

Chapter 3

Workflow description language

The workflow description language is based off the World Wide Web Consortium (W3C)'s candidate recommendation for the Web Ontology Language (OWL). OWL is designed to support ontologies distributed on the Web, using RDF.

The workflow description language is the description of all the constructs necessary to describe and execute a workflow. This includes class definitions, class instances, and modules.

3.1 Workflow metadata

why is workflow metadata?

why do we use it- what issue does it solve?

3.1.1 Namespace declaration

Namespaces are defined in the document's root element

what are namespaces for?

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#">
```

what does this definition do?

3.1.2 Version information

This tag specifies to the Pipeline application the version of the LONI Pipeline workflow syntax which is employed in the document.

```
<owl:Ontology>
  <owl:versionInfo><![CDATA[20050223]]></owl:versionInfo>
</owl:Ontology>
```

Currently the version string is “20050412”.

3.2 Class definitions

why are class definitions important? Class definitions allow us to easily reference all the properties and restrictions of instances of the class, without having to repeat these properties and restrictions.

At the beginning of every module file is a set of class definitions. These class definitions define the various classes of objects to be manipulated, e.g. File, Function, Module. The class “File” can be defined as

```
<rdfs:Class rdf:ID="File" />
```

Class definitions can denote inheritance relations. For example, SystemExecutable is a subclass of both File and FunctionSingleton, and is written as

```
<rdfs:Class rdf:ID="ExecutableFile">
  <rdfs:SubClassOf rdf:resource="File" />
  <rdfs:SubClassOf rdf:resource="FunctionSingleton" />
</rdfs:Class>
```

more to intro?

More information **about?** can be found at

<http://www.w3.org/TR/owl-guide/>

what does being a subclass mean?

3.2.1 File dependencies

File dependencies are another example of the usefulness of definitions. For example, Analyze image files have two separate parts- an image header file, and an image data file. These two have the same path, except with file extension “img” for the data file, and “hdr” for the header file. To describe this, we write (explain that humans have a logical concept of them as a single file, but we need to support programs that take one or the other as input. for example, AIR takes the data (.img) files, while ITK takes the header (.hdr files).

```
<!-- Define a class called "AnalyzeImageHeaderFile".
  One of this class' attributes is that it has an extension "hdr" -->
<rdfs:Class rdf:ID="AnalyzeImageHeaderFile" extension="hdr" />
<!-- Define a class called "AnalyzeImageDataFile".
  One of this class' attributes is that it has an extension "img" -->
<rdfs:Class rdf:ID="AnalyzeImageDataFile" extension="img" />
```

```

<!-- Define a class called "AnalyzeImageFile". -->
<rdfs:Class rdf:ID="AnalyzeImageFile">
  <!-- An AnalyzeImageFile has a subpart which is
  an instance of AnalyzeImageHeaderFile.
  This extension has the same path name as the other parts,
  except for its extension -->
  <HasPart rdf:resource="AnalyzeImageHeaderFile">
    <replace type="extension"/>
  </HasPart>
  <!-- An AnalyzeImageFile has a subpart which is
  an instance of AnalyzeImageDataFile
  This extension has the same path name as the other parts,
  except for its extension -->
  <HasPart rdf:resource="AnalyzeImageDataFile">
    <replace type="extension"/>
  </HasPart>
</rdfs:Class>

```

With this definition, an user can specify an `AnalyzeImageFile`, which the application will know that both files are included. Users can also specify `AnalyzeImageDataFile` or `AnalyzeImageHeaderFile` individually, which the application will know that it is just that file with the particular file extension.

3.3 Resource definitions

Following RDF specifications, everything is a resource. Every resource can be uniquely identified by its ID, specified by its “`rdf:ID`” attribute. IDs are not meant to be human readable. **reword**

3.3.1 Resource references

In order to define relationships between resources, we need a manner of referring to any resource from any other resource. This is done by using the attribute “`rdf:resource`”.

what is the value of the `rdf:resource` attribute?

3.3.2 Labels

