Feature: Workflow Components

Definitions

Workflow – a sequence of processes chained together that can be shared or modified by the user

Component – a process or activity which, given a set of inputs and parameters, produces some desired output. A component is also a set of configuration files and executables which collectively make up the software process included in a workflow.

Programs – one or more programs to be executed by the component

XML – the markup language used by components to communicate with one another.

XML Schema Definition – the component definitions (inputs, outputs, options, and meta-data)

Workflows Overview

Workflows

A workflow is a component-based process model that can be used to analyze, manipulate, or visualize data. Each component acts as a standalone program with its own inputs, options, and outputs. The inputs to each component can be data or files, and the output of each component is made available after the workflow has been run. A generic workflow might consist of the following steps.

1. ***import*** a tab-delimited file
2. ***analyze*** the file
3. ***visualize*** the results of the analysis component

Not all components are interchangeable, as each one has a set of required inputs and options. For instance, some components will require a specific type of file, like tab-delimited or csv. Every component may have a set of inputs and user-defined options. The constraints on inputs and options are defined in each component's XML Schema Definition (XSD) which we will discuss in greater detail, later.

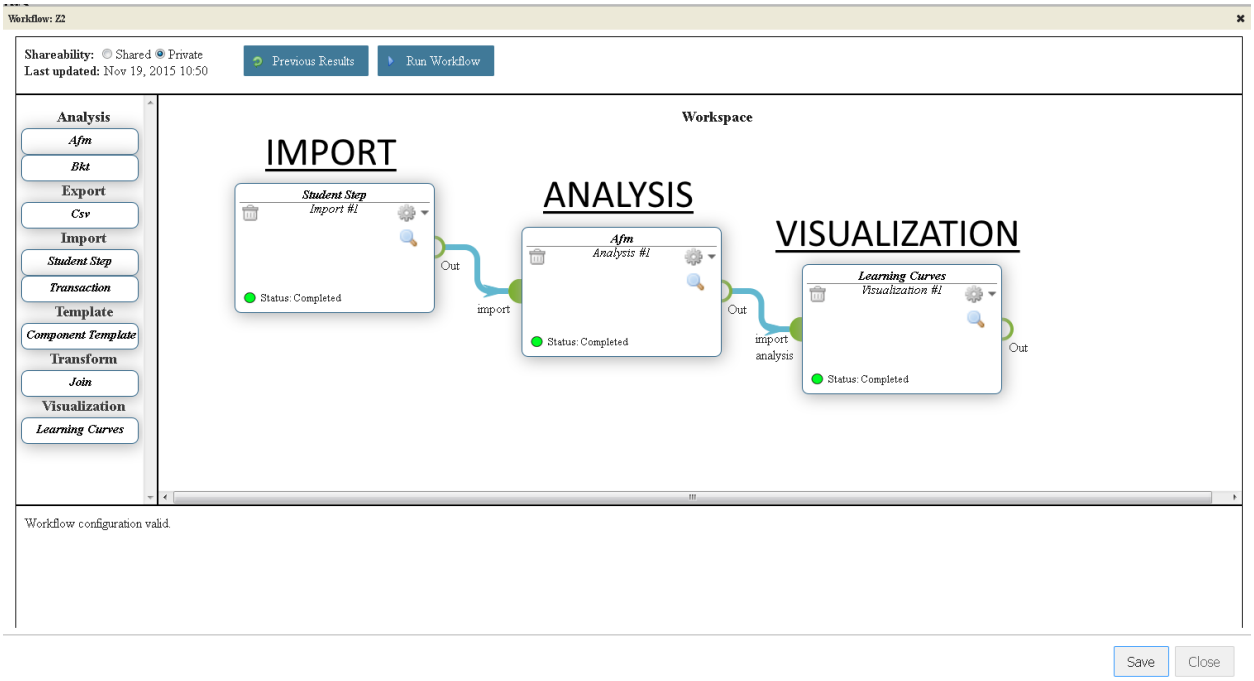


Figure 1 - Each component output in a workflow is a function of its input and user-defined options.

Components

A component is essentially a black box which receives data from other components and carries out an arbitrary task which results in the generation of new data. The newly generated data is then passed to the component's successor. Some common component types could be *analyses*, *imports*, *transformations,* and *visualizations*. However, these types only exist for the sake of organization and to group processes with similar inputs and

outputs, but we are not limited to any particular hierarchy, as a component is merely a generic stand-in for a function. Additional component types could include *generative functions*, *sampling functions*, *transformations*, and a host of other applications. Regardless of the component type, all components match the same pattern, i.e. they all have a set of *inputs*, *options*, and *outputs*. Because of this pattern, the process for creating a component can be broken down into two parts—the **XML schema definition** and the **run-time program**.

XML Schema Definition

The first part of a component is the XML Schema Definition (XSD) file. The XSD file defines the structure of a component.

Inputs - files or data (or both) obtained from the direct predecessor of a component.

Options - user-submitted files or data (or both) accessible from the user interface.

Outputs - files or data (or both) generated by the component and passed to its direct successors.

Basic Schema Definition

The skeleton XML schema for a workflow component describes its **inputs**, **options**, and **outputs**. These will be more thoroughly discussed in the section, The Component Schema. The sections there will describe the exact code you will need to create the desired effect. For now, simply glance over to familiarize yourself with the general concept.

The XML schema below defines the inputs, options, and outputs for the "Analysis - AFM" component. It also includes information about the component instance in the *<component>* element which is used by the software for identification, display purposes, and for passing messages between components.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:include schemaLocation="../../CommonSchemas/WorkflowsCommon.xsd" />

  <xs:complexType name="InputDefinition0">
    <xs:complexContent>
      <xs:extension base="InputContainer">
        <xs:sequence>
          <xs:element type="InFileList0" name="files" />
          <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="InputType">
    <xs:sequence>
      <xs:element name="input0" type="InputDefinition0" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="InputLabel">
    <xs:all>
      <xs:element name="input0" type="xs:string" default="text" minOccurs="0" />
    </xs:all>
  </xs:complexType>
```

```
  <xs:complexType name="InFileList0">
    <xs:choice>
      <xs:element ref="file" minOccurs="0" />
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="OutputDefinition0">
    <xs:complexContent>
      <xs:extension base="InputContainer">
        <xs:sequence>
          <xs:element type="OutFileList0" name="files" />
          <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="OutputType">
    <xs:sequence>
      <xs:element name="output0" type="OutputDefinition0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="OutFileList0">
    <xs:choice>
      <xs:element ref="student-step" />
    </xs:choice>
  </xs:complexType>


  <xs:complexType name="OptionsType">
    <xs:all>
      <xs:element type="FileInputHeader" name="model" id="Model" default="KC\s*(.*)\s*" />
      <xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />
    </xs:all>
  </xs:complexType>

  <xs:element name="component">
    <xs:complexType>
      <xs:all>
        <xs:element type="xs:integer" name="workflow_id" />
        <xs:element type="xs:string" name="component_id" />
        <xs:element type="xs:string" name="component_id_human" />
        <xs:element type="xs:string" name="component_name" />
        <xs:element type="xs:string" name="component_type" />
        <xs:element type="xs:double" name="left" />
        <xs:element type="xs:double" name="top" />
        <xs:element name="connections" minOccurs="0" maxOccurs="1" type="ConnectionType" />

        <xs:element name="inputs" type="InputType" minOccurs="0" />
        <xs:element name="inputLabels" type="InputLabel" minOccurs="0" />
        <xs:element name="outputs" type="OutputType" minOccurs="0" />
        <xs:element name="options" type="OptionsType" minOccurs="0" />

      </xs:all>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Run-time program

The second part of the component is a run-time program. A component is essentially a standalone process. It has no knowledge of other components in the workflow chain except through its input channels. It receives changes to its options by interfacing with the user. Once its inputs and options have been processed, it will execute and send its output to any direct successors.

To make developing components easier, a Java template and a common set of Java methods handle most of the work. New components can be created with just a few lines of code and a schema definition with little overhead. Components can run programs written in any language, i.e. C/C++, Java, Matlab, Perl, Python, Ruby, R, etc.

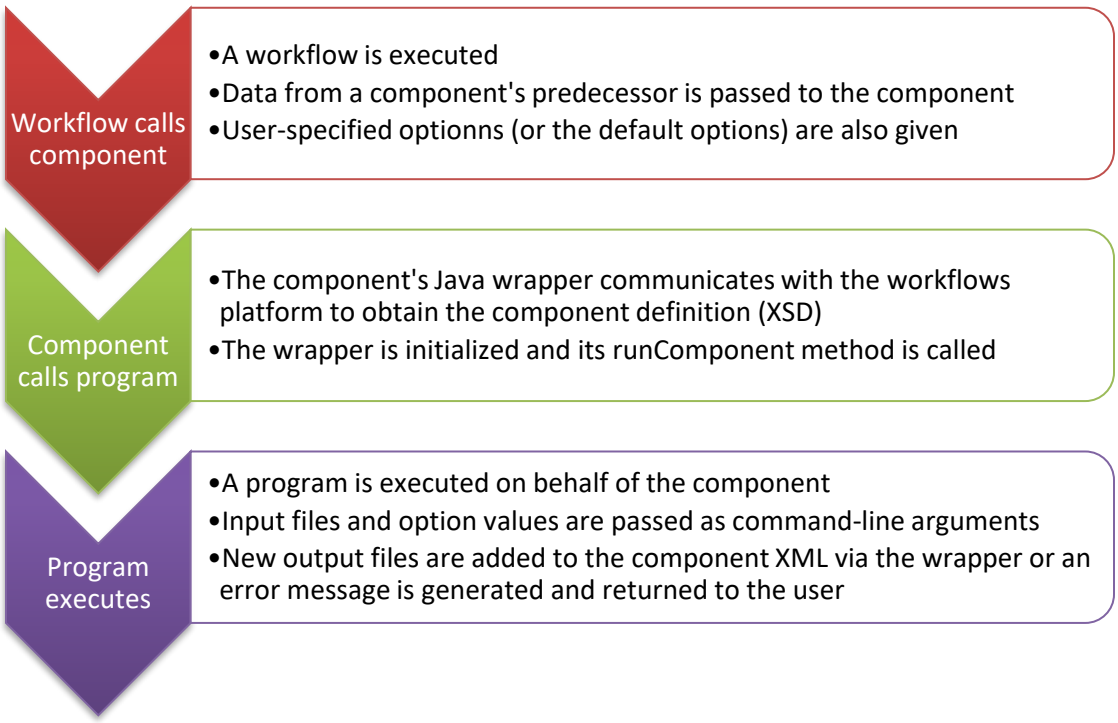


Figure 2 The component call stack.

Language and Library Support

If you unsure whether or not a language or library is supported or would like to request that one be added, then please inquire at datashop-help@lists.andrew.cmu.edu.

Components Overview

Checkout the WorkflowComponents repository with the following command.

```
git clone https://github.com/Learnsphere/WorkflowComponents WorkflowComponents
```

Dependencies

- Ant 1.9 or greater
- Java Enterprise Edition Software Development Kit (J2EE SDK)
- Eclipse, Cygwin, or a terminal / command-prompt

To access the components' source code, you will need to request access to the repository from datashop-help@lists.andrew.cmu.edu.

Organization

Programs used in components can be written in any language, but each component requires an **XML Schema Definition** (XSD) and a **Java wrapper**. The Java wrapper reduces the overhead of writing components by taking care of schema validation, pre- and post-conditions, argument passing, error handling, feedback, and program execution. It also facilitates communication between the workflows platform and components.

Each component is populated with an example in the "test" directory which allows it to be run as a standalone application, locally, if you have installed Apache Ant and the Java Development Kit. Most components will only require the Java EE SDK and a recent version of Ant. These can be acquired at http://www.oracle.com/technetwork/java/javaee/downloads/index.html and http://ant.apache.org/bindownload.cgi, respectively.

Any system with Java and Ant should be able to run the components. However, examples which use additional run-time programs, like AnalysisBkt, may require their programs be compiled on the system intended for use. Such instructions should be documented within the component, itself, in the README.md which exists in each component directory.

Component Organization

Directories

”source" – contains the Java wrapper source code
"dist" – contains the executable jar (standalone component)
"program" – contains any run-time programs
"schemas" – contains the XML Schema Definition (XSD) for this component
"test" – contains example data used for running the components locally
"unit_test" – contains the unit test classes (to be *implemented in the next iteration*)

Building and Running Components

Once you’ve obtained the "WorkflowComponents" repository from GitHub, you should now be able to run them as standalone applications with the test data. The preferred method for working with the projects is to create an Eclipse project. However, the simplest method is to simply use ant.

Build and Run with Ant

1. Each component is in a separate directory and has its own **build.xml** file.
2. In your command-line prompt, change to **C:\dev\WorkflowComponents\AnalysisAfm**
3. Enter "ant -p" into the command-line. Doing so will present you with a list of available ant commands for the project.

```
c:\dev\WorkflowComponentsTrunk\AnalysisAfm> ant -p

Buildfile: c:\dev\WorkflowComponentsTrunk\AnalysisAfm\build.xml
```

```
Java-based AnalysisAfm workflow component.

Main targets:

clean  -  clean up
compile - compile the Java wrapper
dist -    generate the jar from the Java wrapper
javadoc - create javadoc documentation
runComponent -  run the component
```

Now, we can simply type "ant runComponent" in the same directory to execute the standalone component using the example data provided in the "test/" directory. The component outputs an XML entity which describes all of its output. The XML is generated automatically using built-in methods which will be covered later in this document.

```
c:\dev\WorkflowComponentsTrunk\AnalysisAfm> ant runComponent
runComponent:
     [java] <output>
     [java]   <component_id>Component1</component_id>
     [java]   <component_id_human>Component #1</component_id_human>
     [java]   <component_type>Analysis</component_type>
     [java]   <component_name>AFM</component_name>
     [java]   <elapsed_seconds>1</elapsed_seconds>
     [java]   <model>Default</model>
     [java]   <files>
     [java]     <student-step>
     [java]       <file_path>C:/dev/WorkflowComponents/AnalysisAfm/test/Component1/Step-values-with-
predictions-Component_1--081215-091902-286.txt</file_path>
     [java]       <file_name>Step-values-with-predictions-Component_1--081215-091902-286.txt</file_name>
     [java]       <file_type>student-step</file_type>
     [java]       <label>Student Step</label>
     [java]     </student-step>
     [java]   </files>
     [java] </output>

BUILD SUCCESSFUL
Total time: 1 second
```

The lines prefixed with [java] show the structured output of the component—most of the elements are used to describe the component except for model and the student-step file (which are specific to the AnalysisAfm component).

In this example (AnalysisAfm, a pure Java component), the Ant build.properties file is not necessary. Later we discuss how this file is used by components written in other languages. For example, if running a component written in R, you will need to copy the build.properties.sample file into build.properties and edit the file to point to the location of Rscript on your machine.

