used to define elements that process their arguments asynchronously.

```

parallelElement(consumer, [...]
  for(i, ..., print("Received {i}"))
)

element(producer, []
  for(i, range(0, 100)
    i
    print("Sent {i}")
    wait(delay = 100)
  )
)

consumer(producer())

```

Differences in XML:

The differences for the XML syntax from `element` also apply to `parallelElement`

```
<parallelElement name = "consumer" vars = "true">
  <for name = "i" in = "{vars}">
    <print message = "Received {i}" />
  </for>
</parallelElement>

<element name = "producer">
  <for name = "i">
    <range from = "0" to = "100" />

    <variable>i</variable>
    <print message = "Sent {i}" />
    <wait delay = "100" />
  </for>
</element>

<consumer>
  <producer />
</consumer>
```

`channel(...)`

Used to specify channel arguments for an element. Quotes all arguments, such that identifiers are not evaluated. Returns values that can be interpreted by [element](#) and [parallelElement](#) as representing channels.

`optional(...)`

Allows the specification of optional arguments for element definitions. Quotes all arguments and returns values that can be interpreted by [element](#) and [parallelElement](#) as representing optional arguments.

6.2.4. List Manipulation Elements

`list(*items, ...)`

Constructs a list from values received on the default channel.

Alternatively the `*items` argument can be used to specify a string with comma separated list of items. This may be particularly convenient with the XML syntax.

`list:append(list, *items, ...)`

Appends all values received on the default channel to the list indicated by the `list` argument. Does not return anything.

Alternatively, the `*items` argument could be used with a string of comma separated items.

`list:prepend(list, ...)`

Works like `list:append` with the exception that values are added to the beginning of the list. The order of the values in the list will be the reverse of the order in which they are received by `list:prepend`:

```
set(l, list(4, 5, 6))  
  
list.prepend(l, 1, 2, 3)  
  
print(l)
```

will print `[3, 2, 1, 4, 5, 6]`

`list:concat(...)`

Concatenates all lists received on the default channel and returns the resulting list.

`list:size(list)`

Returns the size of the list indicated by the `list` argument.

`list:first(list)`

Returns the first element in a list.

`list:last(list)`

Returns the last element in a list.

`list:butFirst(list)`

Returns a list composed of all but the first element in the specified list.

`list:butLast(list)`

Returns a list containing all elements but the last from the specified list.

`list:isEmpty(list)`

Tests whether a list is empty. Returns `true` if the list is empty, and `false` otherwise.

6.2.5. Map Elements

`map(...)`

Returns a map with the entries received on the default channel.

`map:entry(key, value)`

Allows the specification of an entry that can be used with `map` to construct a map.

`map:put(map, ...)`

Adds the entries received on the default channel to the specified map. Existing entries with the same key are replaced. Does not return any value.

`map:delete(map, key)`

Deletes the entry with the specified key from a map. Does not return a value.

`map:get(map, key)`

Returns the value corresponding to the specified key from a map.

`map:size(map)`

Returns the size of the map (the number of entries in the map).

`map:contains(map, key)`

Tests whether the map contains an entry with the specified key.

6.2.6. Logic Elements

Logic elements do not at this time use shortcut evaluation. They always evaluate all their arguments.

`and(...)`

Aliases: `&`

Returns the boolean *and* value of the arguments received on the default channel.

`or(...)`

Aliases: /

Returns the boolean *or* value of the arguments received on the default channel.

`not(value)`

Aliases: !

Returns the boolean negation of the value in the *value* argument.

`equals(value1, value2)`

Aliases: ==

Tests for the equality of two values. Makes a deep comparison of the arguments.

`true()`

Returns the *true* boolean value. There is no syntactic way of specifying a boolean value of *true* in Karajan.

`false()`

Returns the *false* boolean value.

6.2.7. Numeric Elements

`sum(...)`

Aliases: +

Returns the sum of all the values received on the default channel.

`product(...)`

Aliases: *

Returns the product of all the values received on the default channel.

`subtraction(from, value)`

Aliases: -

Returns the difference between the value specified by the *from* argument and the value indicated by the *value* argument.

`quotient(divisor, dividend)`

Aliases: /

Divides *divisor* by *dividend* and returns the resulting value.

`remainder(divisor, dividend)`

Aliases: %

Returns the remainder of the division of *divisor* and *dividend*

`square(value)`

Returns the square of a number.