```
<rdfs:label locale="en"><![CDATA[this is a label]]></rdfs:label>
```

Every resource can have a set of labels. These labels are intended to be short, human readable names that identify the resource to a human. Notice that the content of the label is quoted **using XML’s CDATA** so that the content is not assumed to be XML content.

Finally, labels can be localized to support localization using the “`locale`” attribute. The above example defines the label to be English.

3.3.3 Comments

```
<rdfs:comment><![CDATA[this is a comment]]></rdfs:comment>
```

Every resource can have a set of comments. These comments are intended to provide (possibly verbose) explanations of the resources themselves. Like labels, comments can be localized.

3.3.4 Resource values

Resource values in the Pipeline description languages are all stored as strings. One example is

```
<Values>
  <Value>
    <string rdf:resource="stats_dir" />
    <string rdf:resource="subject_MRI" filter="base" />
    <string><![CDATA[_GWC_SVTstats.txt]]></string>
  </Value>
</Values>
```

what does this definition do?

These strings are constructed in a complicated manner, taking into account several factors:

String literals

String literal are literal strings. They can be defined as

```
<string><![CDATA[this is a string literal]]></string>
```

String references

String references are references to the string values of other resources, which allows users to not have to redefine the same value if not necessary. The definition

```
<string rdf:resource="stringLiteralVariable" />
```

can be interpreted to mean the value of this string is the value retrieved from the variable “stringLiteralVariable”.

describe/explain above example

Value filtering

When values are retrieved from a resource that is an instance of a file, it is possible to filter that value and extract only a part of the file url. Additionally, all values can be filtered by a regular expression. The definition

```
<string rdf:resource="fileVariable" filter="extension" />
```

specifies that the value of this string resource is the extension of the value of “fileVariable”. **describe/explain above example** Possible url component filters are

1. “root”- the file root; everything before the file extension
2. “file”- the file name; everything before the last file separator
3. “base”- the base file name; the file name minus the extension
4. “extension”- the file extension
5. “directory”- everything before the last file separator

Value grouping

Values retrieved from other resources usually are used one-to-one, with respect to grouping. That is, the number of values retrieved is the number of values for the string reference.

It is possible to treat the values retrieved in other groupings. If a string reference is specified to be many-to-one, then a collection of values is grouped together and treated as a single value. This can be defined as

```
<string rdf:resource="manyValuesVariable" grouping="ManyToOne" />
```

what does it mean to be grouped and treated as one?

Conversely, it is possible to split a single value into many values. The definition

```
<string rdf:resource="singleValueVariable"
      grouping="OneToMany" splitRegex="\n" />
```

specifies that the value from the variable “singleValueVariable” should be split, and it should be split using newline as a delimiter.

List binding

what is list binding? why is list binding useful?

values for a resource are bound to a list if the attribute `bindValueFromURL` exists and its value is set to true.

allows for resource values to be externalized

supports “abstract” sharable workflows. the receiving researcher just “plugs in” data (a list of input/output data)

There are many layers of abstraction. Here “abstract” refers to data abstraction

give an example of how to define “bindValueFromURL”

```
<string bindValueFromURL="true"><![CDATA[pipeline://localhost//example.list]]></string>
```

which tags can have this attribute?

also an example of bindValueFromList(?)

3.4 Class instances

Once the classes have been defined, class instances are defined. In OWL, all objects are resources, and provided an ID, to which it can be referred.