Build and Run with Eclipse

1. Go to File -> Import -> General -> "Existing Projects into Workspace"
2. Choose any component directory from WorkflowComponents/<AnyComponent>
3. Click 'Finish'

4. In the Ant view (Windows -> Show View -> Ant), add the desired component's build.xml to your current buildfiles, e.g. <AnyComponent>/build.xml
5. Double click the ant task "runComponent". The component should produce example XML output if it is setup correctly.

- For debugging, you will want to add the jars in the directory WorkflowComponents/CommonLibraries to your build path.

Modifying and Rebuilding Components

To rebuild a component after modifying the source, simply use the ant command "**ant dist**". Rebuilding is also done automatically when running the target: **ant runComponent**. The following sections detail the steps needed to create a new component that can be added to the workflows platform—i.e., creating the component **XML Schema Definition** (XSD) and **integrating your code**.

Submitting Your Component

After reviewing and testing your code, it can be added to the Workflows platform and made available to all LearnSphere users. If you desire, we can also make your component available on one of our public or private GitHub repositories so that others may benefit from your work. Either send the entire component as a zipped file to us, or make it available via SVN, CVS, Git, or via the web.

When submitting your component, please make sure to only include a build.properties.sample file, not build.properties. The necessary build.properties files are typically platform- and server-specific and will be created as needed.

Automated Component Creation

We have written a Java program (and a script to run it) to facilitate creating new components. The script – runWCC.sh or runWCC.bat -- is available in the main directory of the WorkflowComponents GitHub repository. It is run as:

- sh ./runWCC.sh *components_dir properties_file*

The *components_dir* argument specifies the directory where the components can be found on your system. This is where the program will create your component and is also where it expects to find the Templates directory used to generate component files. The *properties_file* is a properties file you create to specify the information necessary to define your new component, e.g., the component name, type, language, options, etc. There are well-documented example property files for the supported language options – Java, R, Python and Jar – in the Templates directory. Please do not hesitate to contact us for assistance. *To ask for help or to submit a new component for review, please contact datashop-help@lists.andrew.cmu.edu.*

The Component Schema

Each component is defined by its own XML Schema Definition (XSD) located in "*<AnyComponent>*/schemas/". The XSD defines the component's expected **inputs**, **outputs,** and **options**.

Defining Inputs and Outputs

Components do not require input, but they all generate output. Both are defined similarly. **Inputs** are defined using the **InFileList** definitions. **Outputs** are defined using the **OutFileList** definitions. Generally, a component will pass its data in one or more files.

Let's look at an example, the Join component, which inputs two text files and generates a single tab-delimited file which contains the joined data files.

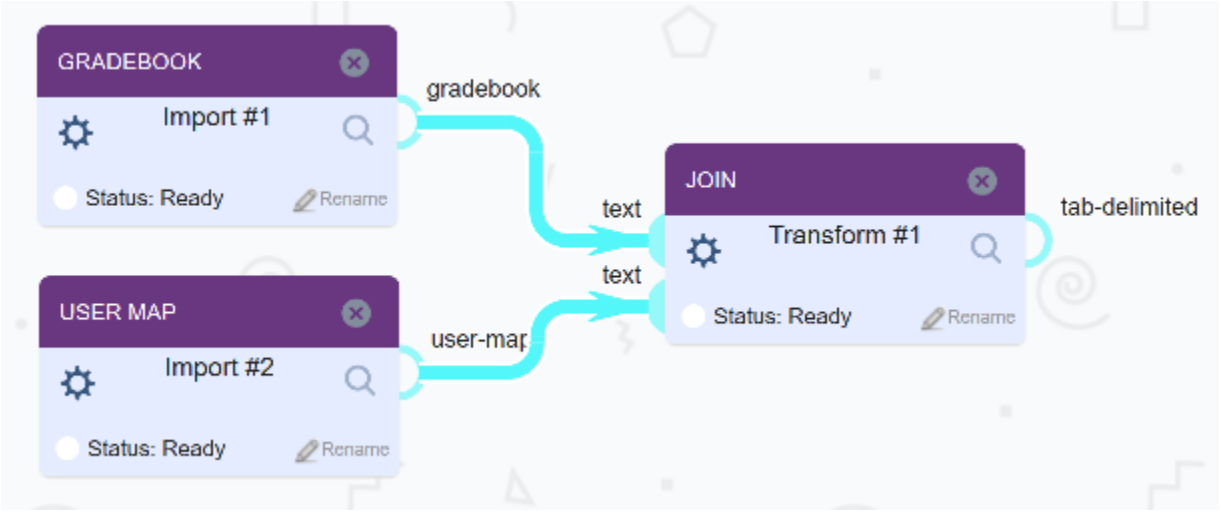


Figure 3 The Join component accepts two text inputs and outputs a tab-delimited file.

The Options panel for the Join component is also defined in the XSD. We will get into option definitions later.

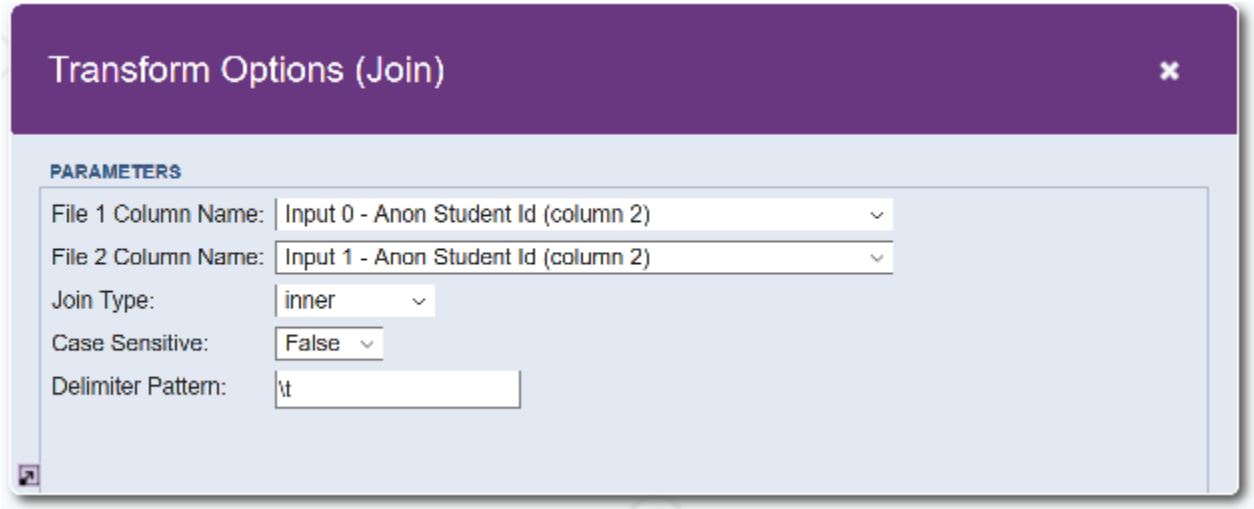


Figure 4 Example Join Options Panel

Let's look at how to define the inputs and outputs in its XSD. Note that the *student-step* file type is a descendant of the *text* file type. If you are uncertain which file type to use, then use the generic type *file*, as it matches any file type. See Appendix A – Existing File Types for more information on the file type hierarchy.

Inputs

A component can have 0 or more inputs. The Join component requires two *text* inputs. We use a numeric suffix to differentiate the input nodes, e.g. InFileList0, InFileList1, etc.

The **InputDefinitionX** blocks never change. You will need one for each input, e.g. InputDefinition0, InputDefinion1. The **InFileListX** blocks define the expected file type—text in this case. The **InputType** block is used to finalize the definition and the order of the inputs (if the text1 were listed first, it would be treated as the first input node).

Lastly, there is an optional element included below, **InputLabel**. It is used to suggest the type of file that the input should receive, and it parallels the **InputType** element. Note that there are no OutputLabel or OptionFileLabel elements since they are not needed. If no InputLabel is included, the UI will use the most generic file type, "file".

```
<xs:complexType name="InputDefinition0">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="InFileList0" name="files" />
            <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="InputDefinition1">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="InFileList1" name="files" />
           <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="InFileList0">
  <xs:choice>
    <xs:element ref="file"  minOccurs="0" />
  </xs:choice>
</xs:complexType>

<xs:complexType name="InFileList1">
  <xs:choice>
    <xs:element ref="file" minOccurs="0" />
  </xs:choice>
</xs:complexType>

<xs:complexType name="InputLabel">
  <xs:all>
    <xs:element name="input0" type="xs:string" default="text" minOccurs="0" />
    <xs:element name="input1" type="xs:string" default="text" minOccurs="0" />
  </xs:all>
</xs:complexType>
```

```
<xs:complexType name="InputType">
  <xs:sequence>
    <xs:element name="input0" type="InputDefinition0" minOccurs="0" />
    <xs:element name="input1" type="InputDefinition1" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Required vs. Optional Inputs

Required inputs will display an error to the user if the input file does not exist. *Optional inputs* will still allow a component to execute without any provided input, but if input is given, it will take advantage. To make an input file required, the InputType definition should have minOccurs="1", or simply set it to "0" to make it optional.

```
<xs:complexType name="InputType">
  <xs:sequence>
   <xs:element name="input0" type="InputDefinition0" minOccurs="1" />
  </xs:sequence>
</xs:complexType>
```

Ordered Inputs (one connection per input node)

Examine the following example component which has two input nodes and one output node. Each input node allows one incoming connection (one file per input node), as defined in the component's XSD. Since the order of files in a Join transformation is important, we use more than one input node to maintain the ordering.

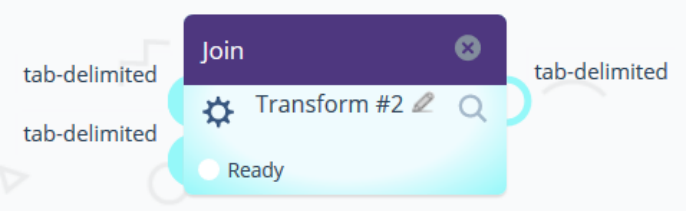


Figure 5 The Join component has two input nodes and 1 output node. Only one incoming connection is allowed per input node since it is defined as such in the XSD.

The corresponding XSD restricts the number of incoming connections per node to 1 file:

```
<xs:complexType name="InputType">
    <xs:sequence >
      <xs:element name="input0" type="InputDefinition0" minOccurs="1" />
      <xs:element name="input1" type="InputDefinition1" minOccurs="1" />
    </xs:sequence>
  </xs:complexType>
```

In the Java wrapper, you can access the file by node and index. The index is always 0 if only one incoming file is allowed per node.

```
File inputFile0 = this.getAttachment(0, 0); // ( node id, index id )

File inputFile1 = this.getAttachment(1, 0); // ( node id, index id )
```

Accordingly, if the wrapper is calling another program, the file paths will be passed via the command-line:

```
-file0 /path/to/file/input0.txt -file1 /path/to/file/input1.txt
```

Please note that previous versions of this documentation said to use `xs:choice` or `xs:all`. It is now recommended to use `xs:sequence` for the InputType definition.

Unordered Inputs (many incoming connections per node)

What if we want to allow any number of components to connect to a single input node? This is where node and file indices come into light. If the input is "unbounded", then any number of connections can be attached to an input node. However, the files are not guaranteed to be in any particular order when executed unless that order is explicitly set by the end-user.

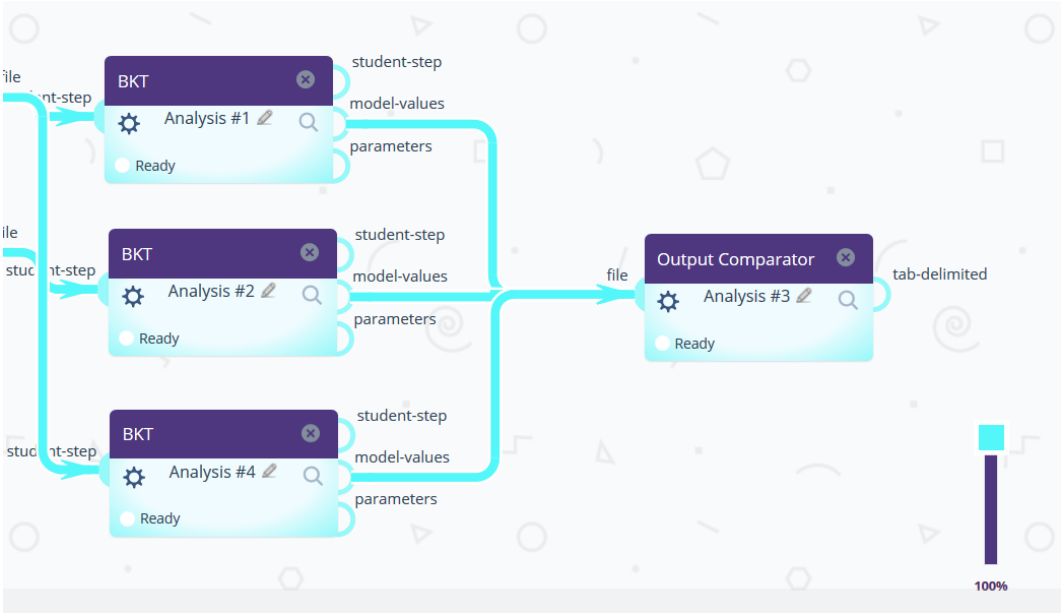


Figure 6 The Output Comparator component compares any number of tab-delimited, XML, or .properties files.

To allow any number of (unordered) connections to a single input node, you must use `maxOccurs="unbounded"` for the InputType definition. Using the following, any number of input files can be received on the input nodes:

```
<xs:complexType name="InputType">
  <xs:sequence>
    <xs:element name="input0" type="InputDefinition0" minOccurs="1" maxOccurs="unbounded" />
    <xs:element name="input1" type="InputDefinition1" minOccurs="1" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

Program parameters

Node and File Indices

If your component is calling a program, the input file paths are put into file bins. An input node that receives more than one file will use positional arguments to specify the node and file indices of each file, e.g.

```
-node 0 -fileIndex 0 /path/to/file/training0.txt

-node 0 -fileIndex 1 /path/to/file/training1.txt

-node 1 -fileIndex 0 /path/to/file/test1.txt

-node 1 -fileIndex 1 /path/to/file/test0.txt
```

In the above example, each node has two incoming files attached. Note that because the incoming connections are to a single node, the order of the files cannot be guaranteed. To maintain the order of two or more arbitrary files, one must use separate nodes for each.