`sqrt(value)`

Returns the square root of a number.

`equalsNumeric(value1, value2)`

Makes a numeric comparison of the values specified by the two arguments. A numeric comparison will try to convert string values to numbers before the comparison is performed. Like `equals`, `equalsNumeric` performs a deep comparison.

```
equalsNumeric(1, "1") //true
equalsNumeric("2", "2.0") //true
equals("2", 2) //false
equalsNumeric([1, 2, "3"], ["1", "2", 3]) //true
```

```
<equalsNumeric value1 = "1">
  <number>1</number>
</equalsNumeric> <!-- true -->

<equalsNumeric value1 = "2" value2 = "2.0" /> <!-- true -->

<equals value1 = "2">
  <number>2</number>
</equals> <!-- false -->

<equalsNumeric>
  <list items = "1, 2, 3" />
  <list>
    <number>1</number>
    <number>2</number>
    <string>3</string>
  </list>
</equalsNumeric> <!-- true -->
```

`greaterThan(value1, value2)`

Aliases: >

Returns *true* if *value1* is strictly larger than *value2*

`lessThan(value1, value2)`

Aliases: `<`

Returns *true* if *value1* is strictly less than *value2*

`greaterOrEqual(value1, value2)`

Aliases: `>=`

Returns *true* if *value1* is larger than or equal to *value2*

`lessOrEqual(value1, value2)`

Aliases: `<=`

Returns *true* if *value2* is less than or equal to *value2*

6.2.8. Error Handling Elements

`ignoreErrors(*match)`

Executes its arguments returning any values as they are received. If any of the arguments fails, the failure will be matched against the regular expression in **match* if present (otherwise it will be treated as `.*`). If the match is successful, then the error will be ignored and the next argument will be executed. If the match fails, the error will be propagated.

`restartOnError(times)`

Evaluates its arguments in sequence. If a failure occurs, all arguments will be re-evaluated for a maximum of *times* times.

`generateError(error, *exception)`

Allows the generation of an error. The message of the error is taken from *error*. **Exception* is used in the current implementation to pass a Java exception to be attached to the error.

`onError(match)`

`OnError` allows the definition of a custom error handler. Handlers defined with `onError` are scoped to anything executed within the scope of the parent of `onError`. Multiple handlers can be defined within the same scope.

Whenever an error occurs within the scope of an error handler, the error will be matched against error handlers starting with the inner-most handlers and ending with the outer-most handlers. If a handler matches, it is invoked. When a handler is invoked it will evaluate all its arguments except for *match* in sequence. The execution takes place in the context of the failing element. The following variables are defined automatically to be used by the body of the error handler:

element The element that caused the error. If the error is corrected by the handler, the execution of the element can be re-started using `executeElement`.

error The message of the error that occurred.

trace A textual representation of the Karajan stack trace.

exception In the current implementation exception can either contain a Java exception or the message “No exception available”.

Error handlers are not re-entrant. If an unhandled error occurs within the body of the handler, the handler will fail immediately.

6.2.9. Miscellaneous Elements

```
print(message, *nl)
```

Returns the value in the *message* argument on the *stdout* channel. If the **nl* argument is not present or set to *true*, it appends a new line character to the *message* argument before returning it. The `project` (root element) automatically prints all argument received on the *stdout* channel on the console.

```
echo(message, *nl, *stream)
```

Immediately prints the value in the *message* argument to the console. If the **nl* argument is not present or set to *true*, it also prints a new line character. If the **stream* argument is present, it instead tries to print the *message* on the specified output stream.

The difference between `print` and `echo` is that `print` does not rely on a side-effect to print values. Therefore the evaluation of `print` cannot be distinguished from returning the value generated by `print`.

It is recommended that `print` be used instead of `echo` if possible.

```
checkpoint(*file, *automatic, *interval, *timestamped, *now)
```

Allows the configuration of checkpointing. If the **now* arguments is present and set to *true*, creates a checkpoint of the state at the time of the evaluation of `checkpoint`, and writes it to the file indicated by the **file* argument.

The **automatic* argument, if set to *true*, indicates that automatic checkpoints should be created at the interval specified by the **interval* argument (in seconds). If the **timestamped* argument is also present and set to *true*, the file names in which the checkpoint is saved will have a date and time appended in the YYYYMMDDhhmm form.

```
wait(*delay, *until)
```

When evaluated waits the number of milliseconds specified by the **delay* argument or until the date in the **until* argument before completing.