```
<AnalyzeImageFile rdf:ID="AnalyzeImageFileInstance1" />
```

defines an `AnalyzeImageFile` having the ID “`AnalyzeImageFileInstance1`”.

This definition automatically assigns all the properties and restrictions of the class to the instance.

3.5 Functions

what is the motivation?

single definition, for same modules (give example)

abstract vs. concrete

how do composite modules reference other modules?

Function definitions provide information about the usage of the function, such as location and parameters. Definitions do not contain information about any particular usage or invocation of the function.

3.5.1 Function labels

Functions, like any other resources, can have labels. Function labels provide a short human readable (non-unique) identifier of the function.

3.5.2 Function comments

Functions can have comments as well. Function comments provide a human readable description of the functionality of the function.

3.5.3 Parameters

Parameters can have associated attributes. Some of the available attributes are “phantom”, “grouping”, and “connectionType”.

how to define? give example

default values. why necessary? how to define?

value range enumeration. how to define?

parameter restrictions - how to define? - why useful? - in singletons - in composite modules

Prefix and suffix

Phantom parameters

Phantom parameters are parameters that do not output a value. If a prefix and/or suffix is defined for the parameter, the prefix/suffix will be output. This

is useful when programs take flags with no variable values. One such example is the “force” flag of the Unix “rm” program.

Phantom parameters without any prefix/suffix defined can also serve another purpose- they can enforce serial execution between two modules when both are defined as phantom, and one’s input parameter is connected to another’s output.

Grouping

Similar to value grouping (section 3.3.4), a parameter can be defined to have a grouping other than one-to-one.

One example of a necessary many-to-one grouping for a parameter is the “sum” function. In this function, the first parameter is a number. The second parameter is one or more numbers.

what happens when grouping is not employed?

3.6 Composite functions

This composition provides an abstraction of the composition as a single task.

Function references

- labels

- comments

Function references can add restrictions, with respect to which function can be used, how it can be used, etc

Function references need to reassign the name of the function

Connection type

connection type is a misnomer

input/output file

port type?

how to define? why useful?

3.6.1 Rescoping parameters

The internals of a function such as the parameters of its child functions are not accessible to external references. In order to expose these parameters, these parameters from child functions can be rescoped. This can be done by placing an element, such as

```
<Parameter rdf:ID="rescopedParameter">
  <owl:allValuesFrom rdf:resource="childFunction#parameter" />
</Parameter>
```

into the definition of the parent function. In this definition, the parameter “parameter” from child function “childFunction” is exposed as a parameter of the parent function, with ID “rescopedParameter”. **describe how this changes the interface**

allow for specification of additional restrictions, **such as?**

why useful? (give examples)

why not do the same as in programming languages, having global and local scope? (or if already the same, explain how so)

3.6.2 Connections

(move to before connection type?)

Connections can be defined between modules to specify that the output of one is the input of another. Connections can be defined only between functions who are children of the same composite function. This definition can be written as

```
<ParameterConnection>
  <source rdf:resource="sourceChildFunction#sourceParameter" />
  <sink rdf:resource="sinkChildFunction#sinkParameter" />
</ParameterConnection>
```

In the definition, the output parameter “sourceParameter” of the source child function “sourceChildFunction” is connected to the input parameter “sinkParameter” of the sink child function “sinkChildFunction”. This definition is placed as a child element of the parent function definition.

connections between parameters of composite modules

modules can be nested indefinitely. connections between modules are context dependent

additional restrictions can be placed on the connection **such as?**

3.6.3 Control flow composite functions

(condition of data availability vs condition of world state)

(make clear that control flow can be guild on top of data flow. possible for the other way around?)

other types of compositions (serial, parallel, branch, iteration)

how does control flow differ from data flow?

what are necessary constructs? parameters? connections?

Branch functions

for the function itself, need to define

1. the switch condition
2. inputs and outputs of the branch function. the set of these inputs and outputs is the union of all the (rescoped) inputs and outputs of each case

3. the cases to be run for possible values of the switch resource
 - for each function, need to define
 1. the value of the condition such that this branch is taken
 2. the single function that contains all the steps to be run
 3. the mapping of the branch function's inputs and outputs to the function's inputs and outputs. must be a subset of the branch function's defined inputs and outputs

```

<SwitchResource>
  <SubClassOf rdf:resource="Parameter" />
</SwitchResource>

<BranchFunction>
  <!-- additional exposed parameters go here -->
  <!-- ... -->

  <SwitchResource connectionType="input">
    <rdf:type>...</rdf:type>
  </SwitchResource>
  <Case>
    <SwitchValue>y</SwitchValue>
    <Assign>
      <!-- need to remap the parameters
            defined in branch function -->