Here's another example of a component calling an R program with 3 incoming connections on a single input node:

```
/usr/bin/Rscript   /dev/WorkflowComponents/MyAnalysis/program/myProgram.R
  -programDir   /dev/WorkflowComponents/MyAnalysis/
   -workingDir   /workflows/201/Analysis-1-x375456/output/
    -userId   drew_straws   -caseSensitive   Yes
     -node 0 -fileIndex 0   /workflows/201/Import-1-x735929/output/test_0.txt
      -node 0 -fileIndex 1   /workflows/201/Import-1-x605023/output/test2.txt
       -node 0 -fileIndex 2  /workflows/201/Import-1-x735929/output/test1.txt
```

Get Input Files – R example

```
i = 1
inputFiles <- vector()

while (i <= length(args)) {
    if (args[i] == "-node") {
       if (length(args) == i) {
          stop("fileIndex and file must be specified")
       }
       nodeIndex <- args[i+1]
       fileIndex = NULL
       fileIndexParam <- args[i+2]
       if (fileIndexParam == "fileIndex") {
           fileIndex <- args[i+3]
       }

       inputFiles[nodeIndex] = args[i+4]
       i = i + 4
```

```
    } else if (args[i] == "-workingDir") {
       if (length(args) == i) {
          stop("workingDir name must be specified")
       }
       # This dir is the "working directory" for the component instantiation, e.g.
/workflows/<workflowId>/<componentId>/output/.
       workingDirectory = args[i+1]
       i = i+1
    } else if (args[i] == "-programDir") {
       if (length(args) == i) {
          stop("programDir name must be specified")
       }
       # This dir is WorkflowComponents/<ComponentName>/
       componentDirectory = args[i+1]
       i = i+1
    }
    i = i+1
}


# do something with inputFiles[0], inputFiles[1], etc...
```

Get Input Files – Python example

```
import sys, argparse

inFile0 = None
inFile1 = None

parser = argparse.ArgumentParser(description='Process datashop file.')

parser.add_argument('-programDir', type=str,
          help='the component program directory')

parser.add_argument('-workingDir', type=str,
          help='the component instance working directory')

parser.add_argument("-node", nargs=1, action='append')
parser.add_argument("-fileIndex", nargs=2, action='append')

parser.add_argument('-userId', type=str,
          help='the user executing the component', default='')

args, option_file_index_args = parser.parse_known_args()

for x in range(len(args.node)):
    if (args.node[x][0] == "0" and args.fileIndex[x][0] == "0"):
        inFile0 = args.fileIndex[x][1]
    if (args.node[x][0] == "1" and args.fileIndex[x][0] == "0"):
        inFile1 = args.fileIndex[x][1]
```

```
parser.add_argument('-myEnumOption', choices=["Apple", "Orange"],
           help='an enum option (default="Apple")',
            default="Apple")

parser.add_argument('-myStringOption', type=str,
           help='a generic string option')
```

Get Input Files – Java wrapper example

From the Java wrapper, you can access the files in several ways.

Input Files

Access incoming files by their node and index values, where order is only guaranteed for the node and not for the files going to a single node.

```
File training0 = this.getAttachment(0, 0); // ( node 0, index 0 )

File training1 = this.getAttachment(0, 1); // ( node 0, index 1 )

File test0 = this.getAttachment(1, 0); // ( node 1, index 0 )

File test1 = this.getAttachment(1, 1); // ( node 1, index 0 )
```

Access all files on a specific node.

```
int nodeIndex = 0;
List<File> inputFiles0 = this.getAttachments(nodeIndex);

if (input0Files != null) {
   for (File file : inputFiles0) {
          // do something here
          logger.info(file.toString());
   }
 }
```

Get the entire input mapping (all nodes and file indices stored in a map).

```
Map<Integer, List<File>> inFilesByNodeIndex = this.getAttachments();

 for (Integer nodeIndex : inFilesByNodeIndex.keySet()) {
       for (File file : inFilesByNodeIndex.get(nodeIndex)) {
              // do something
              logger.info(file.toString());
       }
 }
```

Get the actual file type of an incoming, e.g. “tab-delimited”, “csv”, etc.

This returns the actual file type of the input file, not the expected type. In this case, csv is the actual file type being passed. In future versions, we may automatically convert file types when possible.

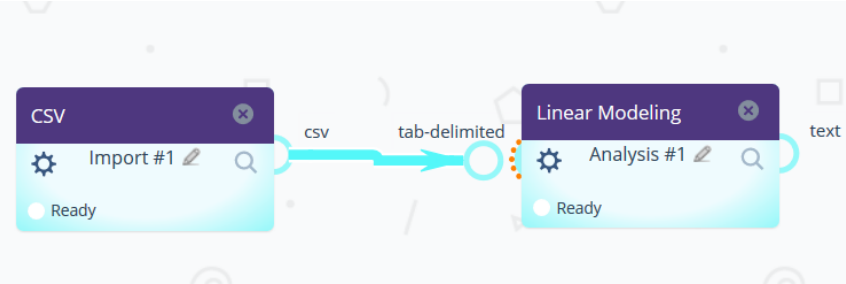


Figure 7 The actual file type is CSV. The expected file type is tab-delimited.

```
int nodeIndex = 0;
String attachType = getAttachmentType(nodeIndex);
```

Get Input Files – Java example (for external Java programs and jars)

```
/* The parameter syntax for files is -node m -fileIndex n <infile>. */
Map<Integer, File> inFiles = new HashMap<Integer, File>();

for ( Integer i = 0; i < args.length; i++) {
    String arg = args[i];
    String nodeIndex = null;
    String fileIndex = null;
    String filePath = null;
    if (i < args.length - 4) {
        if (arg.equalsIgnoreCase("-node")) {
            File inFile = null;
            String[] fileParamsArray = {
                args[i] /* -node */, args[i+1] /* node (index) */,
                args[i+2] /* -fileIndex */, args[i+3] /* fileIndex */,
                args[i+4] /* infile */ };
            String fileParamsString = Arrays.toString(fileParamsArray);
            // Use regExp to get the file path
            String regExp = "^\\[-node, ([0-9]+), -fileIndex, ([0-9]+), ([^\\]]+)\\]$";
            Pattern pattern = Pattern.compile(regExp);
            if (fileParamsString.matches(regExp)) {
                // Get the third argument in parens from regExp
                inFile = new File(fileParamsString.replaceAll(regExp, "$3"));
            }
            nodeIndex = args[i+1];
            Integer nodeIndexInt = Integer.parseInt(nodeInde);
            fileIndex = args[i+3];
            inFiles.put(nodeIndexInt, inFile);
            // 5 arguments, but for loop still calls i++ after
            i += 4;
        }
    }
```

```
}
```

Outputs

The output definitions are similar to the input definitions. One must include **OutputDefinitionX**, **OutFileListX**, and the **OutputType**. As in the InputType, the OutputType defines the order of the output nodes. In our example, we only have one output, but we still must include a reference to it in the **OutputType**.

```
<xs:complexType name="OutputDefinition0">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="OutFileList0" name="files" />
        <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="OutputType">
  <xs:sequence>
    <xs:element name="output0" type="OutputDefinition0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="OutFileList0">
  <xs:choice>
    <xs:element ref="tab-delimited" />
  </xs:choice>
</xs:complexType>
```

Import Components (Components Without Inputs)

Import components do not have an **InputType** definition, as its files are provided by the user or from an external repository. Defining a simple import can be done with the File Upload Option in the next section.

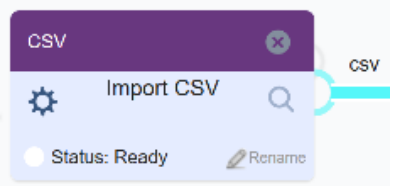


Figure 8 An Import component does not have inputs

Extracting Compressed Input Files (zip, tar.gz, tar.bz2)

The component wrapper provides a convenience method for extracting a compressed input file (.zip, .tar.gz, or .tar.bz2) and returning the output directory. The uncompressed folder can then be used for further work. In the following example, a compressed file is extracted to a temporary directory. This directory (zipOutputDirectory) is used to obtain a file known to exist in the zip (ds1_student_step_export.txt). The text file is then substituted for the zip file in the argument -file0 when runExternal() is called. Note that runExternal() calls the program specified in your component's build.properties file.

```
@Override
protected void runComponent() {
	// Get the first file (zip) on the first input node:
	Integer inNodeIndex = 0;
	Integer inFileIndex = 0;
	File zipOutputDirectory = null;
	try {
		// Un-compress the zip file and return the folder containing the extracted files
		zipOutputDirectory = this.getAttachmentAndUnzip(inNodeIndex, inFileIndex);
	} catch (Exception e) {
		System.err.println("Cannot unzip compressed input file.");
	}

	if (zipOutputDirectory != null) {
		// Get the desired file path (relative)
		String zipOutputPath = zipOutputDirectory.getAbsolutePath().replaceAll("\\\\", "/");
		// In this example, we already know the realtive path to the desired file.
		String desiredFile =  zipOutputPath + "/ds1_student_step_export.txt";
		File rCreatedFile = new File(desiredFile);
		this.setInputFile(inNodeIndex, inFileIndex, rCreatedFile);
	 // Run the program and return its stdout to a file.
	 File outputDirectory = this.runExternal();

	 Integer nodeIndex = 0;
	 Integer fileIndex = 0;
	 String fileLabel = "text";
	 // Add the new file (created by analysis.R) to the output file list.
	 this.addOutputFile(new File(outputDirectory + "/my_output_file.txt"),
		nodeIndex, fileIndex, fileLabel);
	}
    // Send the component output bakc to the workflow.
    System.out.println(this.getOutput());
}
```

Next, an advanced example shows a similar scenario except any number of files will be used as the input to analysis.R, e.g. -file0 firstFile.txt -file1 secondFile.txt …

```
@Override
protected void runComponent() {

	Integer inNodeIndex = 0;
	Integer inFileIndex = 0;
	File zipOutputDirectory = null;
	try {
		zipOutputDirectory = this.getAttachmentAndUnzip(inNodeIndex, inFileIndex);
	} catch (Exception e) {
		System.err.println("Cannot unzip compressed input file.");
	}

	if (zipOutputDirectory != null) {
		// Get the file path in a windows-/linux-/mac- friendly format
		String zipOutputPath = zipOutputDirectory.getAbsolutePath().replaceAll("\\\\", "/");
		for (String fileName : zipOutputDirectory.list()) {

			// Use the full file path replace or add the inputs.
			// Uncompressed directories within the zipOutputPath can be used here:
	    String filePath = zipOutputPath
			+ "/"
		/*  + "someSubDirectory/"             */
```

```
                    + fileName;

            File rCreatedFile = new File(filePath);
            if (rCreatedFile.isFile()) {
                    this.setInputFile(inNodeIndex, inFileIndex, rCreatedFile);
                    inNodeIndex++;
            } else if (rCreatedFile.isDirectory()) {
               // This File object is actually a directory.
               // We could traverse it or simply ignore it.
            }
           }

               // Run the program analysis.R (as specified in build.properties),
               // and return its stdout to "my_output_file.txt".
        File outputDirectory = this.runExternal();

        Integer nodeIndex = 0;
        Integer fileIndex = 0;
        String fileLabel = "text";
        // Add the new file (created by analysis.R) to the output file list.
        this.addOutputFile(new File(outputDirectory + "/my_output_file.txt"), nodeIndex, fileIndex,
fileLabel);
       }

    // Send the component output back to the workflows platform.
    System.out.println(this.getOutput());
}
```

Defining Options

The **OptionsType** defines a set of options which can be modified from the user-interface. The options can include *simple data types*, like strings or doubles, as well as a *file upload option*. Let's take a look at the different configurations available.

File Upload Option

Including a *files* element in an OptionsType allows the end-user to upload a file from the component. Generally, a component will either have Inputs, or it will have a *file upload option*. Components that have both should be split into two separate components—an Import and an Analysis component. Let us look at the *file upload* definition.

Modify t*ext* in the **OptionFileList** to match your desired file type. See the section Existing File Types in Appendix A. Examples include tab-delimited, csv, image, and text, but we can invent new types and place them into a hierarchy, e.g. csv (comma-separated values) is a type of text file so any csv will suffice as a text file, but not all text files are csv files.

```
<xs:complexType name="OptionsType">
  <xs:all minOccurs="1">
    <xs:element name="files" type="OptionFileList" minOccurs="1" maxOccurs="1" />
  </xs:all>
</xs:complexType>
```

```
<xs:complexType name="OptionFileList">
  <xs:choice>
    <xs:element ref="student-step" minOccurs="1" maxOccurs="1" />
  </xs:choice>
</xs:complexType>
```

Simple Data Types Options

Allow users to set several simple data types through your component's interface.

```
<xs:complexType name="OptionsType">
  <xs:all >

    <xs:element type="xs:string" name="model" id="Model" default="Default" />

    <xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />

  </xs:all>
</xs:complexType>
```

Options have several attributes which must be set. They are type, default, name, and id.

- **Type** – Can be any primitive XML data type: **xs:string, xs:double, xs:integer, xs:boolean, xs:date, xs:time, …** For a full list of types, see http://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes
- **Default** – Specifies the default value of the element if none is provided by the user
- **Name** – A unique name for each option. A name must start with a a-z or A-Z and be followed by any alphanumeric characters, hyphens (-), underscores (_), or dashes (-)
  Additionally, if the component executes an external program which accepts command-line arguments, then the **name** attribute is used as the argument identifier, i.e. name="xValidationFolds" default="10" is converted to the command-line pair: -xValidationFolds 10
- **ID** – It follows the same naming convention as the *name* attribute. The ID is displayed in the component options pane to the user, and its underscores are replaced with spaces, e.g. *Cross-validation_Folds* becomes "Cross-validation Folds" when displayed.
- **ls:privateOption** [not required] – An option might contain sensitive data such as a password or personal identifier. In this case, only the owner might want to be able to see this information. However, the owner might want to share the rest of the workflow by making it public. Marking an option as private will ensure that the value of the option is only visible to the owner of the workflow, even if the workflow is public. If you want the option to be private, include the attribute/value: ls:privateOption=”true”, otherwise, this attribute does not need to be included. Only options of type “xs:string” may be marked as private.
    - `<xs:element type="xs:string" name="student_id" id="Student_ID" default="123" ls:privateOption="true" />`

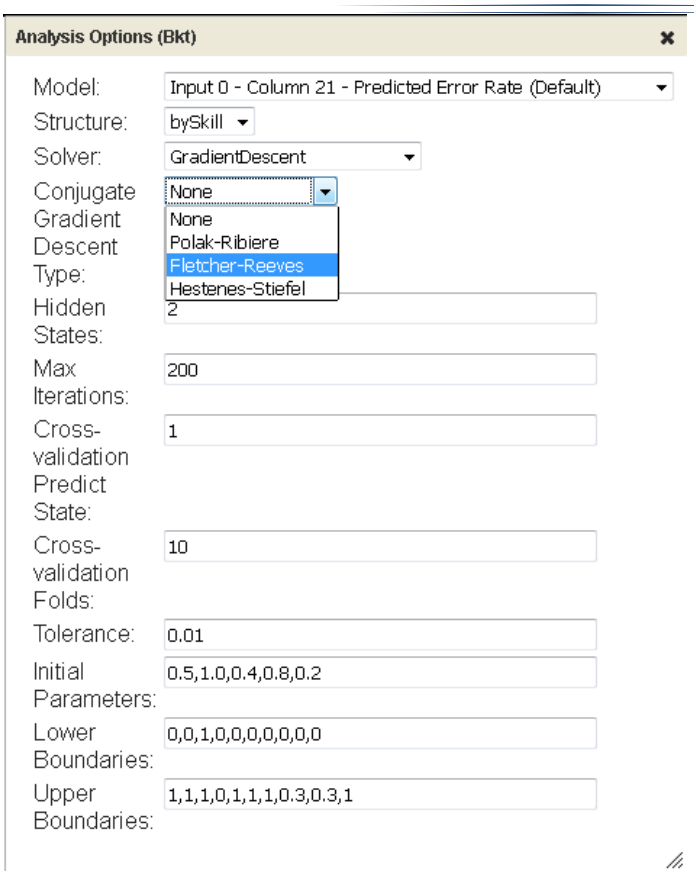


Figure 9 An Options dialog box is automatically created from the XML Schema Definition for each component.