`file:execute(file)`

Aliases: `executeFile`

Parses and executes a file. The difference between `include` and `file:execute` is that `file:execute` always parses and executes the file when evaluated, unlike `include` which only parses the file once. `File:execute` can therefore be used to execute files which change over time, or to execute different files based on a certain context.

`executeElement(element, args, ...)`

Executes an element optionally passing the single value arguments indicated by the *args* argument, and the ... on the default channel. *Args* must be a `map` in which the keys are argument names and the values are argument values.

If *args* is not present, ... will be mapped to named arguments according to the rules in Section 4.1.3.

`elementList()`

Returns a list containing the arguments to `elementList` but in non-evaluated form. The elements in the resulting list can then be evaluated using `executeElement`

`cache(on)`

Caches all return values of the rest of its arguments based on the value of the *on* argument. Subsequent evaluations of this element in which the *on* argument will have the same value will not re-evaluate the rest of the arguments, but return the cached values instead. The cache is bound to this static instance of the `cache` element. In other words, if another `cache` element exists, it will not use the values cached by this element, irrespective of the value of the *on* argument.

Caching is not guaranteed. It is a mechanism that could help improve performance, but it should not be relied on to guarantee that certain elements are only evaluated once. Also, elements that rely on side-effects to perform their function will not be able to perform those functions if their cached value is used. `echo` will, for example, not do anything if cached. However, `print` will, because it does not rely on a side-effect to print values to the console.

`numberFormat(pattern, value)`

Allows the formatting of a decimal number. The *pattern* argument indicates the pattern to be used for formatting (as used by the `java.text.DecimalFormat` class). The *value* argument holds the decimal value that is to be formatted.

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

`file:contains(file, value)`

Aliases: `contains`

`File:contains` determines whether a file contains a specific sequence of characters. The *file* argument points to the file to be checked, while the *value* argument specifies the value to be searched.

`uid(*prefix, *suffix)`

Returns a string with a unique ID. The **prefix* and **suffix* arguments can be used to specify a prefix and a suffix respectively. In the current implementation, the uniqueness of the returned string is relative to the instance of the interpreter.

`file:read(name)`

Aliases: `readFile`

`File:read` reads the contents of a file, pointed to by the *name* argument. 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 element would not be suitable for manipulation of large files.

`outputStream(type, file)`

Returns an output stream which can be used for writing values to. The *type* argument can be one of “stdout”, “stderr”, or “file”. If the *type* is “file”, the *file* argument must be present and indicate a valid file name.

`closeStream(stream)`

Closes a stream opened with `outputStream`.

`concat(...)`

Concatenates all arguments received on ... and returns the resulting string value.

`split(string, separator)`

Returns a list obtained by splitting *string* in tokens separated by *separator*. The separator will not be part of the tokens.

`matches(string, regexp)`

Returns *true* if *string* matches the regular expression specified by *regexp*, and *false* otherwise.

`filter(*regexp, *invert, ...)`

Filters arguments based on a regular expression, specified by **regexp*. If **invert* is *true*, matches will be inverted (values that do not match are returned).

If only one argument is received on the default channel, and that argument is a list, a list is returned with the values of the argument filtered.

If more than one argument is received on the default channel, each argument will be matched against **regexp*.

6.3. The Java Library

Files: *java*

Namespace prefix: *java*

The Java library allows limited interfacing with Java classes and objects from Karajan.

`new(classname, *types, ...)`

Instantiates a new Java object and returns it. *Classname* represents the fully qualified name of the class. The **types* argument is a list of fully qualified class names used to search for a constructor. The arguments on the default channel are passed to the constructor after performing a conversion based on the **types* argument. The **types* argument is not always necessary, but should be used if Karajan cannot determine the types of the arguments that need to be passed to the constructor.

```
set(x
  java:new("java.lang.Double", "1.0")
  /* The types argument is not necessary since
   * the string argument is automatically mapped
   * to java.lang.String
   */
)

set(j
  java:new("java.lang.Integer", types = ["int"], 1)
  /* The types argument is required since the numeric
   * type used by Karajan cannot be mapped automatically
   * to a specific Java numeric type.
   */
)
```

Primitive Java types are represented by their corresponding keywords: “boolean”, “byte”, “char”, “int”, “long”, “float”, and “double”.

