

Java CoG Kit Karajan Workflow Reference Manual

4.1.4

From Java CoG Kit

XML

Mike Hategan and Gregor von Laszewski (gregor@mcs.anl.gov)

Administrative Notes:

Editing

The maintenance of this document is rather simple. We do *NOT* recommend that you edit this document in a single edit. Instead, we do require the edit as part of a subsection. This will prevent the chance that data will get lost.

Versioning

It is extremely easy to adapt the Document to a new version. All that needs to be done is creating a new page with the appropriate version number and pasting and copying the contents of the current document into it. Then you just have to replace the view places where the actual version occurs. This is best done with an editor such as emacs that has easy query replace capabilities.

Contents

- 1 About this Document
 - 1.1 Command Line
 - 1.2 Options
 - 1.3 Experimental Options
 - 1.4 File
 - 1.5 Arguments
- 2 The Karajan Language
 - 2.1 The Karajan Syntax
 - 2.1.1 Elements
 - 2.1.2 Identifiers
 - 2.1.3 Arguments
 - 2.1.4 Named Arguments
 - 2.1.5 Unnamed Arguments
 - 2.1.6 Expressions
 - 2.1.7 Operators
 - 2.1.8 Quoted Lists
 - 2.1.9 Programs
 - 2.1.10 Comments
 - 2.2 The XML Syntax
 - 2.2.1 Particularities of Using XML
 - 2.3 Parameters and Return Values
 - 2.3.1 Parameters and Arguments
 - 2.3.1.1 Single Value Arguments
 - 2.3.1.2 Channels
 - 2.3.1.3 Argument Mapping
 - 2.3.1.4 Optional Arguments
 - 2.3.2 Return values
 - 2.3.3 Argument Evaluation Order

- 2.4 Variables and Scope
 - 2.4.1 Global Variables
 - 2.4.2 Variable Expansion
 - 2.4.3 Futures
- 3 Modularisation
 - 3.1 Source Files
 - 3.2 Libraries
 - 3.3 Namespaces
- 4 Kernel Library
 - 4.1 Kernel Constants
 - 4.1.1 kernel:true
 - 4.1.2 kernel:false
 - 4.1.3 kernel:cmdline:arguments
 - 4.1.4 kernel:user.home
 - 4.1.5 kernel:user.name
 - 4.2 Kernel Elements
 - 4.2.1 kernel:project
 - 4.2.2 kernel:import
 - 4.2.3 kernel:export
 - 4.2.4 kernel:define
 - 4.2.5 kernel:namespace
 - 4.2.6 kernel:elementdef
 - 4.2.7 kernel:named
 - 4.2.8 kernel:number
 - 4.2.9 kernel:string
 - 4.2.10 kernel:variable
 - 4.2.11 kernel:quotedlist
 - 4.2.12 kernel:cache
- 5 System Library
 - 5.1 Flow Control Elements
 - 5.1.1 sys:sequential
 - 5.1.2 sys:parallel
 - 5.1.3 sys:unsynchronized
 - 5.1.4 sys:choice
 - 5.1.5 sys:catch
 - 5.1.6 sys:guard
 - 5.1.7 sys:race
 - 5.1.8 sys:for
 - 5.1.9 sys:parallelFor
 - 5.1.10 sys:while
 - 5.1.11 sys:condition
 - 5.1.12 sys:break
 - 5.1.13 sys:continue
 - 5.1.14 sys:if
 - 5.1.15 sys:then
 - 5.1.16 sys:exclusive
 - 5.2 Elements Dealing with Variables and Arguments
 - 5.2.1 sys:set
 - 5.2.2 sys:default
 - 5.2.3 sys:maybe
 - 5.2.4 sys:global
 - 5.2.5 channel:to
 - 5.2.6 channel:from
 - 5.2.7 channel:close
 - 5.2.8 channel:fork
 - 5.2.9 sys:isDefined
 - 5.2.10 sys:quoted
 - 5.2.11 sys:discard
 - 5.2.12 sys:future

- 5.2.13 sys:futureIterator
- 5.2.14 sys:each
- 5.3 Element Definition Elements
 - 5.3.1 sys:element
 - 5.3.2 sys:parallelElement
 - 5.3.3 sys:channel
 - 5.3.4 sys:optional
 - 5.3.5 sys:self
- 5.4 Service Interaction Elements
 - 5.4.1 sys:remote
- 5.5 List Manipulation Elements
 - 5.5.1 list:list
 - 5.5.2 list:append
 - 5.5.3 list:prepend
 - 5.5.4 list:join
 - 5.5.5 list:size
 - 5.5.6 list:first
 - 5.5.7 list:last
 - 5.5.8 list:butFirst
 - 5.5.9 list:butLast
 - 5.5.10 list:isEmpty
- 5.6 Map Elements
 - 5.6.1 map:map
 - 5.6.2 map:entry
 - 5.6.3 map:put
 - 5.6.4 map:delete
 - 5.6.5 map:get
 - 5.6.6 map:size
 - 5.6.7 map:contains
- 5.7 Logic Elements
 - 5.7.1 sys:and
 - 5.7.2 sys:or
 - 5.7.3 sys:not
 - 5.7.4 sys:equals
 - 5.7.5 sys:true
 - 5.7.6 sys:false
- 5.8 Numeric Elements
 - 5.8.1 math:sum
 - 5.8.2 math:product
 - 5.8.3 math:subtraction
 - 5.8.4 math:quotient
 - 5.8.5 math:remainder
 - 5.8.6 math:square
 - 5.8.7 math:sqrt
 - 5.8.8 math:equalsNumeric
 - 5.8.9 math:greaterThan
 - 5.8.10 math:lessThan
 - 5.8.11 math:greaterOrEqual
 - 5.8.12 math:lessOrEqual
 - 5.8.13 math:min
 - 5.8.14 math:max
 - 5.8.15 math:int
 - 5.8.16 math:ln
 - 5.8.17 math:exp
 - 5.8.18 math:random
- 5.9 Error Handling Elements
 - 5.9.1 sys:ignoreErrors
 - 5.9.2 sys:restartOnError
 - 5.9.3 sys:generateError

- 5.9.4 sys:onError
- 5.10 String Elements
 - 5.10.1 str:concat
 - 5.10.2 str:split
 - 5.10.3 str:strip
 - 5.10.4 str:matches
 - 5.10.5 str:nl
 - 5.10.6 str:chr
- 5.11 Miscellaneous Elements
 - 5.11.1 sys:print
 - 5.11.2 sys:echo
 - 5.11.3 sys:checkpoint
 - 5.11.4 sys:wait
 - 5.11.5 sys:time
 - 5.11.6 sys:file
 - 5.11.7 sys:executeElement
 - 5.11.8 sys:elementList
 - 5.11.9 sys:cacheOn
 - 5.11.10 sys:numberFormat
 - 5.11.11 sys:file
 - 5.11.12 sys:uid
 - 5.11.13 sys:file
 - 5.11.14 sys:file
 - 5.11.15 sys:outputStream
 - 5.11.16 sys:closeStream
 - 5.11.17 sys:sort
 - 5.11.18 sys:dot
 - 5.11.19 sys:cross
 - 5.11.20 sys:stats
 - 5.11.21 sys:filter
 - 5.11.22 sys:info
- 5.12 Notes
- 6 Task Library
 - 6.1 Task Elements
 - 6.1.1 task:scheduler
 - 6.1.2 task:handler
 - 6.1.3 task:resources
 - 6.1.4 task:host
 - 6.1.5 task:service
 - 6.1.6 task:securityContext
 - 6.1.7 task:allocateHost
 - 6.1.8 task:host
 - 6.1.9 task:execute
 - 6.1.10 task:transfer
 - 6.1.11 task:file:list
 - 6.1.12 task:file:remove
 - 6.1.13 task:file:exists
 - 6.1.14 task:dir:make
 - 6.1.15 task:dir:remove
 - 6.1.16 task:file:isDirectory
 - 6.1.17 task:file:chmod
 - 6.1.18 task:file:rename
 - 6.1.19 task:SSHSecurityContext
 - 6.1.20 task:InteractiveSSHSecurityContext
 - 6.1.21 task:passwordAuthentication
 - 6.1.22 task:publicKeyAuthentication
- 7 Java Library
 - 7.1 Java Elements
 - 7.1.1 java:new