Options with File Upload and Simple Data Types

Allow users to set both simple data types and upload a file via the component interface.

```
<xs:complexType name="OptionsType">
    <xs:all >

      <xs:element type="xs:string" name="model" id="Model" default="Default" />

      <xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />

      <xs:element name="files" type="OptionFileList" minOccurs="1" maxOccurs="1" />

    </xs:all>
  </xs:complexType>


  <xs:complexType name="OptionFileList">
    <xs:choice>
      <xs:element ref="text " minOccurs="1" maxOccurs="1" />
    </xs:choice>
  </xs:complexType>
```

Any options you specify will be used to generate an **options pane** for your component. Note that "INF" is the XML symbol for infinity but is only acceptable for "xs:*double*" data types. If you need to allow infinity to be a valid option, the option type must be "xs:double".

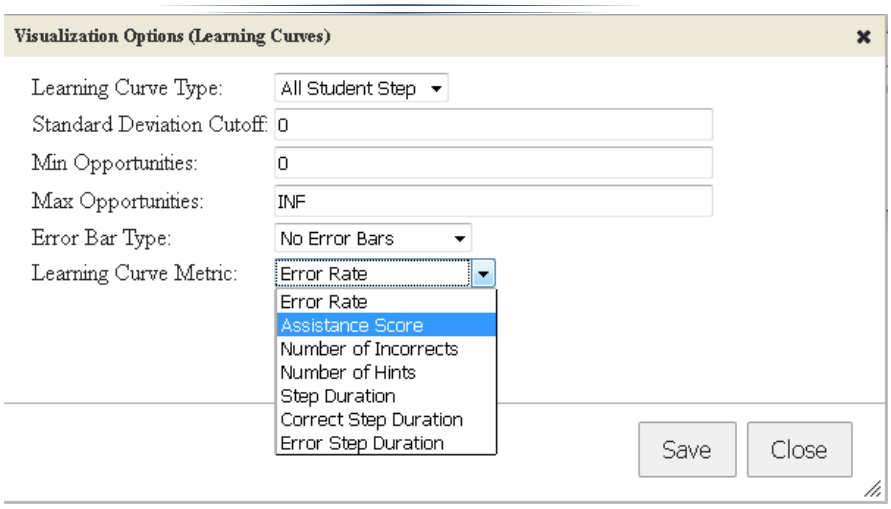


Figure 10 - An example of several different OptionType elements being displayed to the user.

Drop-down Options

Drop-down Options can also be defined in the XSD. They are displayed to the user as drop-downs in the component options interface. Simply define a new enumeration in your component's XSD. You may also use the predefined xs:boolean type. The system requires the suffix "Type" when defining Drop-down Options, e.g. errorBarType in the following example:

```
<xs:simpleType name="errorBarType" final="restriction">

  <xs:restriction base="xs:string">
    <xs:enumeration value="No Error Bars" />
    <xs:enumeration value="Standard Deviation" />
    <xs:enumeration value="Standard Error" />
  </xs:restriction>
</xs:simpleType>
```

After the drop-down option is defined, add it to the **OptionsType** definition which contains all of the options. Note that the type is no longer a **simple data type**—it is the type we defined, *errorBarType*.

```
<xs:complexType name="OptionsType">
  <xs:all >

    ...

    <xs:element type="errorBarType" name="errorBar" id="Error_Bar_Type" default="No Error Bars" />

    ...

  </xs:all>
</xs:complexType>
```

Options Based on Inputs

One useful option type is the **FileInputHeader**. If a component passes any kind of data table (csv, tab-delimited, etc.), then the next component can use the column headers from the data in its own options interface. One can

filter columns by using Java regular expressions in the "**default**" property of the **FileInputHeader** in the **OptionsType** definition.

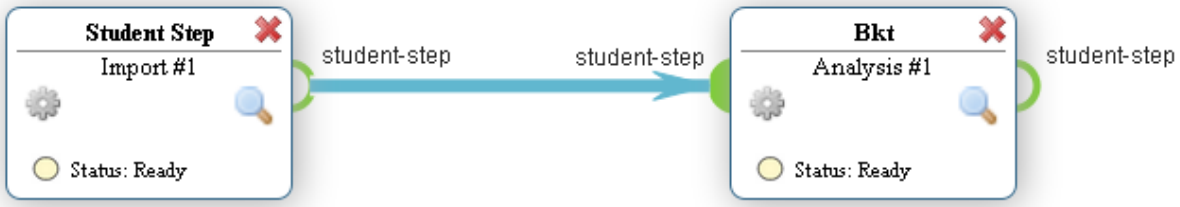


Figure 11 The column headers of the Student Step output can be utilized by the Analysis component.

For example, suppose an input file contains several columns of interest, and all of the contain the word "answer" in the column header. One can match any column header with the word "answer" in it with the Java regular expression "(?i).*answer.*" where (?i) declares it a case-insensitive match.

To allow the user to select from any input column, simply use ".*" which matches any words with 0 or more characters of any kind. Regular expressions are beyond the scope of this document, but feel free to contact us if you have questions.

```
Match all columns which contain one or more letters (a-zA-Z)

  <xs:complexType name="OptionsType">
    <xs:all>
      <xs:element type="FileInputHeader" name="model" id="Model" default="(?i).*answer.*" />
        <!-- Other options... -->
    </xs:all>
  </xs:complexType>
```

The UI will contain an option drop-down, allowing them to select one of the columns returned by the FileInputHeader. We see that it identifies the index of the input node, the column number, and the actual column header.

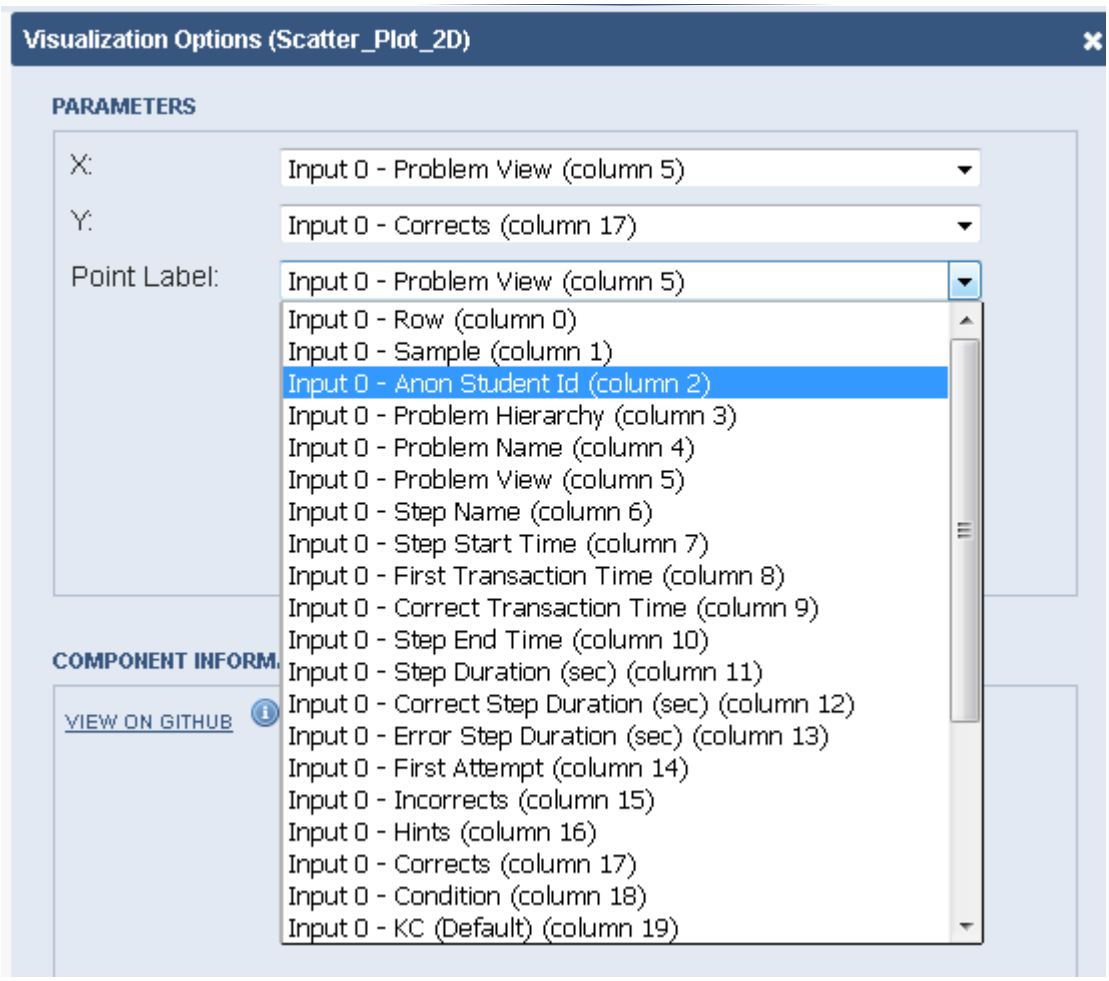
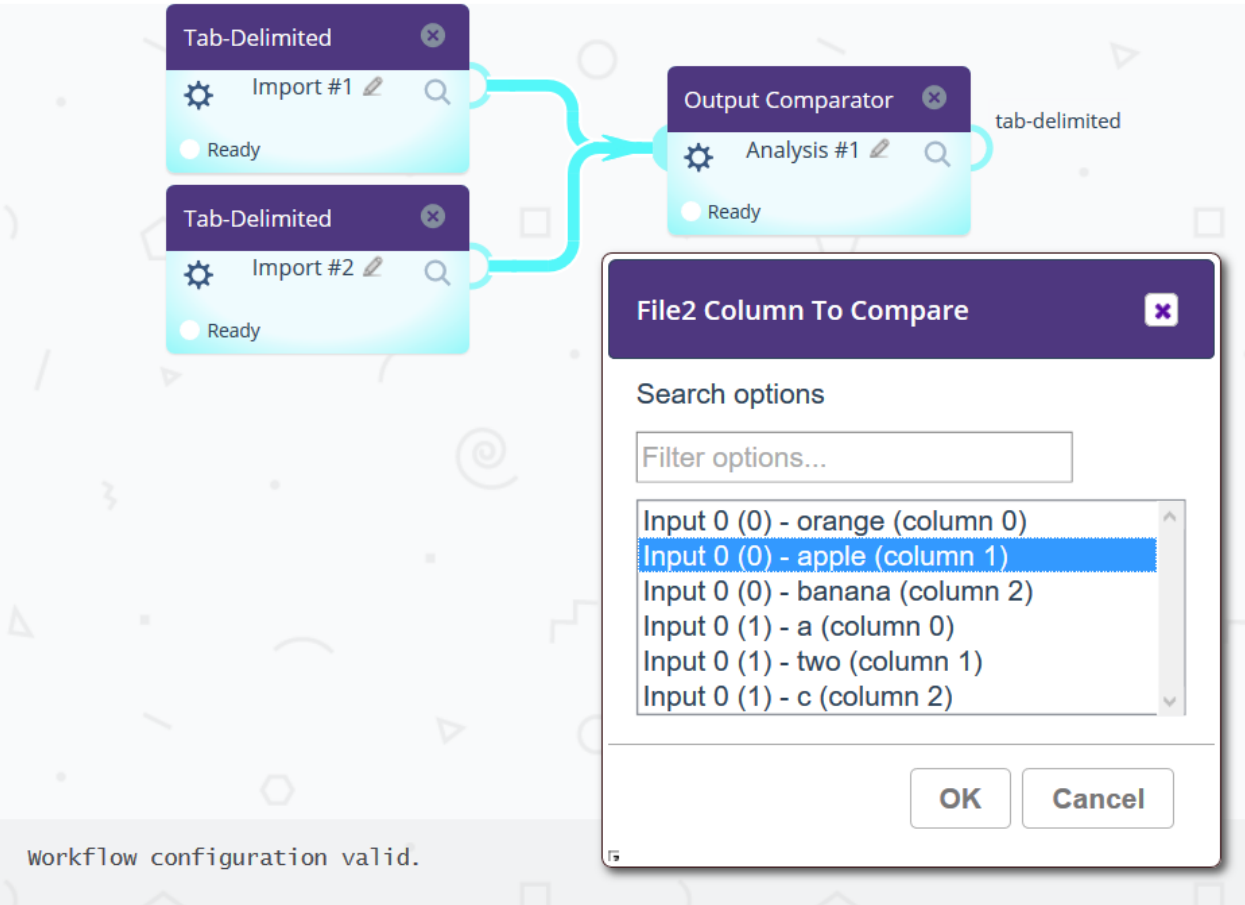


Figure 12 Matching columns using the regular expression ".*" (any characters)

FileInputHeader and MultiFileInputHeader with Multiple Inputs

The (Multi)FileInputHeader option types depend on the columns of the incoming files. In the case where the option depends on multiple incoming nodes or multiple incoming files or both, the *node index* and *file index* are made available to the program.



In the XSD, you can specify which node(s) and file(s) to filter on by adding ls:inputNodeIndex and ls:InputFileIndex to your (Multi)FileInputHeader option definitions:

```
Only show column headers and meta-data from node 0, file index 0.
<xs:element type="FileInputHeader" name="file1ColumnName" id="File_1_Column_Name"
default=".*" ls:inputNodeIndex="0" ls:inputFileIndex="0" />

Show column headers and meta-data from any file on node 0.
<xs:element type="FileInputHeader" name="file2ColumnName" id="File_2_Column_Name"
default=".*" ls:inputNodeIndex="0" ls:inputFileIndex="*"  />

Show column headers and meta-data from the first file on every node.
```

```
<xs:element type="FileInputHeader" name="file2ColumnName" id="File_2_Column_Name"
default=".*" ls:inputNodeIndex="*" ls:inputFileIndex="0"  />

Show column headers and meta-data from all files on all nodes.
<xs:element type="FileInputHeader" name="file2ColumnName" id="File_2_Column_Name"
default=".*" ls:inputNodeIndex="*" ls:inputFileIndex="*"  />
```

Permitted values for **ls:inputNodeIndex** and **ls:inputFileIndex** include an integer value starting at 0, or you can use "*" to use all nodes or files.