`invokeMethod(method, *static, *classname, *object, *types, ...)`

Invokes a method on a Java object or a static method on a Java class. The invocation is static if **static* is set to *true* or **classname* is present. Otherwise the value of **object* is taken to be a Java object and the invocation is done on a virtual method of the object. *Method* indicates the name of the method. Unless the **types* argument is present, Karajan will try to determine the method signature from A method is searched for in the inheritance hierarchy of the object. If one is found, the method is invoked and, if its return value is not *void* the returned value is returned on the default channel. The format of the *type* argument is identical to the one for [new](#).

`executeMain(class, ...)`

Invokes `static void main(String[] args)` on a class. The qualified class name should be passed in the *class* argument. The arguments on the default channel are converted to strings and passed as the *args* to the *main* method.

`getField(field, *object, *classname)`

Returns the value of a field from an object or class. The field name is passed in *field*. If **object* is present, the value of the instance field of the object is retrieved. If **classname* is present, the class field (static field) of the specified class is retrieved. Arguments **object* and **classname* are mutually exclusive.

`waitForEvent(...)`

`waitForEvent` is a rather obscure and incomplete element. It waits for a Java event such as a button being pressed, or a window being closed. Each argument is a list composed by three elements:

- A return value which will be returned when the associated event occurs
- The type of the event to wait for. Currently the following types are supported:
 - `java.awt.events.ActionEvent` Can be used to wait for a button being pressed (or any other actions that have a `addActionListener(java.awt.events.ActionListener)` method.
 - `java.awt.events.WindowEvent` Can be used to listen for the `windowClosing` notification on a window.
- The source object to be used

When the event occurs, `waitForEvent` completes returning the return value specified as the first item in the list corresponding to the event that the list represents.

`classOf(object)`

Returns the Java class name of the specified object.

`null()`

Karajan has no explicit null value. However, it may be necessary that a null value be used when invoking Java methods. `Null` provides that.

6.4. The Task Library

Files: *task*

Namespace prefix: `task`

The task library interfaces with the Java CoG Kit abstraction classes, allowing the use of services for job submission and file operations. The tasks in this library can function in two modes: scheduled or unscheduled. When scheduled, remote tasks are not executed directly. They are rather passed to a scheduler which can handle issues such as throttling, resource allocation, and task-to-resource mapping.

`scheduler(type, resources, handlers, *properties)`

Defines the a scheduler to be used. The scope of the scheduler is the scope of the execution of the parent of *scope*. The *type* describes the particular type of scheduler that is desired.

Currently only “default” is supported. The resources that can be used by the scheduler are passed in the *resources* argument and can be defined using [resources](#). Each scheduler will also require a list of task handlers, specified using *handlers* with the help of the [handler](#) element. Each type a scheduler may support an optional set of properties. The **properties* argument, if specified, must be a map containing string keys and values.

The “default” scheduler uses a round-robin scheduling policy. However it also performs lookahead matching. This means that if a certain host has reached its maximum allowable number of tasks, it will be skipped. Also, if a suitable host is not found for the next task in the queue, other tasks may be scheduled. The scheduler will use the handlers in the order they were specified in the handlers list, with the first handler having the highest priority.

The default scheduler supports the following properties:

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.

`handler(type, provider)`

A handler specifies a Java CoG Kit Abstraction handler. A handler is used to submit tasks. *Type* indicates the type of handler. They type is string and can have one of the following values: “execution”, “file”, and “file-transfer”.

Execution handlers are used for submitting jobs. File handlers are used for file operations (such as renaming, deleting, and listing of files). File transfer handlers are used only for transferring files. It is possible to transfer files using file handlers, but it is not possible to delete a file using a file transfer handler.

The *provider* argument indicates the provider to be used for the handler. For a list of currently supported providers please see the abstractions guide.

`resources(...)`

Encapsulates a set of hosts which can be specified using [host](#).

`host(name, *cpus, ...)`

Returns a host definition. The *name* argument indicates the host name or IP address. The number of CPUs of the host can be specified using the **cpus* argument. A set of services can also be specified on the default channel.

`service(type, provider, *uri, *project,)`

Returns a service definition. The *type* of the service can be one of “execution”, “file”, or “file-transfer”. *Provider* indicates The Java CoG Kit abstraction provider for the service. For a list of currently supported providers please see the abstractions guide.

The **uri* argument can be used to specify a URI for the service. If missing the host name of the host containing the service will be used.