- 7.1.2 java:invokeMethod
- 7.1.3 java:executeMain
- 7.1.4 java:getField
- 7.1.5 java:waitForEvent
- 7.1.6 java:classOf
- 7.1.7 java:null
- 8 HTML Library
 - 8.1 HTML Elements
 - 8.1.1 html:write
 - 8.1.2 html:quickstart
 - 8.1.3 html:html
 - 8.1.4 html:head
 - 8.1.5 html:title
 - 8.1.6 html:body
 - 8.1.7 html:table
 - 8.1.8 html:tr
 - 8.1.9 html:td
 - 8.1.10 html:th
 - 8.1.11 html:h1
 - 8.1.12 html:h2
 - 8.1.13 html:h3
 - 8.1.14 html:h4
 - 8.1.15 html:h5
 - 8.1.16 html:h6
 - 8.1.17 html:ul
 - 8.1.18 html:pre
 - 8.1.19 html:br
 - 8.1.20 html:li
 - 8.1.21 html:a
 - 8.1.22 html:anchor
 - 8.1.23 html:img
 - 8.1.24 html:text
- 9 Forms Library
 - 9.1 Concepts
 - 9.1.1 Component ID
 - 9.1.2 Component Layout
 - 9.1.3 Component Alignment
 - 9.2 Form Elements
 - 9.2.1 form:form
 - 9.2.2 form:hbox
 - 9.2.3 form:vbox
 - 9.2.4 form:label
 - 9.2.5 form:button
 - 9.2.6 form:checkBox
 - 9.2.7 form:radioBox
 - 9.2.8 form:radioButton
 - 9.2.9 form:textField
 - 9.2.10 form:passwordField
 - 9.2.11 form:comboBox
 - 9.2.12 form:comboBox
 - 9.2.13 form:HSeparator
 - 9.2.14 form:VSeparator
 - 9.2.15 form:filler
 - 9.2.16 form:messageDialog
- 10 Restart Library
 - 10.1 rlog:restartLog
 - 10.2 rlog:logged
- 11 Service
 - 11.1 Using the Service

- 11.2 Shared or Personal
- 11.3 Limiting Access to Resources in Shared Mode
- 11.4 Communication Layer Configuration
- 11.5 The Secure (Grid-Mapped) Local Provider
 - 11.5.1 Pre-built Packages
 - 11.5.2 Building From Source
- 12 Embedding Karajan into Java
- 13 Notes
- 14 References

About this Document

This page contains the Reference manual. Although it contains all of the features, we do recommend that you start with the examples document at http://wiki.cogkit.org/index.php/Java_CoG_Kit_Workflow_Guide

Command Line

Usage:

```
cog-workflow <options> file <arguments>
```

Options

(-execute | -e) <string>

Execute the script given as argument

-showstats

Show various execution statistics at the end of the execution

-debug

Enable debugging. This will enable a number of internal tests at the expense of speed. You should not use this since it is useful only for catching subtle consistency issues with the interpreter.

-monitor

Shows a resource monitor

-dumpstate

If specified, in case of a fatal error, the interpreter will dump the state in a file

-intermediate

Saves intermediate XML code resulting from the translation of .k files

(-help | -h)

Display usage information

Experiemental Options

The following options have been added and are considered to be experiemental. They are still in the debugging stage.

-debugger

Starts the internal graphical debugger

-cache

Enables cache persistence

File

The file represents the name of the script to run, in either CoG Kit XML or CoG Kit K syntax. A CoG Kit karajan file structure is rather simple and similar to what you would be used to form other programming languages. A CoG Kit Karajan file starts with a series of imports, followed by a block of statements. It is important to note that we have two ways to express CoG Kit karajan syntax. Both are explained in more detail in the following sections.

All Karajan files usually start with the import of the most generally used libraries

in k syntax:

```
import("cogkit.xml")
print("hello world")
```

in xml syntax:

```
<import file="cogkit.xml">
<print message="Hello World">
```

Arguments

Arguments to the script can be specified after the file name. They are distinct from the interpreter options, and are passed to the script as a constant, in the form of a list named "cmdline:arguments". More information about commandlines can be found [here](http://wiki.cogkit.org/index.php/Java_CoG_Kit_Karajan_Workflow_Reference_Manual_4.1.4#kernel:cmdli)

(http://wiki.cogkit.org/index.php/Java_CoG_Kit_Karajan_Workflow_Reference_Manual_4.1.4#kernel:cmdli)

The Karajan Language

Karajan supports two syntax modes: a syntax called k and an equivalent form called XML. There is no difference on the semantic level between the two forms.

The Karajan Syntax

The following conventions are used:

```
{(xy) is 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
ε represents the empty production
```

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()
```

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 operators ('+', '-', '*', '/', '%', '^', '=', '<', '>', '&', '|'). Identifiers are case insensitive.

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

Example:

```
var, i, v123, @, a$, big_list, grid:task, file.list
```

Identifiers cannot begin with a digit.

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] | ε
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
```

Named Arguments

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

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

Example:

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

Unnamed Arguments

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

```
unnamed_argument ::= numeric_literal | string_literal | variable | quoted_list |
                    | element | expression
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"))
```

Expressions

Expressions consist of unnamed arguments to which operators are applied. Parentheses can be used to override the default precedence of operators in expressions.

```
expression ::= unnamed_argument operator unnamed_argument |
              | '(' expression ')'
```

Examples of expressions:

```
'set(a, 1+2*3-4)
'set(b, subtraction(sum(1, product(2, 3)), 4))
'print(a, " = ", b)
```

```
<set name="a">
  <subtraction>
    <sum>
      <product>
        <number>2</number>
        <number>3</number>
      </product>
    </sum>
    <number>4</number>
  </subtraction>
</set>
<print message="a = {a}"/>
```

Operators

The native Karajan syntax supports basic arithmetic and logic operators:

```
operator ::= '*' | '/' | '%' | '+' | '-' | '<=' | '>=' | '<' | '>' |
           '| '==' | '!=' | '&' | '|'
```

The following lists enumerates operators in the order of precedence, starting with the highest precedence. While the XML syntax does not support the use of operators, each operator has an equivalent element which can be used in both syntaxes (shown in parentheses).

- Multiplicative operators
 - * (product) Multiplication
 - / (quotient) Division
 - % (remainder) Remainder
- Additive operators
 - + (sum) Addition
 - - (subtraction) Subtraction
- High priority relational operators
 - <= (lessOrEqual) Less or equal
 - >= (greaterOrEqual) Greater or equal
 - < (lessThan) Strictly less
 - > (greaterThan) Strictly greater
- Low priority relational operators
 - == (equals) Equals
 - != (No equivalent, but not(equals(...)) can be used) Does not equal
- Multiplicative logic operators
 - & (and) Logical AND
- Additive logic operators
 - | (or) Logical OR

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])
```

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.

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
*/
```

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.

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.

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).

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.

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

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

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:

k:

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

xml:

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

See Also: `list`, `number`, `string`

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.

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>
...

```

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())
```

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:

```

//The following defines an element foo() which takes no
//arguments and returns "Message" and nl = false()
element(foo, [
  "Message", nl = false()
])
print(foo())

```

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

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).

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
  v      // the new binding shadows the one from frame n
  // $v$ now refers to the binding on frame n+1
)
print(v) //v refers again to the binding on frame n
//Therefore the printed value will be 1
...

```

```

...
<set name="v" value="1"/>
<list>
  <variable>v</variable>
  <set name="v" value="2"/>
  <variable>v</variable>
</list>
<print>
  <variable>v</variable>
</print>
...

```

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

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()

```

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: }"/>
```

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`.

Modularisation

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. [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 `import` (equivalent to `include`) element:

```
import("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:

```
import("task.xml")
```

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

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`.

Kernel Library

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.