Getting (Multi)FileInputHeader Options as Program Parameters

Suppose we have a FileInputHeader option which has 3 incoming connections (files) on node 0. The program parameters should reflect the input node index and file index along with the selected column header:

When your component is executed, the command-line parameters passed to your program will include additional arguments using **_nodeIndex** and **_fileIndex** suffixes attached to option name, e.g. if the name of the (Multi)FileInputHeader option is "compareColumn", then your program will receive:

```
-compareColumn   c   -compareColumn_nodeIndex  0   -compareColumn_fileIndex  1
```

Note that these are positional arguments. If using MultiFileInputHeader, there could be multiple compareColumn values, each with their own node and file index:

```
-compareColumn   c   -compareColumn_nodeIndex  0   -compareColumn_fileIndex  0

-compareColumn   d   -compareColumn_nodeIndex  1   -compareColumn_fileIndex  0
```

Getting (Multi)FileInputHeader Options in the Java Wrapper

Since FileInputHeader and MultiFileInputHeader option types depend on incoming files, they will contain node and file index information. To handle this requirement, methods specific to these two option types can be obtained using the getInputHeaderOption methods:

List<InputHeaderOption> **getInputHeaderOption**(String elementName) // all nodes and files

List<InputHeaderOption> **getInputHeaderOption**(String elementName, Integer nodeIndex) // for a specific node

List<InputHeaderOption> **getInputHeaderOption**(String elementName, Integer nodeIndex, Integer fileIndex) // for a specific node and file index

Here's an example of how to retrieve all option values for a MultiFileInputHeader option called "myColumns". The FileInputHeader option works the same way, except that it will limit the user to only one selection while MultiFileInputHeader allows multiple values to be selected. Whichever you use, the method is the same:

```
List<InputHeaderOption> matchColumnList =
    this.getInputHeaderOption("myColumns", 0);

    for (InputHeaderOption iho : matchColumnList) {
       logger.info("InputHeader name: " + iho.getOptionName()
              + ", value: " + iho.getOptionValue()
              + ", node index: " + iho.getNodeIndex()
              + ", file index: " + iho.getFileIndex());
    }
```

Passing Additional Arguments to a Bootstrap Program

We already know how to define options in the XSD, but what if we want to programmatically pass additional arguments to the bootstrap program which aren't defined in the XSD?

If you want to send additional parameters to your program that you do not want in the interface (for example, a -debug option, or a private user/password, then you can update the Java wrapper to do so. Your new parameters will be added to the program call, and you can either make that data known to downstream components or make it private so that other components cannot see the data.

** If your component is pure java and does not rely on build.properties, then this does not apply to you. However, if you are using a different language, like R, Python, or C++, then the following may be useful.

Example

To see how the bootstrap program is called (the program defined in build.properties), we can examine the WorkflowComponent.log. In the following example, *model* is an option defined in the XSD and *file0* is an input defined in the XSD. The other arguments, *programDir* and *workingDir* are always provided to the bootstrap program by the platform.

```
Executing process: [/usr/bin/Rscript,
/dev/WorkflowComponents/AnalysisIAfm/program/MyProgram.R,
-programDir, /dev/WorkflowComponents/AnalysisIAfm/,
-workingDir, /dev/WorkflowComponents/AnalysisIAfm/test/ComponentTestOutput/output/,
-model, "KC (Unique-step)",
-file0, /dev/WorkflowComponents/AnalysisIAfm/test/test_data/ds96_student_step_export.txt]
```

Modifying an existing option

We can *modify* or *add* program parameters using the **setOption** method **before** calling runExternal() or runExternalMultipleFileOutput().

If the option is defined in the XSD and setOption is called, it will replace the current value with the new option value.

```
this.setOption("model", "KC (Default)");
```

```
-model, "KC (Unique-step)" becomes -model, "KC (Default)"
```

Adding an option

If the option is *not* defined in the component XSD, then it will be added to the command-line parameters passed to MyProgram.R. Here is the example. Remember that the path to Rscript and MyProgram.R are defined within the build.properties file. What follows is the accompanying Java code which executes the bootstrap program (MyProgram.R).

// Begin code

```
@Override
    protected void runComponent() {
//_____________________________________________________________________________________
        // Add additional parameters to pass to MyProgram.R not defined in the options XSD
        this.setOption("my_option_id", "my option value");
//_____________________________________________________________________________________

        // Run the program...
        File outputDirectory = this.runExternalMultipleFileOuput();

        // Add the output file created by MyProgram.R to the component output
        Integer fileIndex = 0;
        Integer nodeIndex = 0;
        String fileLabel = "tab-delimited";
        File resultsFile = new File(
             outputDirectory.getAbsolutePath() + "/My_Program_Results.txt");

        this.addOutputFile(resultsFile, nodeIndex, fileIndex, fileLabel);
```

```
        // Send the component output back to the standard output (always required)
        System.out.println(this.getOutput());
    }

// End of code
```

The **setOption** method: **`void setOption ( String optionId, String optionValue )`**

- The first argument is the **option id**, e.g. “my_option_id”, may **only** contain alpha-numeric characters and _ (underscore), and the name cannot begin with an _ (underscore).
- The second argument is the **option value** and may contain any characters (spaces, quotes, punctuation are all acceptable)

Testing

Run your modified code from the command-line with **ant runComponent**

After running your code, check inside *WorkflowComponent.log*. You should see a line containing the executing process and all of the arguments that were passed to your program.

Notice our new option is included in the call to MyProgram.R, now.

```
Executing process: [/usr/bin/Rscript,
/dev/WorkflowComponents/AnalysisIAfm/program/MyProgram.R,
-programDir, /dev/WorkflowComponents/AnalysisIAfm/,
-workingDir, /dev/WorkflowComponents/AnalysisIAfm/test/ComponentTestOutput/output/,
-model, "KC (Unique-step)",
-my_option_id, "my option value",
-file0, /dev/WorkflowComponents/AnalysisIAfm/test/test_data/ds96_student_step_export.txt]
```

Using the Options in the Bootstrap Program

In your script, you can now access the additional parameters. Here is an example in R which retrieves several options, and looks for the tuple, in this case { “-my_option_id”, “my option value” }.

```
while (i <= length(args)) {
_____________________________________________________________________________
      if (args[i] == "-my_option_id") {
         if (length(args) == i) {
            stop("my_option_id requires a value")
         }

         newOptionValue = args[i+1]
         i = i+1
_____________________________________________________________________________

      } else if (args[i] == "-file0") {
         if (length(args) == i) {
            stop("file name must be specified")
```

```
         }
         stuStepFileName = args[i+1]
         i = i+1
      } else if (args[i] == "-model") {
         if (length(args) == i) {
            stop("model name must be specified")
         }
         modelName = args[i+1]
         i = i+1
      } else if (args[i] == "-workingDir") {
         if (length(args) == i) {
            stop("workingDir name must be specified")
         }
         workingDir = args[i+1]
         i = i+1
      }
      i = i+1
}
```

Component Output Changes

Not only can MyProgram.R access the new option, but downstream components can see it, as well. This can be observed by examining the component XML output generated by **ant runComponent**

The component output should contain a new "optionmeta" object in each of the output elements:

```
<optionmeta>
       <model>KC (Unique-step)</model>
       <my_option_id>my option value</my_option_id>
</optionmeta>
```

Adding Private Options (Sensitive Data)

If sensitive data should be passed to a bootstrap program, then we do not want to leak the data to the platform (via optionmeta elements). We can add a private parameter, but we cannot make private an existing option that was already defined in the XSD. Downstream components will not be able to see these options in the *optionmeta* element.

void **addPrivateOption**(String optionId, String optionValue)

```
@Override
    protected void runComponent() {
//_____________________________________________________________________________________
        // Add additional parameters to pass to MyProgram.R not defined in the options XSD
       this.addPrivateOption("my_option_id", "my option value");
//_____________________________________________________________________________________

        // Run the program...
        File outputDirectory = this.runExternalMultipleFileOuput();
       …
```

Using the *addPrivateOption* method, we can add the parameters to the command-line call to our program without letting downstream components see the option id or value of the new parameter(s). See how the private options are now appended to the platform-generated command-line call.

```
Executing process: [C:/R-3.4.2/bin/Rscript,
C:/Users/mkomisin/git/WorkflowComponents/AnalysisIAfm/program/iAFM.R,
-programDir, C:/Users/mkomisin/git/WorkflowComponents/AnalysisIAfm/,
-workingDir,
C:\Users\mkomisin\git\WorkflowComponents\AnalysisIAfm/test/ComponentTestOutput/output/, -
model, "KC (Default)",
-file0,
C:/Users/mkomisin/git/WorkflowComponents/AnalysisIAfm/test/test_data/ds96_student_step_expo
rt.txt,
-my_private_data, "secret" ]
```

Component Output Changes

The private option cannot be seen by downstream components. They only see options defined in the XSD OR those added with setOption. Note that my_private_data is missing from the generated output:

```
<optionmeta>
      <model>KC (Default)</model>
</optionmeta>
```

Option Array Types

Tigris supports array types for simple data types (double, integer, string), FileInputHeaders, and enumerated types (drop-down lists). The user can define default values for each value added, as well as the minimum and maximum number of allowed values. Here, we can see an array of type “double” with a minimum of 2 values and a maximum of 5 values allowed.

Program Changes

When an array type is defined, it can either be a **simple data** type (integer, double, string), a **FileInputHeader** type (a headered column in an input file), or it can be an **enumerated** type (drop-down list).

If a bootstrap program is executed, the array arguments will be passed in the command-line-- this consists of the option name, the number of values, and then the values, themselves.

-weights **3** 0.1 0.3 0.5 (3 to indicate the # of values, followed by the values)

To access these values from inside the Component’s Java wrapper, you can use the built-in method, getOptionAsList:

```
List<String> weights = null;
```

```
if (this.getOptionAsList("weights") != null) {
        weights = this.getOptionAsList("weights");
           logger.info("weights:: " + weights);
 }

for (String weight : weights) {
        try {
               Double dbl = Double.parseDouble(weight);
        } catch (NumberFormatException nfe) {
               System.err.println("Invalid weight value: " + weight);
        }
}
```

Defining Array Types

Simple Array Type

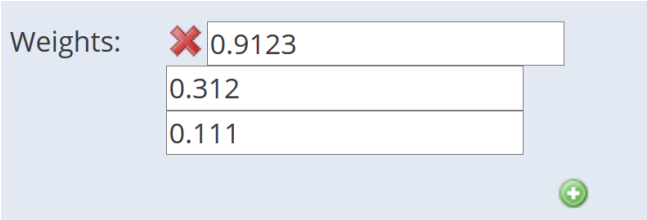


Figure. An array of xs:double values. To add or remove a value, click (+) or (X). Fields without an (X) are mandatory.

The XSD used to create an array of **xs:double** values is as follows. You can also use **xs:string** or **xs:integer**.

1. First, we define the weight type as a double with array attributes (hidden and optional).

```
<xs:complexType name="weight">
    <xs:simpleContent>
      <xs:extension base="xs:double">
        <xs:attribute name="type" type="xs:double"/>
        <xs:attribute name="hidden">
                        <xs:simpleType>
                         <xs:restriction base="xs:boolean">
                         </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="optional">
                        <xs:simpleType>
                         <xs:restriction base="xs:boolean">
                         </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
 </xs:complexType>
```

2. Next we define the actual array with default values, using minOccurs to indicate if the value is mandatory.

```
  <xs:all minOccurs="0" >
    <xs:element type="weight" name="weights0" default="0.1" minOccurs="1" />
    <xs:element type="weight" name="weights1" default="0.3" minOccurs="1" />
    <xs:element type="weight" name="weights2" default="0.5" minOccurs="0" />
    <xs:element type="weight" name="weights3" default="0.7" minOccurs="0" />
    <xs:element type="weight" name="weights4" default="0.9" minOccurs="0" />
  </xs:all>
 </xs:complexType>
```

3. Finally, we include the array in the OptionsType, as we do with any defined options.

```
<xs:complexType name="OptionsType">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element type="lineTypeArray" name="lineType" id="Line_Type" />
    <xs:element type="weightsArray" name="weights" id="Weights" />
    <xs:element type="columnHeaderArray" name="columns" id="Columns" />
  </xs:choice>
 </xs:complexType>
```

Enumerated Array Type

Arrays can consist of drop-down lists with a finite number of selections by employing “enumeration”.

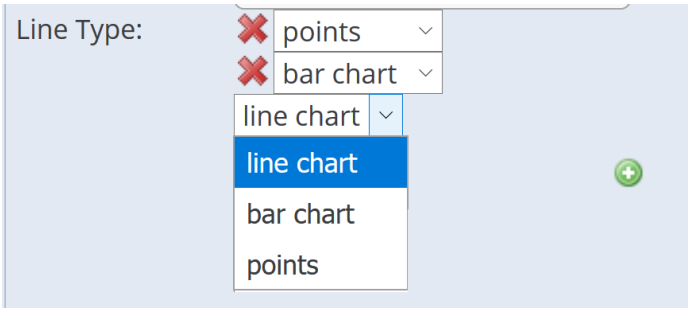


Figure. The drop-down boxes with (X) in front were added by the user and can be removed. Each drop-down contains all possible values for the enumerated type. In our example, this is { line_chart, bar_chart, points }.