      <!-- what happens if one does not
            match the parameter ref? -->

      <owl:allValuesFrom />
    </Assign>
  </Case>
</BranchFunction>

```

how does parameter(*singletonParameter*, *parameterReference*) hierarchy support this? **what is “this”**? probably need another parameter, such as *BranchFunctionParameter*

Iteration functions

need to define

1. the collection to iterate over

2. the function to be run
3. the connections between the function's outputs and inputs, if any

```
<IterationFunction>
  <!-- define the collection to iterate over -->

  <!-- define the parameter connection
        from the function's output to its input on each iteration -->
</IterationFunction>
```

```
<Union />
<Intersection />
```

```
<NumericRange>
  <Minimum />
  <Maximum />
  <Step />
</NumericRange>
```

```
<Enumeration>
  <li> ... </li>
</Enumeration>
```

3.7 Modules

Modules instantiate functions, but still provide a level of abstraction in that the data is external to the module, and that the same module can be bound to different variables.

Modules are the workflow components that describe the task to be completed. There are atomic and composite modules, which correspond to atomic and composite functions, respectively. Atomic modules provide information to perform exactly one task. Composite modules is a composition of other modules.

(show definition of function composition)

(show xml description, ie how to define?)

3.7.1 Labels

3.7.2 Comments

3.7.3 Arguments

Module arguments become the binding of resources to function parameters.

```
<Assign rdf:resource="parameterID">
  <owl:allValuesFrom rdf:resource="variableID" />
</Assign>
```

In the above example, the values of the variable having ID “variableID” is bound to the parameter having ID “parameterID”.

The values of the arguments are populated by variable assignments.

(need older version of module, where arguments nodes are defined using “Argument”)

arguments are unordered. the eventual composition of the command is taken from the order of the parameters

3.7.4 Argument values

argument values are taken from the values of...

3.8 Function location abstraction

1st step- allow multiple locations in the ontology 2nd step- allow users to specify locations in addition to those in the ontology. save the trouble of having to redefine

To prevent malicious users from breaking the system, ontology editing is restricted to a small set of users whose membership does not necessarily include the user.

why not just copy the entire definition into the workflow document? in a dev environment, a user may want to specify additional location, but remove it once deve/testing/qa is over and share the workflow with others. while they could just remove the additional from the copied, that does not resolve that there are now 2 definitions of the same.

why is function location abstraction useful?

differentiate between function location and function usage. what is the motivation?

allow users to specify one of many function locations

allow users to specify additional function locations not in ontology

```
ontologyDefinedFunction -> ontologyDefinedFunctionURLs
ontologyDefinedFunction -> ontologyDefinedFunctionUsage
```

```
userDefinedFunction -> allValuesFrom(ontologyDefinedFunction)
userDefinedFunction -> userDefinedFunctionURLs
```

```
module -> userDefinedFunction
```

how is each defined in figure ???

move to same section in execution?

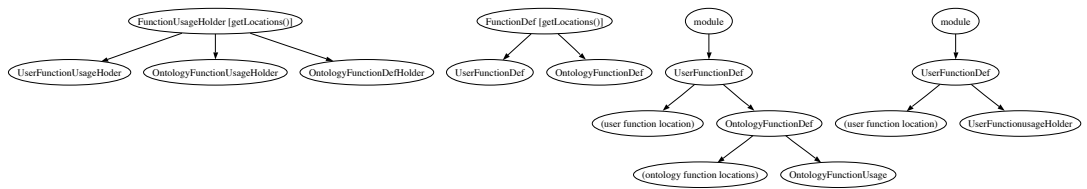


Figure 3.1: Multiple function location

3.9 Placeholder functions

what does the def look like? what does the def afford?

workflow design at a high, conceptual level

define using only restrictions

minimal commitment to any specific algorithm. allows user to specify restrictions on what could be place in the workflow component, without actually specifying the actual algorithm

useful for process validation and automatic algorithm selection

allow only those that satisfy the parameter restrictions to replace a placeholder

allow those that replace a placeholder to be removed (with all restrictions in place)

3.10 DTD

include somewhere, perhaps in appendix?