Kernel Constants

kernel:true

`true`
Represents the **true** boolean value

kernel:false

`false`
Represents the **false** boolean value

kernel:cmdline:arguments

`cmdline:arguments`

Holds a list with the command line arguments (if any). Arguments to the CoG Kit karajan script can be specified after the file name. They are distinct from the interpreter options, and are passed to the script as a constant, in the form of a list named "cmdline:arguments".

kernel:user.home

`user.home`
Contains the path to the user home directory

kernel:user.name

`user.name`
Contains the current user's name

Kernel Elements

kernel:project

`kernel:project(stdout)`

Aliases: `karajan`

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

kernel:import

`kernel:import(file, ...)`

Aliases: `include`

Executes the file specified by the *file* argument. All arguments received on the default channel are considered to be element definitions (created with `export`), which `import` binds to the parent environment, such that after `import` completes execution, the definitions will be available for use.

```
-----  
foo(  
  import("file.k")  
) //scope of foo() ends  
-----
```

`import` uses the library search path specified in `etc/karajan.properties`, which defaults to searching the current directory first, then the Java class path. If no file with the given name is found in the library search path, `import` will fail.

kernel:export

`kernel:export(name, value)`

Returns the pair (*name*, *value*). The *value* argument should be a lambda, which can be bound by `import`.

```
-----  
export(foo, element([x], print(x)))  
-----
```

It is also possible for `export` to be used without arguments, but with a set of *immediately enclosed* definitions. In this case it would take all the definitions and export them. This makes it easier to have old code somewhat cleanly converted:

```
-----  
export(  
  element(x, [], print("x"))  
  element(y, [], print("y"))  
)  
-----
```

kernel:define

`kernel:define(name, value)`

Binds a lambda, specified by the *value* argument to the given *name*.

kernel:namespace

`kernel: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.

kernel:elementdef

`kernel:elementdef(type, classname)`

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

kernel:named

`kernel: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.

kernel:number

`kernel: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.

kernel:string

`kernel:string(value)`

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

kernel:variable

`kernel: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].

kernel:quotedlist

`kernel: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)
```

kernel:cache

`kernel:cache()`

Caches the evaluation of the arguments. Upon subsequent executions of `cache`, the arguments will not be re-evaluated. Instead, the cached values will be returned. Cache does not make any checks for the invariance of the evaluation of the arguments.

System Library

Files: `sys.k`, `sys.xml`

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

Flow Control Elements

sys:sequential

`sys:sequential()`

Executes all arguments in sequence.

sys:parallel

`sys:parallel()`

Executes all arguments in parallel.

sys:unsynchronized

`sys: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`

sys:choice

`sys: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.

See also: `catch`

sys:catch

`sys:catch(match)`

`Catch` will match the error variable against the regular expression in *match*. If successful, it will execute the rest of its arguments, otherwise it will fail with the last failure encountered. `Catch` can be used with `choice` to selectively handle specific failures:

```
choice(  
  ...  
  catch(".*File not found.*"  
    print("File not found")  
  )  
  catch(".*Connection refused.*"  
    print("Connection refused")  
  )  
)
```

sys:guard

`sys:guard()`

`Guard` expects two sub-elements. It will execute the first, then the second, even if the first one fails. In other words, the second sub-element will always be executed. If the first element failed, after executing the second element, `guard` will also fail. If the second element fails, `guard` will fail with the same error, regardless of whether the first element failed or not. `Guard` can be used to implement clean-up actions, in a fashion similar to `try/finally` from C++, Java or Python:

```
set(myhost, "sunny.mcs.anl.gov")
transfer(srcfile="a", desthost=myhost)
guard(
  sequential(
    execute(executable="/usr/bin/hammer", arguments="a", host=myhost,
      provider="gt2")
  )
  sequential(
    //always clean up
    file:remove(name="a", host=myhost, provider="gridftp")
  )
)
```

sys:race

`sys:race()`

Aliases: `parallelChoice`

Can be used to race a number of arguments. `Race` 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 `race` will fail.

`Race` is similar in behavior to the *discriminator* workflow pattern.

sys:for

`sys: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 *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:

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

will return `true`

sys:parallelFor

`sys: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 (which is also the case with `for`). Each scope will have the variable indicated by the *name* argument set to

one of the values obtained from the *in* argument.

sys:while

`sys: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>
    <number>1</number>
    <number>2</number>
    <number>3</number>
  </while>
</list>
```

sys:condition

sys:condition(*value*)

Aliases: ?

Evaluates the *value* argument and returns its value on the **condition** channel.

sys:break

sys:break()

Can be used to break out of a `while` loop. By contrast with using the **condition** channel, `break` will immediately exit the loop, no matter how deep the nesting level. It does that by generating a failure, which is intercepted by the enclosing `while`.

sys:continue

sys:continue()

Can be used to skip the evaluation of the remaining arguments in an iteration in a `while` loop and jump to

the next iteration. Similar to `break`, `continue` achieves its purpose by generating a failure which is intercepted by `while`.

sys:if

`sys:if()`

Executes its elements using the following scheme:

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

Informally:

```

if(
  <condition> <then>
  [<condition2> <then2>
   <condition3> <then3>
   ...]
  [<else>]
)

```

`<condition>`, `<then>`, and `<else>` can be any elements. However, each `<condition>`, `<then>`, and `<else>` must be only one element (possibly with multiple child elements). For convenience and clarity `then` and `else` can be used.

sys:then

`sys:then()`

Aliases: `else`

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

```

if(
  a == 1
  then(print("a is 1"))
  a == 2
  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>

```

sys:exclusive

`sys:exclusive()`

Defines a mutual exclusion block. It guarantees that at any give time, within a given execution, only one instance of this (lexically) `exclusive` element is executing.

Elements Dealing with Variables and Arguments

sys:set

`sys: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>

```

sys:default

sys: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
!//b is already accessible within the current scope
-----

```

sys:maybe

sys:maybe()

Evaluates its arguments. If the evaluation completes successfully, it returns all the arguments. If the evaluation fails at any point, `maybe` completes without returning anything. In particular, `maybe` can be used in extending existing elements with optional arguments:

```

-----
element(one, [a, b, optional(c, d)]
  print("a = {a}", "b = {b}", maybe("c = {c}"), maybe("d = {d}"))
)
element(two, [a, b, optional(c, d)]
  one(a = a, b = b, maybe(c = c), maybe(d = d))
)
-----

```

sys:global

sys: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.

sys:...()

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

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

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

channel:from

```
channel:from(name, <varies>)
```

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

channel:close

```
channel:close(name)
```

Closes a channel. All iterations that are active on that channel and waiting for values will complete.

channel:fork

```
channel:fork(name, count)
```

Splits a channel into a number of identical channels and returns these channels. All values written to the initial channel will be available in all the forked channels. Reading from the original channel will cause inconsistencies and should not be done. When the original channel is closed, all the forked channels will also be closed.

sys:isDefined

```
sys:isDefined(name)
```

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

sys:quoted

```
sys:quoted(name)
```

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

sys:discard

```
sys: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.

sys:future

```
sys: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.

sys:futureIterator

```
sys: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.

sys:each

```
sys:each(items)
```

Returns all elements in *items* as separate values. It is roughly equivalent to the following:

```
-----
for(i, items, i)
-----
```

Element Definition Elements

sys:element

```
sys: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, 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)

```

Each definition of an element keeps a reference to the environment that was used at the time of the definition. When evaluated, elements in the body of the definition will be resolved by first searching in the local scope (eventually for elements defined by the execution of the body of this element) and, if not found, in the environment that was used at the time of the definition. In the following example, the result will be the printing of the string "a":

```

element(foo, [
  element(a, [], print("a"))
  //return an anonymous element
  element([
    a() //this is the a() defined above
  ])
)
set(b, foo())
element(a, [], print("b"))
executeElement(b)

```

This behaviour is particularly important when `import` and `export` are used.

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 = "{args}">
    <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>
  <to name="channelOne">
    <number>5</number>
    <number>6</number>
    <number>7</number>
    <number>8</number>
  </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"/>

```

sys:parallelElement

`sys: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>

```

sys:channel

`sys: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.

sys:optional

`sys: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.

sys:self

`sys:self()`

Self can be used to build recursive anonymous elements:

```

iset(f
  element([x]
    if(x == 0 1 x*self(x - 1))
  )
)
print(executeElement(f, 6))

```

Service Interaction Elements

sys:remote

`sys:remote(host)`

Uses the Karajan:Service that runs on the specified *host* to evaluate the rest of the arguments. The element supports forwarding of the variable environment to the remote service ^[1], and returning values from the service. Therefore the following code will work as expected:

```

iset(a, 1)
iset(b
  remote("https://somehost:1984", a+1)
)
print(b) //will print 2

```

A consequence is that if `print` is used remotely, the actual output is going to be automatically and transparently forwarded (since it is merely a return value on a particular channel) to the execution instance that initiated the remote execution.

It is also possible to use this feature recursively, such that a remote code snippet in turn delegates part of the execution to other services.

Please note that the code inside `remote` might reference entities that are sensitive to the actual location of the execution. In particular, the meaning of “localhost” if used with `execute` or `transfer` will not refer to the machine where the execution originated.

Element resolution is performed separately when a script is submitted to a service. In other words, `import`-ed files on the client side will be re-imported on the service side, if and only if they are system libraries. If imports refer to user-defined libraries not part of the system libraries, the code defining the libraries will be forwarded to the service, thus enabling the use of client-side user-defined elements on the remote side. Similarly, in-line user defined elements are also available for use on the remote side:

```

element(foo, [], print("Foo"))
remote("https://somehost:1984"
  foo()
)

```

List Manipulation Elements

list:list

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

Constructs a list from values received on the default channel.

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

list:append

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

Works like 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 prepend:

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

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

list:join

```
list:join(...)
```

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

list:size

```
list:size(list)
```

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

list:first

```
list:first(list)
```

Returns the first element in a list.

list:last

```
list:last(list)
```

Returns the last element in a list.

list:butFirst

```
list:butFirst(list)
```

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

list:butLast

```
list:butLast(list)
```

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

list:isEmpty

```
list:isEmpty(list)
```

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

Map Elements

map:map

```
map:map(...)
```

Returns a map with the entries received on the default channel. See `entry`.

map:entry

```
map:entry(key, value)
```

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

map:put

```
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:delete(map, key)
```

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

map:get

```
map:get(map, key)
```

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

map:size

```
map:size(map)
```

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

map:contains

`map:contains(map, key)`

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

Logic Elements

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

sys:and

`sys:and(...)`

Aliases: &

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

sys:or

`sys:or(...)`

Aliases: |

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

sys:not

`sys:not(value)`

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

sys>equals

`sys>equals(value1, value2)`

Aliases: ==

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

sys:true

`sys:true()`

Returns `true`.

sys:false

`sys:false()`

Returns `false`.

Numeric Elements

math:sum

`math:sum(...)`

Aliases: +

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

math:product

`math:product(...)`

Aliases: *

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

math:subtraction

`math:subtraction(from, value)`

Aliases: -

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

math:quotient

`math:quotient(divisor, dividend)`

Aliases: /

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

math:remainder

`math:remainder(divisor, dividend)`

Aliases: %

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

math:square

`math:square(value)`

Returns the square of a number.

math:sqrt

`math:sqrt(value)`

Returns the square root of a number.

math:equalsNumeric

`math: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.

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

```
<math:equalsNumeric value1 = "1">
  <number>1</number>
</math:equalsNumeric>
<math:equalsNumeric value1 = "2" value2 = "2.0"/>
<math:equals value1 = "2">
  <number>2</number>
</math:equals>
<math:equalsNumeric>
  <list items = "1, 2, 3"/>
  <list>
    <number>1</number>
    <number>2</number>
    <string>3</string>
  </list>
</math:equalsNumeric>
```

math:greaterThan

`math:greaterThan(value1, value2)`

Aliases: >

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

math:lessThan

`math:lessThan(value1, value2)`

Aliases: <

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

math:greaterOrEqual

`math:greaterOrEqual(value1, value2)`

Aliases: >=

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

math:lessOrEqual

```
math:lessOrEqual(value1, value2)
```

Aliases: <=

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

math:min

```
math:min(...)
```

Returns the minimum of all numeric values on the default channel.

math:max

```
math:max(...)
```

Returns the maximum of all numeric values on the default channel.

math:int

```
math:int(value)
```

Returns the integer part of the argument, where “integer part” is to be understood in the mathematical sense (also equivalent to the floor function).

math:ln

```
math:ln(value)
```

Returns the natural logarithm of the value argument.

math:exp

```
math:exp(value)
```

Returns e raised to the power of *value*.

math:random

```
math:random()
```

Returns a pseudo-random number in the interval [0,1) with a uniform distribution [2].

Error Handling Elements

sys:ignoreErrors

```
sys: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.

sys:restartOnError

```
sys:restartOnError(match, times)
```

Evaluates its arguments in sequence. If a failure matching the regular expression in *match* occurs, all arguments will be re-evaluated for a maximum of times indicated by the *times* argument.

sys:generateError

```
sys:generateError(error, *exception)
```

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

sys:onError

```
sys:onError(match)
```

`OnError` allows the definition of a custom error handler. Handlers defined with `onError` are valid for 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.

String Elements

str:concat

```
str:concat(...)
```

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

str:split

```
str:split(string, separator)
```

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

str:strip

```
str:strip(value)
```

Strips all leading and trailing whitespace characters from *value*.

str:matches

```
str:matches(string, regexp)
```

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

str:nl

```
str:nl()
```

Returns the new-line separator.

str:chr

```
str:chr(code)
```

Returns the character whose code is represented by *code*.

Miscellaneous Elements

sys:print

```
sys: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.

sys:echo

```
sys: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, `echo` 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.

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

sys:checkpoint

Allows the configuration of checkpointing. If the **now* argument 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` format.

sys:wait

```
sys: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.

sys:time

```
sys:time()
```

Evaluates all arguments and returns the total time elapsed, in milliseconds.

```

-----
set (t
  time(
    execute(executable="/bin/wait", arguments="10", ...)
  )
)
print("Job done in ", t/1000, " seconds.")
-----

```

sys:file

```
sys:file:execute(file)
```

Aliases: `executeFile`

Parses and executes a file. The difference between `import` and `file:execute` is that `file:execute` always parses and executes the file when evaluated, unlike `import` 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.

sys:executeElement

```
sys: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. It is also possible to pass named arguments directly using the named form. However, this does not allow passing of arguments named *element* or *args*.

If *args* is not present, . . . will be mapped to named arguments according to the rules in [Argument Mapping](#)

sys:elementList

`sys: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`

sys:cacheOn

`sys:cacheOn(value)`

Caches all return values of the rest of its arguments based on the value of the *value* argument. Subsequent evaluations of this element in which the value 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 `cacheOn` element. In other words, if another `cacheOn` element exists, it will not use the values cached by this element, irrespective of the value of the *value* 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.

sys:numberFormat

`sys: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.

In short, the following characters can be used in patterns:

| | |
|---|-------------------------------|
| # | Digit; zero not shown |
| 0 | Digit; zero is shown |
| . | Decimal separator |
| ' | Grouping separator |
| E | Scientific notation separator |

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

sys:file

`sys: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.

sys:uid

`sys: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.

sys:file

`sys: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.

sys:file

`sys:file:write(name, *append, ...)`

`File:write` writes all arguments received on the default channel to the file with the given *name*. The file is truncated first, unless the **append* argument is `true`. When the element terminates, either successfully or not, the file is closed.

sys:outputStream

`sys: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.

sys:closeStream

`sys:closeStream(stream)`

Closes a stream opened with `outputStream`.

sys:sort

`sys:sort(*descending, ...)`

Returns all arguments in sorted order. The values are sorted in ascending order, unless the **descending* argument is set to `true`.

sys:dot

`sys:dot(...)`

Returns the "dot product" of all the arguments, which are expected to be some form of vectors (lists or channels). `Dot` returns the result asynchronously if any its arguments is a channel that is not closed. The returned values are lists with a value extracted from each of the vectors.

sys:cross

`sys:cross(...)`

Returns the "cross product" of its vector arguments. Each value returned is a list with a value from each vector. `Cross` does not work asynchronously.

sys:stats

`sys:stats(*asmap)`

If *asmap* is `false` then it returns a string summarizing information such as the amount of used and free memory, the number of CPUs, and the current number of active threads. If the *asmap* argument is set to `true`, it returns the information as a map with the following keys:

- `memused`
- `heapsize`
- `heapmax`
- `cpucount`
- `threadcount`

sys:filter

`sys: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*.

sys:info

`sys:info(*prefix, *name)`

Prints information (such as name and arguments) about elements. If the *prefix* argument is present, it will print information about all elements currently defined within that namespace prefix. If the *name* argument is present it will only print information about the element with the given name. If none of the two possible arguments are present, `info` will print information about all currently defined elements (within the scope of `info`).

Notes

1. At the time of this writing, futures are not properly forwarded if unbound at the time of the remote invocation, possibly causing errors or the remote execution to hang indefinitely.
2. The uniform distribution relies in the current implementation on the properties of the Java `Math.random()` function

Task Library

Files: task.k, task.xml

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.