1. First, define the enumerated type. It must be named with the suffix *Type.

```
 <xs:simpleType name="lineType" final="restriction">
   <xs:restriction base="xs:string">
     <xs:enumeration value="line chart" />
     <xs:enumeration value="bar chart" />
           <xs:enumeration value="points" />
```

```
  </xs:restriction>
 </xs:simpleType>

<xs:complexType name="lineStyle">
      <xs:simpleContent>
        <xs:extension base="lineType">
          <xs:attribute name="type" type="lineType"/>
          <xs:attribute name="hidden">
                              <xs:simpleType>
                               <xs:restriction base="xs:boolean">
                               </xs:restriction>
                              </xs:simpleType>
                    </xs:attribute>
                    <xs:attribute name="optional">
                              <xs:simpleType>
                               <xs:restriction base="xs:boolean">
                               </xs:restriction>
                              </xs:simpleType>
                    </xs:attribute>
        </xs:extension>
      </xs:simpleContent>
 </xs:complexType>
```

2. Next, define the array. It must match the name of the enumeration and use the suffix *Array.

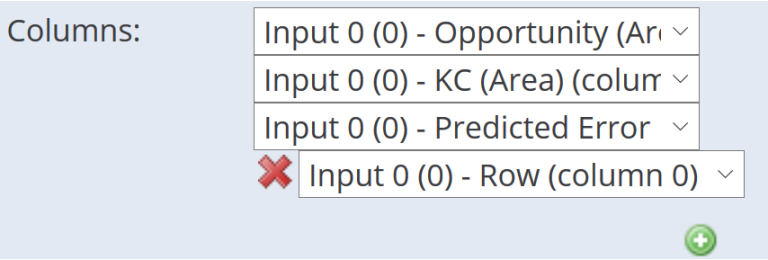
```
<xs:complexType name="lineTypeArray">
   <xs:all minOccurs="0" >
    <xs:element type="lineStyle" name="lineStyle0" default="line chart" minOccurs="1" />
    <xs:element type="lineStyle" name="lineStyle1" default="bar chart" minOccurs="0" />
    <xs:element type="lineStyle" name="lineStyle2" default="points" minOccurs="0" />
   </xs:all>
 </xs:complexType>
```

3. Include the array definition in the OptionsType. The “type” must be the array type. The “name” is used as the parameter passed to any programs. The “id” is used as the label in the GUI.

```
<xs:complexType name="OptionsType">
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element type="lineTypeArray" name="lineType" id="Line_Type" />
    <xs:element type="weightsArray" name="weights" id="Weights" />
    <xs:element type="columnHeaderArray" name="columns" id="Columns" />
   </xs:choice>
 </xs:complexType>
```

FileInputHeader Array Type

Although multiple column headers can be selected using MultiFileInputHeader options, using arrays offers more powerful features, such as specifying a min or max number of array values, as well as specifying different defaults for each new row.



1. First, define a unique element which extends FileInputHeader. Give it an arbitrary name, like column.

```
<xs:complexType name="column">
    <xs:simpleContent>
      <xs:extension base="FileInputHeader">
        <xs:attribute name="type" type="FileInputHeader"/>
        <xs:attribute name="hidden">
                        <xs:simpleType>
                          <xs:restriction base="xs:boolean">
                          </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="optional">
                        <xs:simpleType>
                          <xs:restriction base="xs:boolean">
                          </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
 </xs:complexType>
```

2. Next, create an array using the column type we defined. In this example,
   a. The first 3 array values are required (minOccurs="1")
   b. The first 3 default values use different regular expressions; the last 2 values match any column (.*)
   c. The UI will automatically select the first matching column from the input file for each row

```
<xs:complexType name="columnHeaderArray">
  <xs:all minOccurs="0" >
    <xs:element type="column" name="column0" default="\s*KC\s*\((.*)\)\s*" minOccurs="1" />
```

```
    <xs:element type="column" name="column1" default="\s*Opportunity\s*\((.*)\)\s*" minOccurs="1" />
    <xs:element type="column" name="column2" default="\s*Predicted Error Rate\s*\((.*)\)\s*" minOccurs="1" />
    <xs:element type="column" name="column3" default=".*" minOccurs="0" />
    <xs:element type="column" name="column4" default=".*" minOccurs="0" />
   </xs:all>
  </xs:complexType>
```

3. Finally, add the columnHeaderArray to the OptionsType definition.

```
<xs:complexType name="OptionsType">
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element type="FileInputHeader" name="model" id="Model" default="\s*KC\s*\((.*)\)\s*" />
          <xs:element type="lineTypeArray" name="lineType" id="Line_Type" />
    <xs:element type="weightsArray" name="weights" id="Weights" />
    <xs:element type="columnHeaderArray" name="columns" id="Columns" />
   </xs:choice>
  </xs:complexType>
```

Dynamic Options

Dynamic options allow us to build interactive user interfaces by making the visibility of one option dependent on one or more constraints. In other words, if an option depends on the value of one or more other options, then it will only be shown if the constraints applied to the independent option(s) are met. These dependencies and constraints can be defined in the component schema (XSD).

XSD changes

To use dynamic options in your component, then first you must replace the standard schema root node (first line in the XSD) with the following which defines the LearnSphere namespace.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ls="http://learnsphere.org/ls" >
```

Simple Example

Suppose your component allows the end-user to choose between two classifiers, SVM and Naive Bayes. The first classifier, SVM, requires a float argument, C, while the second, Naive Bayes, requires a boolean argument, "laplace smoothing". We only want to show the option "C" (float) if the algorithm selected is "SVM". If the selected algorithm is "Naive Bayes", then show the "laplace smoothing" (boolean) option.

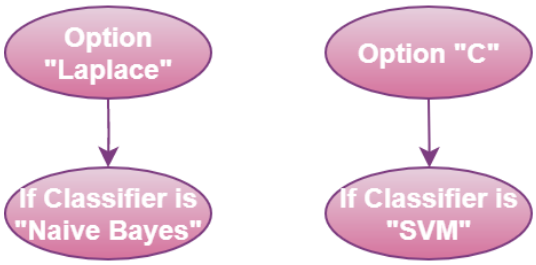
Pseudocode

if (CLASSIFIER.matches(“NAÏVE BAYES”))

SHOW LAPLACE

if (CLASSIFIER.matches(“SVM”))

SHOW C



Actual code

Let’s examine the relevant schema records that will be used to describe these two mutually exclusive relationships. Since the dependencies are mutually exclusive, i.e. they do not share the same dependent option, then it does not matter whether you choose conjunctive or disjunctive as the dependency type in this case.

```
<xs:complexType name="option_dependency">
	<xs:choice>

	<xs:element type="disjunctive" name="dependency1"
		ls:dependentOption="laplace" ls:dependsOn="classifier"
			ls:constraint="matches(Naive Bayes)" />

	<xs:element type="disjunctive" name="dependency2"
		ls:dependentOption="c" ls:dependsOn="classifier"
			ls:constraint="matches(SVM)" />

	</xs:choice>
</xs:complexType>


<xs:simpleType name="classifierType" final="restriction">
	<xs:restriction base="xs:string">
		<xs:enumeration value="Naive Bayes" />
		<xs:enumeration value="SVM" />
	</xs:restriction>
</xs:simpleType>

<xs:complexType name="OptionsType">
	<xs:choice minOccurs="0" maxOccurs="unbounded">
		<xs:element type="MultiFileInputHeader" name="file1ColumnName"
			id="Column_Names" default=".*" />
		<xs:element type="classifierType" name="classifier" id="Classifier"
			default="Naive Bayes" />
		<xs:element type="xs:boolean" name="laplace" id="Laplacian Smoothing"
			default="false" />
		<xs:element type="xs:double" name="c" id="C" default="0" />
	</xs:choice>
</xs:complexType>
```

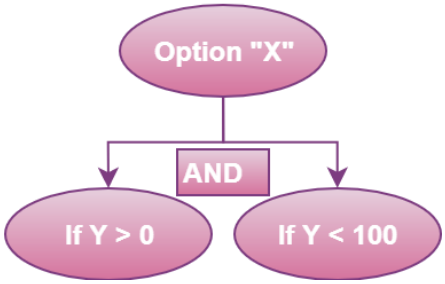
The above example shows how one can have custom menus based on simple dependencies. However, more can be achieved by combining constraints using logical operators.

Logical Operators

Logical operators AND and OR can be used to build complex logical dependencies. Let's look at an example using a logical conjunction with AND. Option X depends on two constraints placed on option Y. If the value of option Y meets both constraints, then a previously hidden option X will be presented to the user.

if (Y > 0 and Y < 100)

Show X



The relevant XSD below.

```
<xs:complexType name="option_dependency">
	<xs:choice>

	<!-- Show x if y > 0 and y < 100 -->
	<xs:element type="conjunctive" name="dependency1"
		ls:dependentOption="x" ls:dependsOn="y"
			ls:constraint="lt(100)" />

	<xs:element type="conjunctive" name="dependency2"
		ls:dependentOption="x" ls:dependsOn="y"
			ls:constraint="gt(0)" />

	</xs:choice>
</xs:complexType>

<xs:complexType name="OptionsType">
	<xs:choice minOccurs="0" maxOccurs="unbounded">
		<!-- Note that the default value for y is 0 so option X
		will not initially be shown until y is changed and the
		constraints are met. -->
		<xs:element type="xs:integer" name="x" id="x" default="0" />
		<xs:element type="xs:integer" name="y" id="y" default="0" />
	</xs:choice>
</xs:complexType>
```

Attributes of a dependency

1. **name** - the unique name for each dependency; Note that a dependency can be defined that names itself as the dependent option, but a dependency cannot depend on itself.

2. **dependentOption** - the dependent option is only displayed to the user if all its dependency constraints are met

3. **dependsOn** - the indepdendent option can be any option name or any other dependency name; an option cannot depend on itself.

4. **constraint** - a logic statement that can be resolved to true or false; constraints are usually in the form "matches(someString)", "greaterThan(someNumber)", etc...

5. **negation** (optional) - if used in a dependency, will return the logical complement of the resolved constraint; when used, it is always in the form **ls:negation="true"**

6. **type** - *disjunctive* or *conjunctive*, this attribute defines the logical operator used in resolution of options which depend on more than one dependency; If a "dependentOption" is used in only one dependency, then the connective type is superfluous. However, if more than dependency uses the same "dependentOption" value, then the group of dependencies are resolved according to their type. Conjunctive types are AND'ed together while disjunctive types are OR'ed together. The resulting logical relation determines whether or not the "dependentOption" value is displayed to the user.

Constraints

The dependent option is presented to the user if the constraints on the independent option(s) are met. The following logical constraints determine the truth values of the dependencies.

String-based Constraints

String constraints are met if the value of the independent option matches the regular expression specified in the constraint.

matches(<regular_expression>) - Resolves true if the value of the "dependsOn" option matches the regular expression. More on JavaScript regular expressions can be found here, https://www.w3schools.com/jsref/jsref_obj_regexp.asp

Examples:

matches(.*) will match any string, even the empty string

matches(\d+) will match any integer (one or more digits)

matches([a-zA-Z0-9]+) will match any alpha-numeric string

Numeric Constraints

Numeric constraints are only to be used with integer and double types.

lt(<number>) - less than; lte(<number>) - less than or equal to

gt(<number>) - greater than; gte(<number>) - greater than or equal to

equals(<number>) - equal to (only for numeric values)

Examples:

lt(3)

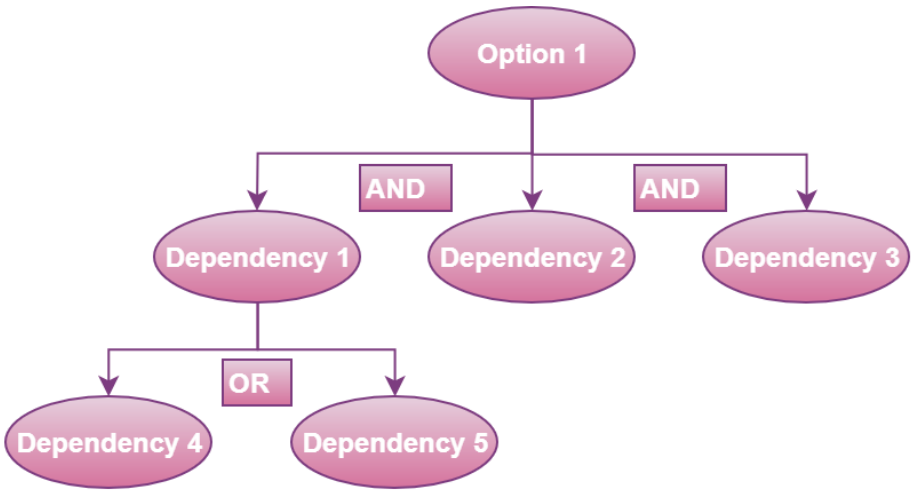
gte(1.1)

equals(0.01)

Complex Dependencies

By defining logical dependencies and constraints, a component developer can tell the system which options to use depending on the user-specified values of other options. To define the dependency, we need the tuple { logical operator (AND or OR), dependent option name, independent option name, constraint, negation }. Multiple constraints defined for a single dependent option can create more meaningful logic,

e.g. show Option 1 if (Dep1 AND Dep2 AND Dep3) where Dep1 = (Dep4 OR Dep5)



Dependency Examples

Example 1. Simple case: if (OPTION_Y matches (SOME_VALUE)), then provide the user with OPTION_X

OPTION_X is the dependent option.

OPTION_Y is the independent option (dependsOn attribute).

If the constraint resolves to true, then the interface will display OPTION_X.

We can ignore the type for now (disjunctive) because there is only one constraint to resolve.

```
<xs:complexType name="option_dependency">
     <xs:choice>
            <xs:element type="disjunctive" name="dependency1"
                  ls:dependentOption="OPTION_X" ls:dependsOn="OPTION_Y"
                        ls:constraint="matches(SOME_VALUE)" />
     </xs:choice>
</xs:complexType>
```

Example 2. Simple case with negation: if (NOT (OPTION_Y matches (SOME_VALUE))), then provide the user with OPTION_X

OPTION_X is the dependent option.

OPTION_Y is the independent option (dependsOn attribute).

If the constraint resolves to false when a negation is used, then it will be considered true.

If the constraint resolves to true when a negation is used, then it will be considered false.

We can still ignore the "type" attribute here.

```
<xs:complexType name="option_dependency">
     <xs:choice>
            <xs:element type="disjunctive" name="dependency1"
                  ls:dependentOption="OPTION_X" ls:dependsOn="OPTION_Y"
                        ls:constraint="matches(SOME_VALUE)"
                  ls:negation="true" />
     </xs:choice>
</xs:complexType>
```

Example 3. Combining disjunctions: if (OPTION_Y matches (SOME_VALUE) OR OPTION_Z matches (OTHER_VALUE)), then provide the user with OPTION_X

Disjunctions use the "OR" operand to combine the constraint results. If any one of the constraints are met, then the user is provided with the dependent option.

OPTION_X is the dependent option.

OPTION_Y is the independent option (dependsOn attribute).

If one or more of the constraints resolve to true, then provide the user with OPTION_X.

```
<xs:complexType name="option_dependency">
    <xs:choice>
        <xs:element type="disjunctive" name="dependency1"
            ls:dependentOption="OPTION_X" ls:dependsOn="OPTION_Y"
                ls:constraint="matches(SOME_VALUE)" />

        <xs:element type="disjunctive" name="dependency2"
            ls:dependentOption="OPTION_X" ls:dependsOn="OPTION_Z"
                ls:constraint="matches(OTHER_VALUE)" />
    </xs:choice>
</xs:complexType>
```

Example 4. Combining conjunctions: if (OPTION_Y matches (SOME_VALUE) AND OPTION_Z matches (OTHER_VALUE)), then show OPTION_X

Conjunctions use the "AND" operand to combine the constraint results. If all of the constraints are met, then the user is provided with the dependent option.

OPTION_X is the dependent option,

OPTION_Y is the independent option (dependsOn attribute),

If all of the constraints resolve to true, then display OPTION_X.