The *project* argument can be used to automatically bind a queuing system project to the service in order to alleviate the need to do it with the `task:execute` element.

`securityContext(provider, ...)`

Returns a Java CoG Kit abstraction security context. The returned context will be instantiated for the specified provider. The default channel can be used to specify additional properties for

`securityContextProperty(name, value)`

Defines a property for a security context.

`allocateHost(name)`

Allows tasks to be grouped on one host. By default, the scheduler assigns a different host to each task. `AllocateHost` can be used to reserve a host from the scheduler until it completes. The *name* indicates the name of the variable to be set with the allocated host, and is automatically quoted.

```
//Define a scheduler
scheduler(
  ...
)

allocateHost(host1
  task:execute("/bin/date", stdout="date", host=host1)
  task:transfer(srcfile="date", srchost=host1, desthost="localhost")
)
```

Or, in XML:

```
<!-- Define a scheduler -->
<scheduler>
  ...
</scheduler>

<allocateHost name="host1">
  <task:execute executable="/bin/date" stdout="date" host="{host1}"/>
  <task:transfer srcfile="date" srchost="{host1}" desthost="localhost"/>
</allocateHost>
```

The default scheduler uses a late binding mechanism with `allocateHost`. It generates a *virtual host* that is only bound to an actual host when the first task using it is submitted to the scheduler. This removes the limitation on the number of parallel `allocateHost` that can be running, and allows contained jobs to be submitted to the scheduler, which will later handle the throttling issues.

Multiple `allocateHost` can be nested allowing the grouping of tasks on multiple dependent hosts.

`host:hasService(host, type, provider)`

Checks if a host, specified with the `host` element contains a service of the specified *type* and with the specified *provider*. Returns *true* if such a service exists, and *false* otherwise.

`execute(executable, arguments, directory, stdout stderr, stdin, redirect,
provider, host, count, jobtype, maxtime, maxwalltime, maxcputime, environment,
queue, project, minmemory, maxmemory)`

Executes a remote job. *executable* indicates the executable to be run. Arguments can be passed to the executable using *arguments*. If present, the *directory* argument specifies the remote directory in which the job will be executed. *Stdout* and *stderr* allow the redirection of the output and error streams to a remote file. *Stdin* allows the redirection of the standard input from a remote file. If *redirect* is set to *true* the standard output and standard error of the remote job is redirected to the local console. The *host* argument allows the job to be executed on a specific host, and the *provider* argument allows the job to be executed using a specific provider.

The rest of the arguments are passed to the underlying provider.

`transfer(srcfile, srcdir, srchost, destfile, destdir, desthost, provider)`

Transfers a file. The file can be transferred between the local machine and a remote machine, or between two remote machines. The name of the source file is specified by the *srcfile* argument. If present, *destfile* specifies the name of the target file, otherwise the source file name is used.

The *srcdir* argument indicates the directory on the source machine where the source file can be found. If the *srcdir* argument is missing, the default directory will be assumed (provider dependent).

The *destdir* argument indicates the directory on the target machine where the file will be copied. If the *destdir* argument is missing, the default directory will be assumed (provider dependent).

Srchost and *desthost* indicate the source and the target machines respectively, while the *provider* argument can be used to force the scheduler to use a specific provider, or in the event a scheduler is not used.

`file:list(dir, host, provider)`

Returns a list of files in a directory specified by *dir*, on the *host* machine. The *provider* argument can be used to select a specific provider for the operation. *Provider* defaults to the *local* provider.

`file:remove(name, host, provider)`

Removes a file specified by *name*, on the *host* machine. The *provider* argument can be used to select a specific provider for the operation. *Provider* defaults to the *local* provider.

`file:exists(name, host, provider)`

Returns *true* if the file specified by *name* exists on the **host* machine. The **provider* argument can be used to select a specific provider for the operation. **Provider* defaults to the *local* provider.

`dir:make(name, *host, *provider)`

Creates a directory specified by *name*, on the **host* machine. The **provider* argument can be used to select a specific provider for the operation. **Provider* defaults to the *local* provider.

`dir:remove(name, *host, *provider)`

Removes an empty directory.

`file:isDirectory(name, *host, *provider)`

Returns *true* if the file specified by *name* exists on the **host* machine and it is a directory. The **provider* argument can be used to select a specific provider for the operation. **Provider* defaults to the *local* provider.