Task Elements

task:scheduler

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

Defines the a `scheduler` to be used. A scheduler in Karajan has the role of managing resources and assigning abstract tasks (such as `execute` and `transfer`) to concrete resources. More details about the role of Karajan schedulers can be found in Karajan:The role of schedulers.

The type describes the particular type of scheduler that is desired. 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 following schedulers are currently available:

1. **Default** 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.
2. **Weighted** The weighted scheduler is an experimental adaptive scheduler that maintains a "performance" history of all the hosts that it manages. Each host starts with a score of 1. If a task fails on a host, its score is decreased by a certain factor, if a task succeeds, the score is increased by a certain factor, and so on. Scores are also temporarily decreased with each job running on a host. Periodically, a normalization of the scores is performed. The normalization process involves multiplying each score with the same factor such that the geometric average of the scores after the normalization is 1. The weighted scheduler has two host selection strategies: "random" and "best". The "best" strategy means that the host with the

highest score at the time of the submission of a task will be chosen for that task. By contrast the "random" strategy uses a weighted random choice, which gives higher chances for a host with a higher score. With a weighted random policy, every host, assuming that `scoreLowCap > 0`, and given a sufficiently large number of tasks, will eventually get a chance to be used (and thus possibly increase its score). The following properties are available for the weighted scheduler (default values are listed in parentheses, after the property name; factors are values with which the score for a host is multiplied in a certain event):

`connectionRefusedFactor`

(0.1) factor for connection refused exceptions while submitting to a host

`connectionTimeoutFactor`

(0.05) factor for connection timeout exceptions while submitting to a host

`jobSubmissionTaskLoadFactor`

(0.9) used when a job is submitted successfully; reversed when the job completes (either successfully or in failure)

`transferTaskLoadFactor`

(0.9) used when a task transfer is started; reversed when the transfer completes

`fileOperationTaskLoadFactor`

(0.95) used when a file operation is started; reversed when the operation completes

`successFactor`

(1.2) factor used upon successful completion of a task

`failureFactor`

(0.9) used when a task fails

`scoreHighCap`

(100) maximum value for a score

`scoreLowCap`

(0.001) minimum value for a score

`renormalizationDelay`

(100) number of tasks submitted after which a normalization occurs

`policy`

("random") use either a weighted "random" host selection policy or a "best" score host selection policy.

The scope of the `scheduler` is similar to a deeply accessible variable. In fact, the current implementation works by setting a hidden variable (named "#scheduler") in the current scope, and which would be visible in any dynamic sub-scope.

The following example shows a typical scheduler definition:

```

import("sys.k")
import("task.k")

scheduler("default"
  resources(
    host("host1", cpus = 256
      service("execution", provider = "gt2", jobManager = "PBS", uri = "host1.example.net:2119")
      service("file", provider = "gsiftp", uri = "host1.example.net:2911")
    )
    host("host2", cpus = 2
      service("execution", provider = "gt2", uri = "host2.example.net:2119")
      service("file", provider = "gsiftp", uri = "host2.example.net:2911")
    )
  )
  handlers = list(
    handler("execution", "gt2")
    handler("execution", "gt4")
    handler("file", "gsiftp")
    handler("file-transfer", "ssh")
  )
  properties = map(
    entry("jobsPerCpu", "1")
  )
)

```

Or in XML:

```
<import file="sys.xml"/>
<import file="task.xml"/>
<scheduler type = "default">
  <resources>
    <host name="host1" cpus="256">
      <service type="execution" provider="gt2" jobManager="PBS"
        uri = "host1.example.net:2119"/>
      <service type="file" provider="gsiftp" uri="host1.example.net:2911"/>
    </host>
    <host name="host2" cpus="2">
      <service type="execution" provider="gt2" uri="host2.example.net:2119"/>
      <service type="file" provider="gsiftp" uri="host2.example.net:2911"/>
    </host>
  </resources>
  <argument name="handlers">
    <list>
      <handler type="execution" provider="gt2"/>
      <handler type="execution" provider="gt4"/>
      <handler type="file" provider="gsiftp"/>
      <handler type="file-transfer" provider="ssh"/>
    </list>
  </argument>
  <argument name="properties">
    <map>
      <entry key="jobsPerCpu" value="1"/>
    </map>
  </argument>
</scheduler>
```

task:handler

`task: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. The 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.

task:resources

`task:resources(...)`

Encapsulates a set of hosts which can be specified using host.

task:host

`task: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.

task:service

```
task:service(type, provider, *uri, *project, *jobManager, *securityContext)
```

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 execute element.

The **jobManager* argument can be used to specify a job manager different from the default. Examples of job managers include *Fork*, *PBS*, and *Condor*.

A non-default security context can be specified using the **securityContext* argument.

task:securityContext

```
task:securityContext(provider, credentials)
```

Returns a Java CoG Kit abstraction security context. The returned context will be instantiated for the specified provider. The credentials argument can be used to pass a specific set of credentials to security context.

task:allocateHost

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

```

Or, in XML:

```

-----
<scheduler>
  ...
</scheduler>
<allocateHost name="host1">
  <execute executable="/bin/date" stdout="date" host="{host1}"/>
  <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.

task:host

```
task:host(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.

task:execute

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

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 **delegation* can be used to enable credential delegation with providers which support it. Credential delegation is disabled by default.

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

A native specification (such as a classic GRAM RSL, or WS-GRAM RSL) can be passed to the provider using the **nativespec* argument.

task:transfer

```
task:transfer(*srcfile, *srcdir, *srchost, *destfile, *destdir, *desthost,
*provider, *srcprovider, *destprovider, *thirdparty, *srcOffset, *length,
*destOffset)
```

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).

**Src*host and **dest*host 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. If the source and the destination use distinct providers, the **src*provider and **dest*provider arguments can be used.

The **third*party can be used to indicate that a direct transfer between two machines, none of which are the local host, is requested. At this time, only GridFTP supports third party transfers. By default, the Java CoG Kit Abstractions will use simulated third party transfers (routed through the local host) even if both the source and destination are different from the local host.

Partial transfers can be achieved using **src*Offset, **length*, and **dest*Offset. Currently these are only supported with GridFTP 3rd party transfers.

task:file:list

```
task: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.

task:file:remove

```
task: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.

task:file:exists

```
task: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.

task:dir:make

```
task: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.

task:dir:remove

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

Removes an empty directory.

task:file:isDirectory

```
task: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.

task:file:chmod

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

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.

task:file:rename

```
task: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.

task:SSHSecurityContext

```
task:SSHSecurityContext(credentials)
```

Instantiates a SSH security context. This is simply a convenience function for `securityContext("ssh", credentials)`.

task:InteractiveSSHSecurityContext

```
task:InteractiveSSHSecurityContext(*username, *privateKey, *nogui)
```

Instantiates a SSH security context which will lazily display a dialog window allowing the user to input a user-name/password pair or a user-name/private key/passphrase set. The dialog will only be displayed once per each instance of an interactive SSH security context.

If **username* and/or **privateKey* are specified, the values are used to pre-fill the corresponding dialog fields.

The `InteractiveSSHSecurityContext` makes use of a class present in the SSH provider of the Java CoG Kit. This class will try to determine whether a GUI can be displayed or not (by checking `GraphicsEnvironment.isHeadless()`). If a Swing dialog cannot be displayed, a text-mode interface is used instead. The **nogui* argument can be used to force the use of the text-mode interface (by setting it to `true`).

task:passwordAuthentication

```
task:passwordAuthentication(username, password)
```

Returns a *username/password* pair suitable to be used as a credential for a `securityContext`.

task:publicKeyAuthentication

```
task:publicKeyAuthentication(username, privatekey, passphrase)
```

Returns a *username/privatekey/passphrase* set which can be used as credentials for `securityContext`. The *privatekey* argument must point to a file containing the private key.

Java Library

Files: java.k, java.xml

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

Java Elements

java:new

`java: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
  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
  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`.

java:invokeMethod

`java: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 the types of 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`.

java:executeMain

```
java: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.

java:getField

```
java: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.

java:waitForEvent

```
java: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.

java:classOf

```
java:classOf(object)
```

Returns the Java class name of the specified *object*.