```
<xs:complexType name="option_dependency">
    <xs:choice>
        <xs:element type="conjunctive" name="dependency1"
            ls:dependentOption="OPTION_X" ls:dependsOn="OPTION_Y"
                ls:constraint="matches(SOME_VALUE)" />

        <xs:element type="conjunctive" name="dependency2"
            ls:dependentOption="OPTION_X" ls:dependsOn="OPTION_Z"
                ls:constraint="matches(OTHER_VALUE)" />
    </xs:choice>
</xs:complexType>
```

Example 5. Dependency Trees

Complex options can be created by crafting dependencies which are dependent on other dependencies or constraints.

if ((OPTION_Y matches (SOME_VALUE) AND OPTION_Z matches (SOME_VALUE))

OR NOT (OPTION_Y matches (SOME_VALUE))),

then show OPTION_X

```
<xs:complexType name="option_dependency">
    <xs:choice>
        <!-- define "matches" dependencies for SOME_VALUE -->

        <xs:element type="conjunctive" name="dependency1"
            ls:dependentOption="dependency1" ls:dependsOn="OPTION_Y"
                ls:constraint="matches(SOME_VALUE)" />

        <xs:element type="conjunctive" name="dependency2"
            ls:dependentOption="dependency2" ls:dependsOn="OPTION_Z"
                ls:constraint="matches(SOME_VALUE)" />

        <!-- define first part of logical dependency -->

        <xs:element type="conjunctive" name="dependency5"
            ls:dependentOption="dependency5" ls:dependsOn="dependency1"
                ls:constraint="true" />

        <xs:element type="conjunctive" name="dependency6"
            ls:dependentOption="dependency5" ls:dependsOn="dependency2"
                ls:constraint="true" />

        <!-- define second part of the logical dependency;
            Take note of the ls:negation attribute. -->

        <xs:element type="conjunctive" name="dependency7"
            ls:dependentOption="dependency7" ls:dependsOn="dependency1"
                ls:negation="true" ls:constraint="true" />

        <!-- Combine two complex conjunctive statements into the highest level
        disjunction -->

        <xs:element type="disjunctive" name="dependency9"
            ls:dependentOption="OPTION_X" ls:dependsOn="dependency5"
                ls:constraint="true" />

        <xs:element type="disjunctive" name="dependency10"
            ls:dependentOption="OPTION_X" ls:dependsOn="dependency7"
                ls:constraint="true" />

    </xs:choice>
</xs:complexType>
```

Example 6. Another complex example

This example combines two conjunctive statements into a disjunctive clause.

if ((OPTION_Y matches (SOME_VALUE) AND OPTION_Z matches (SOME_VALUE))

OR (OPTION_Y matches (OTHER_VALUE) AND OPTION_Z matches (OTHER_VALUE))),

then show OPTION_X

```
<xs:complexType name="option_dependency">
    <xs:choice>
        <!-- define "matches" dependencies for SOME_VALUE -->

        <xs:element type="conjunctive" name="dependency1"
            ls:dependentOption="dependency1" ls:dependsOn="OPTION_Y"
                ls:constraint="matches(SOME_VALUE)" />

        <xs:element type="conjunctive" name="dependency2"
            ls:dependentOption="dependency2" ls:dependsOn="OPTION_Z"
                ls:constraint="matches(SOME_VALUE)" />

        <!-- define "matches" dependencies for OTHER_VALUE -->

        <xs:element type="conjunctive" name="dependency3"
            ls:dependentOption="dependency3" ls:dependsOn="OPTION_Y"
                ls:constraint="matches(OTHER_VALUE)" />

        <xs:element type="conjunctive" name="dependency4"
            ls:dependentOption="dependency4" ls:dependsOn="OPTION_Z"
                ls:constraint="matches(OTHER_VALUE)" />

        <!-- define conjunctive group for OPTION_Y, OPTION_Z, SOME_VALUE -->

        <xs:element type="conjunctive" name="dependency5"
            ls:dependentOption="dependency5" ls:dependsOn="dependency1"
                ls:constraint="true" />

        <xs:element type="conjunctive" name="dependency6"
            ls:dependentOption="dependency5" ls:dependsOn="dependency2"
                ls:constraint="true" />

        <!-- define conjunctive group for OPTION_Y, OPTION_Z, OTHER_VALUE -->

        <xs:element type="conjunctive" name="dependency7"
            ls:dependentOption="dependency7" ls:dependsOn="dependency3"
                ls:constraint="true" />

        <xs:element type="conjunctive" name="dependency8"
            ls:dependentOption="dependency7" ls:dependsOn="dependency4"
                ls:constraint="true" />

        <!-- Combine two complex conjunctive statements into the highest level
        disjunction -->

        <xs:element type="disjunctive" name="dependency9"
            ls:dependentOption="OPTION_X" ls:dependsOn="dependency5"
```

```
                    ls:constraint="true" />

            <xs:element type="disjunctive" name="dependency10"
                  ls:dependentOption="OPTION_X" ls:dependsOn="dependency7"
                    ls:constraint="true" />

    </xs:choice>
</xs:complexType>
```

Program Integration

A component can run any number of programs, but the bootstrap program is the one which is explicitly run by the platform. The bootstrap will then run any subsequent programs (procedure calls, file manipulation, etc.) before it returns control to the component wrapper.

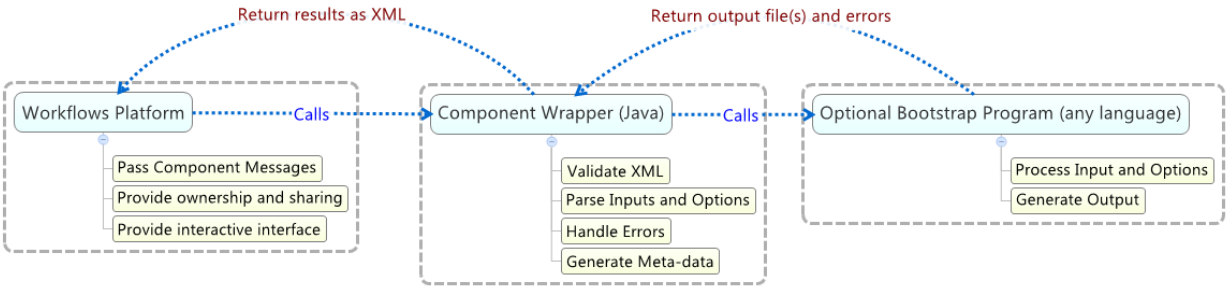


Figure 13 The Workflows call stack.

Bootstrap Program

To ease portability, each component contains a **build.properties** file which specifies the location of the interpreter (if one is needed) and the location of the component's bootstrap program. Interpreted languages, like Python, R, and Ruby, must include the interpreter and program paths. Here is an example. RScript.exe is the interpreter that runs the R program (or script), DataShop-AFM.R.

```
component.interpreter.path=c:/R-3.2.1/bin/Rscript.exe
component.program.path=program/DataShop-AFM.R
```

Compiled programs that do not require an interpreter only require that the program path be specified.

```
component.program.path=program/trainhmm.exe
```

The default heap (1024 MB) size can be overridden in the same build.properties file:

```
initial.heap.mb=2048
```

Once the Bootstrap Program is loaded, it has full control and can execute additional programs before returning its output (files or stdout) to the Component Wrapper. Any messages printed to the error stream while the program is running are also passed back to the Component Wrapper.

R Caveats

A program is not permitted to write data to the stdout stream, as that stream is used to pass the XML output to the system. For this reason, you may have to suppress R messages for certain libraries.

Turn off library loading messages:

```
echo <- false
```

Suppress messages when loading a library:

```
suppressMessages(library(lme4))
```

Suppress warnings for a given command:

```
suppressWarnings(fwrite(origFile, file=outputFile3,sep="\t", quote=FALSE, na=""))
```

Component Wrapper (Java)

An essential piece to integrating new programs is the Component Wrapper. It is a standalone program written in Java which is called during workflow execution. Then, if a bootstrap program is defined, it will execute it. The Java wrapper automates schema validation, pre- and post-condition checks, argument passing, error handling, user feedback, and program execution. While the bootstrap program can be written in any language, the Component Wrapper must be in Java to utilize the existing API.

The bootstrap program will be executed by **runExternalMultipleFileOutput()**. All **files** and **simple data types** are passed to the bootstrap program via command-line arguments. A working directory will be created for each component and returned by the method. Once your bootstrap program finishes processing the data, then any files or simple data types added to its output will be passed along to the next component(s) in the workflow.

In the following example, we define a program which uses **runExternalMultipleFileOutput**. The method runs the bootstrap program and returns a single directory which contains all of the files generated by the custom code. In the example, the bootstrap program is expected to generate two files, image1.gif and text1.txt. It then adds these files to their appropriate output nodes—the file types correspond directly to the Output definitions in the component's XSD file as discussed in the section on The Component Schema.

Override the runComponent method in the Java wrapper to call our bootstrap program as declared in the build.properties file discussed earlier.

```
@Override
protected void runComponent() {
    // Run the program and add the files it generates to the component output.
    File outputDirectory = this.runExternalMultipleFileOuput();
    // Attach the output files to the component output with addOutputFile(..>)
    if (outputDirectory.isDirectory() && outputDirectory.canRead()) {
        File file0 = new File(outputDirectory.getAbsolutePath() + "/text1.txt");
        File file1 = new File(outputDirectory.getAbsolutePath() + "/image1.gif");

        if (file0 != null && file0.exists() && file1 != null && file1.exists()) {

            Integer nodeIndex0 = 0;
            Integer fileIndex0 = 0;
            String label0 = "text";
            this.addOutputFile(file0, nodeIndex0, fileIndex0, label0);

            Integer nodeIndex1 = 1;
            Integer fileIndex1 = 0;
            String label1 = "image";
            this.addOutputFile(file1, nodeIndex1, fileIndex1, label1);

        } else {
            this.addErrorMessage("The expected output files could not be found.");
        }

    }

    // Send the component output back to the workflow.
    System.out.println(this.getOutput());
}
```

Bootstrap Program Parameters

When the *component* is executed, its inputs and option values are passed to the bootstrap program as command-line arguments. The argument labels are defined in your XML Schema Definition's options and inputs. Each bootstrap program is provided a set of tagged command-line arguments for easy parsing. You can then read these parameters using your preferred programming language's standard argument handling capabilities.

Example Parameters

```
/path/to/MyComponent/program/analysis.exe
        -programDir /path/to/MyComponent
        -workingDir /path/to/MyComponent_WorkingDirectory
        -xValidationFolds 100
        -someOtherOption value2
        -file0 input.txt
```

Example Parameter Summary

programDir – the base directory of the component, i.e. the directory containing all of the component code

workingDir – the working directory for each component in each workflow, e.g. /data/workflows/1/ImportComponent3/ -- the working directory is the CWD used to run the external program

xValidationFolds – an example option

someOtherOption – an example option

file0 – the relative path to the input file on Node 0

file1 – the relative path to the input file on Node 1

…

The **simple data types** are passed as strings while **files** are passed as absolute file paths.

Note: Components are executed on Linux so programs should expect absolute file paths in accordance with this fact. For more information on the Linux file system and what to expect, see https://help.ubuntu.com/community/LinuxFilesystemTreeOverview. Almost all programs can simply ignore this requirement unless they utilize the programDir or workingDir arguments in some platform-specific way, an unlikely event.

Simple Input Parameters

For simple data types, the **name** attribute prefixes the argument as a tag, e.g. -xValidationFolds.

```
<xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />
```

```
The XSD element is converted to the command-line argument:
```

```
-xvalidationfolds 10
```

File Input Parameters

- Components may have 0 or more input nodes but only 1 file per input node
- Files passed to the bootstrap program are tagged, e.g. -file0 someFile.txt -file1 anotherFile.txt
- -file0 is the file passed to the first input node, -file1 is the second, etc.

```
<xs:complexType name="InputType">
    <xs:sequence>
      <xs:element name="input0" type="InputDefinition0" minOccurs="0" />
      <xs:element name="input1" type="InputDefinition1" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
```

Parameter Order

Parameters will always follow this order

```
[1] –programDir /path/to/WorkflowComponents/GeneratePfaFeatures/
```

```
[2] -workingDir /path/to/workflows/16/Transform-1-x432172/output/
```

```
[3] data type options (alphabetical order), e.g. -model Default -nonce 123 –xValidationFolds 4
```

```
[4] input files (ordered by node index), e.g. -file0 somefile.txt –file1 someOtherFile.txt
```

Generating Output

A component outputs an XML entity which contains all the data needed by the workflows platform and other components. The XML is automatically generated once the component's **program** has completed. There are several options for generating component output. The most direct way is using the output stream. The **output stream** of your program is written to a file which can be added to the component output, if desired.

From Output Stream

The output stream of a program is written to a file that can be attached to the component output. Each language has a standard set of calls for writing to the output stream. In Java, this is done via the System.out.println statement. In C or C++, this is done via printf or cout statements.

In the following example, we see how to interact with the component output. The bootstrap program specified in the build.properties file is executed, and a file is automatically created from the program's standard output stream. Then, the file is attached to the component's output (as a file path), and then the component output is sent to the next component.

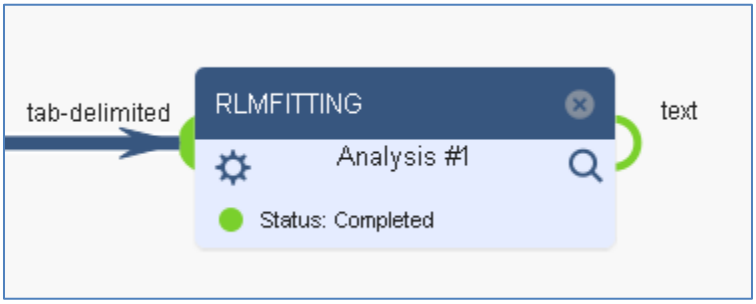


Figure 14 Component with a single file output.

```
@Override
protected void runComponent() {
    // Run your program. In this example, our program creates “Results.txt”, but you are allowed
   // to generate more than one file, as long as it is added via the addOutputFile method.
    this.runExternal();

    Integer nodeIndex = 0;
    Integer fileIndex = 0;
    String fileLabel = "text";

    this.addOutputFile(“Results.txt”, nodeIndex, fileIndex, fileLabel);
    // Send the component output back to the workflow.
    System.out.println(this.getOutput());
}
```

File Manipulation

The alternative way of generating file output is to simply write files from your external program to the "output" subdirectory of the working directory. Then, you can add the file(s) to the component output in the Java wrapper with a set of commands similar to those when using the standard output stream.