`file:chmod(name, mode, *host, *provider)`

Changes the permissions on the file specified with the *name* argument to the mode string indicated by the *mode* argument. If **host* and **provider* are present, the operation is done remotely using the respective provider.

`file:rename(from, to, *host, *provider)`

Renames a file. The source and target name are specified using the *from* and *to* arguments. If **host* and **provider* are present, the operation is done remotely.

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 available 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. 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`

It is not allowed to reproduce this document or the source. This documentation is copyrighted and is not distributed under the CoG Kit license.

B. 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](http://www-unix.globus.org/cog/contact/bugs/) system at <http://www-unix.globus.org/cog/contact/bugs/>

C. 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

- 0 TASKS
 - To do
 - Gregor, 3
- Administrative Contact, 39
- Contact, 39
- Grid
 - Scripting, 3
- Karajan, 3
 - Methods, 27
 - `*`, 27
 - `+`, 27
 - `-`, 27
 - `...`, 20
 - `/`, 27
 - `i`, 29
 - `i=`, 29
 - `==`, 27
 - `!`, 28
 - `! =`, 29
 - `?`, 16
 - `allocateHost`, 36
 - `and`, 26
 - `break`, 18
 - `cache`, 31
 - `channel`, 21, 24
 - `channel:from`, 20
 - `channel:to`, 20
 - `checkpoint`, 30
 - `choice`, 15
 - `classOf`, 34
 - `closeStream`, 32
 - `concat`, 32
 - `condition`, 16
 - `contains`, 31
 - `continue`, 18
 - `default`, 20, 22
 - `dir:make`, 38
 - `dir:remove`, 38
 - `discard`, 21
 - `echo`, 30, 31
 - `element`, 20, 21, 23, 24
 - `elementdef`, 13
 - `elementList`, 31
 - `else`, 18
 - `equals`, 27, 28
 - `equalsNumeric`, 28
 - `execute`, 37
 - `executeElement`, 22, 30, 31
 - `executeFile`, 31
 - `executeMain`, 33
 - `false`, 27
 - `file:chmod`, 38
 - `file:contains`, 31
 - `file:execute`, 31
 - `file:exists`, 37
 - `file:isDirectory`, 38
 - `file:list`, 37
 - `File:read`, 32
 - `file:read`, 32
 - `file:remove`, 37
 - `file:rename`, 38
 - `filter`, 32
 - `for`, 15, 16
 - `future`, 12, 15, 21
 - `futureIterator`, 12, 15, 21
 - `generateError`, 29
 - `getField`, 34
 - `global`, 20
 - `greaterOrEqual`, 29
 - `greaterThan`, 28
 - `handler`, 35
 - `host`, 35, 37
 - `host:hasService`, 37
 - `if`, 18
 - `ignoreErrors`, 29
 - `include`, 12, 13, 31
 - `invokeMethod`, 33
 - `isDefined`, 20
 - `karajan`, 6, 13
 - `lessOrEqual`, 29
 - `lessThan`, 29
 - `list`, 7, 10, 24
 - `list:append`, 24, 25
 - `list:butFirst`, 25
 - `list:butLast`, 25
 - `list:concat`, 25
 - `list:first`, 25
 - `list:isEmpty`, 25
 - `list:last`, 25
 - `list:prepend`, 24
 - `list:size`, 25
 - `map`, 26, 31
 - `map:contains`, 26
 - `map:delete`, 26
 - `map:entry`, 26

map:get, 26
map:put, 26
map:size, 26
matches, 32
named, 13
namespace, 12, 13
new, 33
not, 27
null, 34
number, 5, 14
numberFormat, 31
onError, 29
optional, 21, 24
or, 26
outputStream, 32
parallel, 9, 15
parallelChoice, 15
parallelElement, 23, 24
parallelFor, 16
print, 6, 8–10, 30, 31
print(v), 10
product, 27
project, 6, 13, 30
quoted, 20
quotedlist, 14
quotient, 27
readFile, 32
remainder, 28
resources, 35
restartOnError, 29
scheduler, 34
securityContext, 36
securityContextProperty, 36
sequential, 15, 18
service, 35
set, 10, 19, 20
split, 32
sqrt, 28
square, 28
string, 5, 14
subtraction, 27
sum, 27
task:execute, 36
then, 18
transfer, 37
true, 27
uid, 32
unsynchronized, 15
vargs, 20
variable, 14
wait, 30
waitForEvent, 34
while, 16, 18

Registration, 2

Workflow, 3
Karajan, 3