java:null

```
java: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.

HTML Library

Files: html.k, html.xml

The HTML library can be used to produce HTML output from Karajan scripts. It is incomplete, but able to generate simple HTML pages. It is listed here in the hope that it will be useful. However, not many guarantees are made about it. The library elements utilize the **html** channel.

test jump to html:write write

HTML Elements

html:write

`html:write(file, html)`

Writes all arguments received on the **html** channel to the specified file. It is roughly equivalent to the following:

```
file:write(name=file
  from(html
    ...
  )
)
```

The following code produces a simple HTML file:

```
import("html.k")
write(file="out.html"
  html(
    head(
      title("Title")
    )
    body(
      h1("Heading 1")
      text("Some text")
    )
  )
)
```

The XML syntax may appear more convenient for using the HTML library:

```
<import name="html.xml"/>
<write file="out.html">
  <html>
    <head>
      <title>Title</title>
    </head>
    <body>
      <h1>Heading 1</h1>
      <text>Some text</text>
    </body>
  </html>
</write>
```

html:quickstart

`html:quickstart(title, html)`

Produces an HTML skeleton output with the given title, and the arguments on the html channel substituted

inside the `<body>` tag.

html:html

```
html:html(html)
```

Generates the `<html></html>` tag pairs with the *html* arguments substituted inside.

html:head

```
html:head(html)
```

html:title

```
html:title(title)
```

html:body

```
html:body(*bgcolor, html)
```

html:table

```
html:table(*width, *height, *border, *cellpadding, *rules, *bgcolor, *class,  
*style, html)
```

html:tr

```
html:tr(*width, *height, html)
```

html:td

```
html:td(text, *width, *height, *colspan, *bgcolor, *align, *valign, html)
```

A note that needs to be made about `td` is that the *text* argument is mandatory, which means that it needs to always be present, even if a HTML nested element is used. The following example shows the correct usage for a table cell containing just an image:

```
...  
table(  
  ...  
  tr(  
    td("", {[el|html|img]}(src="image.jpg"))  
  )  
  ...  
)  
...
```

or

```
...  
<table>  
  ...  
  <tr>  
    <td text="">  
        
    </td>  
  </tr>  
  ...  
</table>  
...
```

html:th

`html:th(text, *width, *height, *colspan, html)`

See also: `td`

html:h1

`html:h1(text, html)`

html:h2

`html:h2(text, html)`

html:h3

`html:h3(text, html)`

html:h4

`html:h4(text, html)`

html:h5

`html:h5(text, html)`

html:h6

`html:h6(text, html)`

html:ul

`html:ul(html)`

html:pre

`html:pre(text, html)`

html:br

```
html:br()
```

html:li

```
html:li()
```

html:a

```
html:a(text, *href, *style, html)
```

html:anchor

```
html:anchor(name)
```

Anchor is used to define a HTML anchor since the a element does not support this functionality. It effectively returns `` on the ***html*** channel.

html:img

```
html:img(src, *border)
```

html:text

```
html:text(text)
```

Returns the value of the *text* argument on the ***html*** channel.

Forms Library

Concepts

Files: forms.k, forms.xml

The forms library can be used to build and gather information from GUI forms. The current implementation uses the Java Swing library.

Component ID

Each component whose state is user-modifiable (such as button, textField, etc.) must have an ID, which can be used to retrieve the state of the component when using the form element.

Component Layout

The layout of components is done using `hbox` and `vbox`.

Component Alignment

Components support alignment using the **halign* and **valign* arguments. **Halign* and **valign* have values between 0.0 and 1.0, with 0.0 representing left/top alignment, and 1.0 representing right/bottom alignment. By default, both **halign* and **valign* are set to 0.5 (centered).

Form Elements

form:form

```
form:form(id, title, waiton)
```

This element represents the main form element. It creates a window based on the specification received in . . . , and displays it. It will then wait for a control with the **id** that matches the *waiton* value, and terminates returning a map pairing control **ids** and their values. The layout of the controls in the window is performed by recursive nesting of vertical and horizontal containers (**vbox** and **hbox** respectively).

form:hbox

```
form:hbox(*homogeneous, . . .)
```

Represents a horizontal container. All controls specified by . . . are laid out one after another horizontally. By default, all controls are assigned an area of equal height, but the width varies according to the natural dimensions of the control (the total width is proportionally divided between the components according to their size). The **homogenous* argument can be used to force all cells to have an equal width. The various possibilities are illustrated in Figure 1, Figure 2, and Figure 3.



Figure 1: A non-homogenous hbox



Figure 2: A scaled non-homogenous hbox



Figure 3: A homogenous hbox

form:vbox

```
form:vbox(*homogenous, . . .)
```

A **vbox** is similar to a **hbox** but with the sub-components laid vertically.

form:label

```
form:label(text, *halign, *align)
```

Represents a text label. The text content is described by the *text* argument.

See also: Component Alignment

form:button

```
form:button(id, caption, *halign, *valign)
```

Represents a push-button. The text used for the label is specified with the *caption* argument. The ID is specified with the *id* argument.

See also: Component Alignment

form:checkBox

```
form:checkBox(id, *caption, *checked, *halign, *valign)
```

Constructs a check-box, whose initial state can be set using the **checked* argument. By default, the check-box is not checked. The label of the checkbox is specified using the *caption* argument.

See also: Component Alignment

form:radioBox

```
form:radioBox(id, caption, ...)
```

Allows the construction of a radio-button group. The group will have a titled border with the text defined by the *caption* argument. `RadioBox` expects radio button definitions on the default channel (...).

form:radioButton

```
form:radioButton(id, caption, *selected, *halign, *valign)
```

Defines a radio button, with the text set by the *caption* argument. By default, the first button in a `radioBox` is selected, unless the **selected* argument is used on a different radio button.

See also: Component Alignment

form:textField

```
form:textField(id, *columns, *halign, *valign)
```

Describes a text field, used for text input. A text field has only one line of text. The number of columns of the text field is indicated using the **columns* argument.

See also: Component Alignment

form:passwordField

```
form:passwordField(id, *columns, *halign, *valign)
```

A `passwordField` is similar to a `textField`, with the difference that the characters entered are visually represented by asterisks (*).

See also: `textField`

form:comboBox

```
form:comboBox(id, *halign, *valign, ...)
```

Constructs a combo-box, which can be used to select from multiple items in a drop-down list. The items are specified as `...`, using `comboBoxItem`.

See also: Component Alignment

form:comboBoxItem

```
form:comboBoxItem(*text, *selected)
```

Specifies a `comboBox` item. The text of the item is indicated by the `text` argument. By default, the first item in a `comboBox` is selected, unless the `*selected` argument is used on a different item.

form:HSeparator

```
form:HSeparator()
```

Constructs a horizontal separator component.

form:VSeparator

```
form:VSeparator()
```

Constructs a vertical separator component.

form:filler

```
form:filler(*width, *height)
```

Defines a filler component. A filler component serves no functional purpose, but can be used to influence the layout of a form by providing a visually invisible component of the specified `*width` and `*height` (in pixels).

form:messageDialog

```
form:messageDialog(message, title)
```

Used to display a popup window displaying the specified `message` and having the specified `title`. The element completes when the popup message is closed by the user, either by using the window controls or the **OK** button.

Restart Library

The Restart Log Library provides fault tolerance in a style similar to that of Condor DAGMan. The execution of certain operations is recorded into a log file on the disk. In case of a failure the execution can be resumed using the information saved in the log file. The operations that previously completed successfully will not be re-executed. The library defines two elements: `restartLog` and `logged`.

This mechanism offers no guarantees of semantic consistency after a restart if the control flow is influenced by factors that change between the original execution and the resumed execution. It is generally safe to use this mechanism with `for` and `parallelFor` if the iteration values are fixed. Additionally, return values of `logged` elements are not recorded. Consequently a resumed `logged` element will not return anything.

Usage example:

```
-----
import("sys.k")
import("task.k")
import("rlog.k")

logged(
  transfer(srcfile="a.txt", desthost="host.example.org", provider="gt2")
)

parallel(
  logged(
    execute(executable="/bin/cat", arguments="a.txt", stdout="b.txt",
      host="host.example.org", provider="gt2")
  )
  logged(
    execute(executable="/bin/cat", arguments="a.txt", stdout="c.txt",
      host="host.example.org", provider="gt2")
  )
)

parallelFor(out, list("b.txt", "c.txt"))
  logged(
    transfer(srcfile=out, srchost="host.example.org", provider="gt2")
  )
)
-----
```

Resuming:

```
-----
cog-workflow workflow.k -rlog:resume=workflow.0.rlog
-----
```

rlog:restartLog

`rlog:restartLog(*resume, *name, restartlog)`

`RestartLog` performs the following functions:

1. Opens a log file. The prefix of the log file name is taken from the **name* argument or, if the **name* argument is missing, from the file name of the current script being executed. The actual file name is obtained by appending a dot character ("."), a unique numeric identifier, and the ".rlog" extension. `RestartLog` will attempt to successively use increasing numeric identifiers, starting from 0 (zero). If a log file with that identifier already exists or if an exclusive lock on the file cannot be obtained, the next number is used. An exclusive lock is acquired on the log file such that other processes will not attempt to use the same file.
1. Accepts arguments on the **restartlog** channel and writes them to the log file. After writing each value, the file buffers are immediately flushed to the disk using the `FileDescriptor.sync()` ([http://java.sun.com/j2se/1.4.2/docs/api/java/io/FileDescriptor.html#sync\(\)](http://java.sun.com/j2se/1.4.2/docs/api/java/io/FileDescriptor.html#sync())) method.
1. In the case of a restart it also parses a previous log and builds the necessary data in a way that the

logged elements can use. If during a restart `restartLog` cannot acquire an exclusive lock on the log file, it will not attempt to use another log file, but fail instead.

1. Upon successful completion, it closes and deletes the log file.

Restarts can be triggered in two ways:

1. Using the `*resume` argument with the full file name of a log file
1. By specifying the `-rlog:resume=<logname>` command line argument to the script (not to the interpreter).

rlog:logged

`rlog:logged()`

Logged elements execute their child elements and, upon termination, they return, on the **`restartlog`** channel an identifier that uniquely identifies a logged element within a given execution, and a thread id, which is used to differentiate between concurrent runs of the same element. The same (element id; thread id) pair can occur multiple times in a log file, but it only reflects successive executions of an element within the same thread.

If a restart is in effect, logged elements will analyze data parsed from the log, and if there exists an entry with the same (element id; thread id) pair which has a count larger than 0, the count will be decreased and logged will complete without executing its arguments/child elements. It will consequently not return any values whatsoever, not even values that were returned by its child elements in a previous run. It is therefore recommended that logged only wrap elements that do not return values.

Service

Karajan features a service which can be used to execute parts of a script remotely. Some of the features of the service are listed below:

Security

The service and client use GSI security enabling strong authentication and encryption of data

Configurable communication channels

The Karajan networking infrastructure consists of configurable communication sub-systems, which separate the high level messaging protocols from the low level implementation details. The current implementation uses SSL sockets over TCP/IP, and can be configured to use persistent connections, or callbacks, or polling (or a combination of them), and the configurations can be different for each host or for a specific domain. This provides firewall transparency when needed, yet can minimize resource consumption and maximize performance when firewalls are not involved.

Semantic transparency

Remote execution can be seamlessly integrated with local execution, while preserving most of the system semantics, such as return values, definitions, etc.

Using the Service

The Karajan service can be accessed from Karajan scripts using the `remote` element.

Shared or Personal

The Karajan service can be used in two modes: *personal* or *shared*. In *personal* mode, the service runs under a user's credential, loaded from a proxy certificate. It is therefore necessary that a valid proxy exists before the service is started in *personal* mode. In this mode, all libraries are available for use without restrictions, but the connections are limited to clients using the same credential as the one used for starting the service.

In *shared* mode, the service requires a host credential. In this mode, connections can be initiated by multiple users, and authorization is performed based on a grid-map file. Access to certain libraries or functions that could be used to access files, resources, or other information belonging to other users using the service are restricted. Any attempt to execute such functions will result in the execution failing. The following list enumerates libraries and elements whose use is restricted in shared mode:

- The Java Library
- Elements not in the Task Library which can be used to access local files: `file:contains`, `file:read`, `outputStream`, `closeStream`, `checkpoint`, `file:execute`, and `file:write`.

In addition, the Task Library requires a special local provider, which can use operating system services to execute local tasks under non-shared privilege, achieved through grid-mapfile mapping. The service will refuse to run in *shared* mode if the normal local provider is detected. Details about building and configuring the secure local provider are available In the Service section.

When running in *shared* mode the service should be started under a non-privileged account. **Please do not run the service from the root account or any other administrative/privileged account!**

Limiting Access to Resources in *Shared* Mode

Using the service in *shared* mode has a number of security requirements:

- Access to resources should be restricted based on user identities.
- A user must not be able to access any other user's data/resources. Consequently, this requirement is divided into:
 1. Restricting access to the shared environment, such that privilege escalation cannot be done in order to override any of the security measures in place
 2. Delegating certain required privileged operations (such as execution and file access) to local security domains (running privileged operations under specific user accounts)

Access restriction is achieved using a "all-or-nothing" access control list: the grid-mapfile. A given identity (materialized in a GSI/X509 certificate) is only able to use the service if there exists an entry for that identity in the grid-mapfile.

Shared environment access restrictions is done in two ways: ^[1]

1. Execution of Karajan elements that could be used for escalating privileges or accessing other users' data/resources is prohibited. Attempts to use these will result in an error instead. For example, allowing arbitrary Java method invocations inside the shared interpreter environment (the service JVM) is prohibited. Similarly, elements that can be used to access files belonging to the account under which the shared interpreter is running is also prohibited, and their use will result in an error.
2. Instantiation of arbitrary Java objects, regardless of the means, is restricted to known "safe" object types. A subset of the problem is presented below:

The Karajan service allows passing various arguments from the client to the service by means of serialization and de-serialization. However this process does not guarantee the invariance of all objects when copied to a different resource. It is possible, that a certain object exists within the libraries of both the client and the service, which could, through the simple process of de-serialization, be initialized with privileged data,

which can later be retrieved through other means. For example, let us consider the following Java class:

```
public class Foo implements Serializable {
    private File file;
    private transient String contents;

    public Foo(File file) {
        this.file = file;
        this.contents = read contents of file
    }

    public String toString() {
        return contents;
    }
}
```

The transfer of an instance of `Foo` from the client to the service, by means of serialization/de-serialization will not preserve the local meaning of the file attribute. Instead, the mere instantiation of `Foo` combined with a seemingly unprivileged operation (`print("foo is {foo}")`) could allow access to arbitrary files in the service shared environment.

Another possible scenario is objects whose constructors implement side effects that execute privileged operations. Another `Foo`, should illustrate this:

```
public class Foo implements Serializable {
    public Foo() {
        something that amounts to "rm -rf /"
    }
}
```

It is clear that de-serializing an instance of the latter `Foo` in the service environment is undesirable, to say the least. Surely, both `Foo` versions are somewhat extreme, and somewhat unlikely to exist in the libraries of both the client and the service, but there is no certainty, and auditing all classes in the libraries would be very tedious if at all possible. The chosen solution to the problem is to have a set of allowed classes/packages which can be safely migrated between client and server, while excluding anything that is not explicitly in the set. A configuration file (`etc/karajan-restricted-classes.properties`) defines the de-serialization policies. The file allows two types of entries:

`package.allow`

Define a package which is allowed. All classes in the specified package and all its sub-packages are considered safe to be de-serialized.

`class.disallow`

Explicitly disallow a class from being de-serialized, even if it is contained in an allowed package.

Communication Layer Configuration

The communication layer supports three basic modes of operation, described below:

Persistent

In this mode, connections are kept open for as long as needed. All client→server and server→client communications will re-use the same connection, even if multiple requests are sent concurrently from the client. It is also possible to keep connections open after all current communications have terminated, in the event that future communications will be needed. This mode can be safely used if the client is behind a firewall, but it may keep resources allocated unnecessarily if no communication happens between the server and the client.

Callback

The callback mode can be used to instruct the service to initiate a connection to the client if the service needs to send any messages to the client. This type of configuration will most likely not work if a client is behind a firewall.

Polling

Polling is somewhat similar to persistent connections in that it will safely work if the client is behind a firewall. It instructs the client to disconnect idle connections, but re-establish them at specific intervals. It is therefore more conservative in terms of resource usage, but it may cause unwanted delays.

These modes of operation can be combined. For example, a combination between using callbacks and polling will ensure that the system will work whether the client is behind a firewall or not, but it will provide better performance in the best case scenario (no firewall).

The communication layer is conservative, in the sense that before a connection is initiated, it will try to search for any other existing connection that can be used to transmit a certain bit of information, irrespective of the initiator of the connection, and only if such a search fails, will it consider establishing a new connection.

The configuration can be changed by editing `etc/remote.properties`. The file contains a set of pairs of domain expressions and connection properties. The entries are considered in the reverse order from that in which they appear in the file. In other words, the first entry will be considered last, only if no other matching entry can be found. The domain expression is a regular expression which is matched against the domain name of the service host. The connection properties are a set of comma separated options. These options are described in the following table:

keepalive(timeout)

Indicates that the connection is to be kept alive. The optional timeout indicates that the connection should be also kept alive for the specified number of seconds, even if no actual data is sent through.

reconnect

This option instructs the system to re-initiate a connection in the event that a connection loss occurs. By default, failed connections will not be re-established.

callback

Instructs services to connect to the client if no existing connection can be used for sending data. A client will have started a local service before the first connection with the service for which a callback configuration exists is established.

poll(interval)

Instructs the client to poll the service for updates. The service will buffer all data that it needs to send to the client, and commit the buffered data when the client initiates a polling run. The interval indicates, in seconds, the interval at which the client should initiate the polls.

An example configuration is listed below:

```
-----
#default to persistent connections
#this one is reached if no other entry matches
"." keepalive(120), reconnect
.
#callbacks can safely be used within our own domain
".*mydomain.com" callback
.
#there's a firewall between mydomain.com and otherdomain.com, so
#we poll every 2 minutes
".*otherdomain.com" poll(120)
.
#for the sake of this example, a combination of callback and polling
#if callbacks don't work, polling will pick things up
".*thirddomain.org" callback, poll(120)
-----
```

The Secure (Grid-Mapped) Local Provider

Pre-built Packages

Binary packages for the secure local provider are available in the following formats: [.tar.gz (http://www.cogkit.org/release/4_1_4/cog-4_1_4-provider-slocal-bin.tar.gz)] [.zip (http://www.cogkit.org/release/4_1_4/cog-4_1_4-provider-slocal-bin.zip)]

Simply unpack either one of them in the `cog-4_1_4` directory, overwriting any existing files.

Building From Source

In order to build the secure local provider, the following steps must be taken:

- If a previous build was performed, run `ant distclean`
- Edit the abstractions dependency file in `cog/modules/abstraction/dependencies.xml`
- Change the dependency on the `provider-local` to `provider-slocal`:

```
<ant antfile="${main.buildfile}" target="dep">
  <property name="module" value="provider-slocal"/>
</ant>
```

- Re-compile with `ant dist`

At this time, the secure local provider only implements job execution (`execute`). File operations are not yet implemented.

The secure local provider uses a gridmap file, which describes a mapping of GSS distinguished names to local user accounts. For details about gridmap files, take a look at [Specifying Identity Mapping Information](http://globus.org/toolkit/docs/4.0/admin/docbook/ch05.html#id2516656) (<http://globus.org/toolkit/docs/4.0/admin/docbook/ch05.html#id2516656>)

The actual mapping is done using `sudo` and a simple job wrapper. `Sudo` must be configured to allow the execution of the wrapper under target user accounts without a password. An example `sudo` configuration is shown below:

```
<shared_username> ALL=(<target_usernames_list>) NOPASSWD: <cog_path>/libexec/job-wrapper
```

Where `<shared_username>` is the username of the account under which the shared service is running, `<target_usernames_list>` is a comma separated list of user-names which the provider can map to (these must also be correctly configured in the gridmap file), and `<cog_path>` is the absolute path to the CoG/Karajan installation.

If the target user account is the same as the shared user account, the secure local provider can be configured to bypass `sudo` (which it does by default).

The provider configuration file (`etc/provider-slocal.properties`), supports the following properties:

`grid.mapfile`

Indicates the location of the gridmap file. The default is `/etc/grid-security/grid-mapfile`.

`sudo`

Points to the location of `sudo`.

`nosudo`

If the target user is the same as the user under which the shared service is running, then the value of this property will be used as the path to a program used to start the wrapper.

`job.wrapper`

The location of the wrapper. It defaults to `<cog_path>/libexec/job-wrapper`, therefore it should not be necessary to specify this property unless a non-standard configuration is used.

An example configuration file is shown below:

```
-----
grid.mapfile=/etc/grid-security/grid-mapfile
sudo=/usr/bin/sudo
nosudo=/bin/bash
job.wrapper=/home/karajan/cog/libexec/job-wrapper
-----
```

Embedding Karajan into Java

This document provides details on the two modes through which Karajan scripts can be used from Java code

The first possibility is to use the Karajan parser to build an internal representation of the code, and execute it. Example:

```
-----
try {
    ElementTree tree = Loader.loadFromString("include(\"sys.k\"), print(\"Hello world!\");");
    ExecutionContext ec = new ExecutionContext(tree);
    ec.start();
    ec.waitFor();
}
catch (Exception e) {
    e.printStackTrace();
}
-----
```

An alternative, seemingly easier way to do the same is:

```
-----
KarajanWorkflow workflow = new KarajanWorkflow();
workflow.setSpecification("...");
workflow.start();
workflow.waitFor();
if (workflow.isFailed()) {
    System.err.println("Failed:");
    workflow.getFailure().printStackTrace();
}
-----
```

The other possibility is to build an `ElementTree` without parsing:

```

try {
    ElementTree tree = new ElementTree();

    Sequential s = new Sequential();
    Print p1 = new Print();
    p1.setProperty("message", "Hello");
    p1.setProperty("nl", Boolean.FALSE);
    Print p2 = new Print();
    p2.setProperty("message", " world!");
    s.addElement(p1);
    s.addElement(p2);
    tree.setRoot(s);

    ExecutionContext ec = new ExecutionContext(tree);
    ec.start();
    ec.waitFor();
}
catch (Exception e) {
    e.printStackTrace()
}

```

This would be roughly equivalent to:

```

sequential(
  print(message = "Hello", nl = false)
  print(message = " world!")
)

```

It is worth noting that it is not necessary to import the System Library because there is no need to resolve names of elements to implementations, since the implementations are specified directly.

More details about the mapping of element names to implementations can be found by consulting the actual library definitions found in the resource (<http://svn.sourceforge.net/viewvc/cogkit/trunk/current/src/cog/modules/karajan/resources>) directory. Listed below are a few common libraries with links to their sources (from which the mapping can be inferred):

- `sys-common.xml`
(<http://svn.sourceforge.net/viewvc/cogkit/trunk/current/src/cog/modules/karajan/resources/sys-common..>)
- `sys.xml`
(<http://svn.sourceforge.net/viewvc/cogkit/trunk/current/src/cog/modules/karajan/resources/sys.xml?view=>)
- `task.xml`
(<http://svn.sourceforge.net/viewvc/cogkit/trunk/current/src/cog/modules/karajan/resources/tsk.xml?view=>)

Notes

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

References

All CoG Kit references can be found [here
(http://wiki.mcs.anl.gov/gregor/index.php/Gregor_von_Laszewski#PUBLICATIONS_AND_PRESENTATIO
]

las06karajan
(http://wiki.mcs.anl.gov/gregor/index.php/Gregor_von_Laszewski#las06karajan)

las05workflow-jgc
(http://wiki.mcs.anl.gov/gregor/index.php/Gregor_von_Laszewski#las05workflow-jgc)

Java CoG Kit Workflow in to be
Mihael Hategan, and Deepti Kodel
Argonne IL, 60430, USA gregor@

Java CoG Kit Workflow Concept
Hategan, in Journal of Grid Comput
<http://www.mcs.anl.gov/~gregor/p>;
Electronic Publication Link, Jan. 2
<http://dx.doi.org/10.1007/s10723-1>

Retrieved from

["http://wiki.cogkit.org/index.php/Java_CoG_Kit_Karajan_Workflow_Reference_Manual_4.1.4"](http://wiki.cogkit.org/index.php/Java_CoG_Kit_Karajan_Workflow_Reference_Manual_4.1.4)

- This page was last modified 15:09, 28 August 2006.
- This page has been accessed 161 times.
- Privacy policy
- About Java CoG Kit
- Disclaimers