```
...
File outputDirectory = this.runExternalMultipleFileOuput();
File outputFile = outputDirectory + "/MyOutput.txt";
this.addOutputFile(outputFile, nodeIndex, fileIndex, fileLabel);
```

The above lines will run the external file and the file "MyOutput.txt" to the component output.

Any number of files can be added to the output so long as they are defined in the XML Schema Definition.

Summary of Methods

public void runExternal()

This method executes the component's program, passing all inputs and options to it via the command-line. The standard output stream of the program is automatically written to the file returned by this method.

public String runExternalMultipleFileOutput()

This method executes the component's bootstrap program declared in the build.properties file. When executed, all inputs and options are passed via the command-line to the bootstrap program. The program will then be expected to create one or more output files which are added to the component output.

public void addOutputFile(File file, Integer nodeIndex, Integer fileIndex, String label)

This method attaches an arbitrary file to the component's output. It requires

file – the output file
nodeIndex – the output node index (0 being the first output node)
fileIndex - more than one file may be attached to an output node but usually the relationship is 1 file to 1 output node
label – the file label; the most generic label is **file**, while more specific types include **student-step**, **transaction**, **zip**, or **image**. See *File Types* in **Appendix A**.

getOutput()

This method produces the program output and component meta-data required by the system to process the workflow. It should be written to the standard output stream at the end of the runComponent() method which is required by all components except Import components.

Accessing Options from within the Component Wrapper (Java)

You may want to access component options or inputs prior to the actual execution of the bootstrap program. You can access them via built-in method calls in Java. Otherwise, you may access these same options via the command-line arguments passed to the bootstrap program.

Example XSD Option Definition

```
<xs:complexType name="OptionsType">
  <xs:all >
```

```
    <xs:element type="xs:string" name="model" id="Model" default="Default" />
    <xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />
    <xs:element type="xs:double" name="weight" id="Weight" default="1.0" />
    <xs:element type="xs:boolean" name="useSampling" id="Use_Sampling" default="false" />
  </xs:all>
</xs:complexType>
```

Example Option Accessors

```
String model = getOptionAsString( "model" );
Integer xValidationFolds = getOptionAsInteger( "xValidationFolds" );
Double weight = getOptionAsDouble( "weight" );
Boolean useSampling = getOptionAsBoolean( "useSampling" );
```

Accessing Inputs from within the Java wrapper

```
protected File getAttachment(int nodeIndex, int fileIndex)

       nodeIndex – the input node index (0 is the first)
       fileIndex – the file index of the file associated with the given input node
```

Example XSD InputData Definition

```
xs:complexType name="InputData">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="xs:string" name="model"  />
        <xs:element type="xs:integer" name="xValidationCrossFolds"  />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

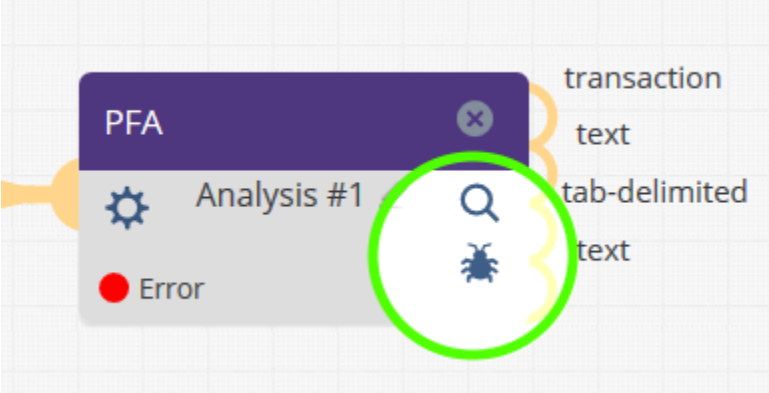
Example Input Accessors

```
File inputFile0 = getAttachment(nodeIndex, fileIndex);
File inputFile1 = getAttachment(1);
String model = getInputAsString(fileIndex, "model");
```

Generating Debug Info

From External Program

It might be tempting to write logging (debugging) statements to “stdout” from your program. However, when an external program is called, it must not use stdout or else it will corrupt the XML output generated by the Java wrapper. Instead, any files with the extension “.wfl” in the current working directory of the program are considered workflow log files and will be made available to the owner of the workflow when running components from the Workflow Editor.



R example

```
# Creates output log file (use .wfl extension if you want the file to be treated as
# a logging file (and hidden from everyone except the workflow owner)
clean <- file(paste("PFA-features-log.wfl", sep=""))
sink(clean,append=TRUE)
sink(clean,append=TRUE,type="message") # get error reports also
options(width=120)
# do work here
…
# Stop logging
sink()
sink(type="message")
```

From Java Wrapper

The Java wrapper already implements the log4j package. The following statements can be called from the wrapper, logging to WorkflowComponent.log, without affecting stdout.

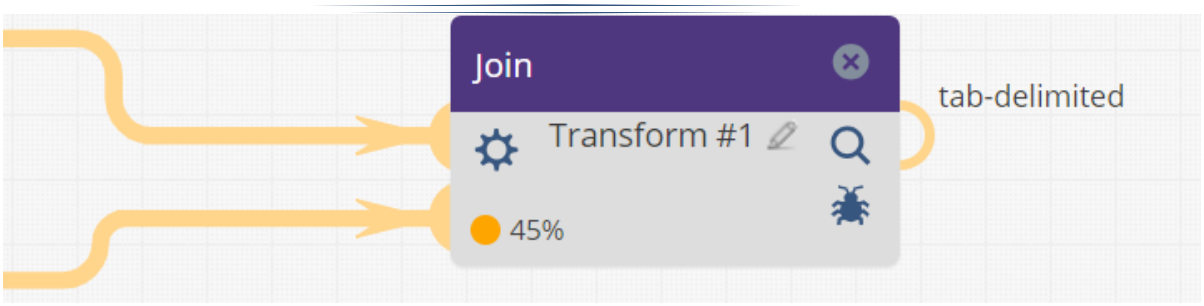
```
logger.info("This points to /datashop/workflow_components/MyTool/: " + this.getToolDir());

if (this.getOptionAsString("model") != null) {
        // If the option "model" is specified, print it to WorkflowComponent.log
        logger.info("The value of the model option is " + this.getOptionAsString("model"));
}
```

The logging levels in order of importance are trace, debug, info, warn, error, fatal. The system ultimately determines which level of logging is allowed. In general, logger.info, logger.warn, and logger.error are enabled on LearnSphere servers. For this reason, it is wise to use something like logger.trace when generating copious amounts of log data to minimize the server load while still allowing for detailed logging during local debugging. Reserve logger.info, logger.warn, and logger.error for logging messages that would help the end-user narrow down their problem without overloading them with information.

Component Progress Messages

While a component is running, it can communicate with the front-end small messages that indicate how much progress they have made like so:



In this example, the component has logged that it is "45%" complete. However, other short messages can also be logged besides percentages. If no progress messages are logged, the interface displays "Running" where the "45%" is, in the figure above.

Logging a Progress Message

To have your component display its progress, it must log a line to either the WorkflowComponent.log or [fileName].wfl (See previous section on Generating Debug Info). The messages that are displayed as progress take the form:

%Progress::@[YYYY-MM-DD HH:MM:SS]@[Message]

The logging message must begin with "%Progress::". This is followed by the time in the format abovet surrounded by "@". The last part is the actual message to display. In the example above, the log looked like:

%Progress::@2018-12-10 11:47:48@45%

The most recent one of these progress messages, judged by the date, is displayed on the component.

Java Progress Messages

The AbstractComponent class has a helper function to output progress messages. In the Java wrapper of your component, you can call this to display a progress message:

```
this.addComponentProgressMessage("Training the Model");
```

This calls a method that looks like:

```
public void addComponentProgressMessage(String message) {
   String COMPONENT_PROGRESS_PREPEND = "%Progress::";
   String timeStamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date());
   logger.info(COMPONENT_PROGRESS_PREPEND + "@" + timeStamp + "@" + message);
}
```

You can adapt this code above if your component is written in java but is not in the Java Wrapper of the component. If you are not using the Java wrapper, log these progress messages to a [filename].wfl log file.

R Progress Messages

Some example code for making progress updates in an R component is:

```
log_progress_message <- function(log_file, message) {
  time = encodeString(Sys.time())
  write(paste("%Progress::", "@", time, "@", message, sep=""), log_file, append=TRUE);
}
logFile <- paste(workingDir, "/row_remover.wfl", sep="")
log_progress_message(logFile, "45%")
```

The workingDir can be obtained from the command line arguments.

Python Progress Messages

Some example code for making progress updates in a Python component:

```
progressLogFilePath = args.workingDir + "/progress_log.wfl"
def progressMessage(message):
	f = open(progressLogFilePath, "a");
	now = datetime.datetime.now()
	progressPrepend = "%Progress::"
	f.write(progressPrepend + "@" + str(now) + "@" + message);
	f.close();
```

Error Handling

Error Stream

When a component executes a program, it reads from the *stderr* stream. If any error messages exist, they are passed pack through the wrapper and presented to the user on the workflows message pane. Most languages have their own functions for writing to the error stream—e.g., Java's *System.err.println* or the C++ object *std::cerr*.

Pre- and Post-condition Checks

The XSD is the primary driver for pre- and post-condition checks. If defined correctly, you can always expect a component's inputs and options to meet the criteria defined in the component schema. If any expected inputs are options are absent, or if one of the values does not match their designated type, then an error will be returned to the user and the component will exit before executing its program. For example, if "abc" is entered into an xs:integer type option, the user will receive the following feedback.

```
Previous run results - Analysis #1
cvc-datatype-valid.1.2.1: 'abc' is not a valid value for 'integer'.
```

In addition to type checks, **files** attached to a component (input or options) are automatically tested to ensure they exist and can be read. If not, the component will not run its program and will return an error to the user.

User-defined Tests

More advanced tests can be implemented by the user. User-defined tests can test input and options values. If any of the tests fail, the component will return a specific error before running the program. This can be useful

when the XSD restrictions are not enough. To create the user-defined test method in the Java wrapper, simply create and override the component's **test** method.

```
/**
 * The test() method is used to test the known inputs prior to running.
 * @return true if passing, false otherwise
 */
@Override
protected Boolean test() {
    Boolean passing = true;

    // Constrain the xValidationFolds options to be between 2 and 100.
    Integer xValidationFolds = this.getOptionAsInteger("xValidationFolds");

    if (xValidationFolds < 2 || xValidationFolds > 100) {

        // Add the error to the user interface
        this.addErrorMessage("Cross-validation folds " + xValidationFolds
            + " must be between 2 and 100, inclusive.");
        passing = false;
    }


     for (String err : errorMessages) {
         logger.error(err);
     }

    return passing;
}
```

Component Warnings

Warning Stream

When a component executes a program, it creates a WorkflowComponent.log file. Since the stdout stream is reserved by the system, programs are encouraged either to write logging and debugging information to a file (or files) with the extension .wfl which tells the system not to include the file as an output file. Instead, *.wfl files are treated as logging files which can be viewed by the owner of a workflow. Because debugging information may contain sensitive data, only the owner may see it.

Any **warning** messages will be given priority—these are messages which include the case-sensitive string “WARN:” or “WARNING:”. Programs can simply write the string to any *.wfl (workflow log) files. In the Java wrapper, one may issue the following statement: logger.warn("Some example warning message.");

Appendix A

Existing File Types

From TableTypes.xsd (as of 11/7/17)

```
  <xs:element name="file" type="FileContainer" />
  <xs:element name="text" substitutionGroup="file" />
  <xs:element name="image" substitutionGroup="file" />
  <xs:element name="audio" substitutionGroup="file" />
  <xs:element name="video" substitutionGroup="file" />
  <xs:element name="binary" substitutionGroup="file" />

  <xs:element name="tab-delimited" substitutionGroup="text" />
        <xs:element name="user-sess-map" substitutionGroup="tab-delimited" />
        <xs:element name="user-map" substitutionGroup="tab-delimited" />
        <xs:element name="resource-use" substitutionGroup="tab-delimited" />
        <xs:element name="outcome" substitutionGroup="tab-delimited" />
        <xs:element name="resource-use-to-outcome" substitutionGroup="tab-
delimited" />
        <xs:element name="student-step" substitutionGroup="tab-delimited" />
        <xs:element name="transaction" substitutionGroup="tab-delimited" />
        <xs:element name="student-problem" substitutionGroup="tab-delimited" />
        <xs:element name="model-values" substitutionGroup="tab-delimited" />
        <xs:element name="parameters" substitutionGroup="tab-delimited" />
        <xs:element name="gradebook" substitutionGroup="tab-delimited" />
        <xs:element name="correlation" substitutionGroup="tab-delimited" />
        <xs:element name="cronbachs-alpha" substitutionGroup="tab-delimited" />

  <xs:element name="analysis-summary" substitutionGroup="text" />
        <xs:element name="csv" substitutionGroup="text" />
        <xs:element name="sql" substitutionGroup="text" />
              <xs:element name="coursera-hash-mapping" substitutionGroup="text" />
<xs:element name="coursera-anonymized-forum" substitutionGroup="text" />
<xs:element name="coursera-anonymized-general" substitutionGroup="text" />
      <xs:element name="MOOCdb" substitutionGroup="file" />
<xs:element name="MOOCdb-features" substitutionGroup="tab-delimited" />
<xs:element name="longitudinal-features" substitutionGroup="tab-delimited" />
<xs:element name="MOOCdb-feature-description" substitutionGroup="tab-delimited" />
        <xs:element name="discoursedb" substitutionGroup="text" />
        <xs:element name="xml" substitutionGroup="text" />
              <xs:element name="inline-html" substitutionGroup="xml" />
              <xs:element name="html" substitutionGroup="xml" />
              <xs:element name="ctat-log" substitutionGroup="xml" />

      <xs:element name="tetrad-graph" substitutionGroup="tab-delimited" />
      <xs:element name="tetrad-knowledge" substitutionGroup="tab-delimited" />
      <xs:element name="tetrad-regression" substitutionGroup="tab-delimited" />
      <xs:element name="tetrad-im" substitutionGroup="tab-delimited" />

  <xs:element name="png" substitutionGroup="image" />
  <xs:element name="jpg" substitutionGroup="image" />
  <xs:element name="gif" substitutionGroup="image" />
  <xs:element name="psd" substitutionGroup="image" />
  <xs:element name="bmp" substitutionGroup="image" />
```

```
  <xs:element name="svg" substitutionGroup="image" />

  <xs:element name="compressed" substitutionGroup="binary" />
        <xs:element name="zip" substitutionGroup="compressed" />
        <xs:element name="bz2" substitutionGroup="compressed" />
        <xs:element name="gz" substitutionGroup="compressed" />
        <xs:element name="tar" substitutionGroup="compressed" />
        <xs:element name="_7zip" substitutionGroup="compressed" />

  <xs:element name="microsoft" substitutionGroup="binary" />
        <xs:element name="excel" substitutionGroup="microsoft" />
        <xs:element name="word" substitutionGroup="microsoft" />
        <xs:element name="powerpoint" substitutionGroup="microsoft" />

  <xs:element name="pdf" substitutionGroup="binary" />
```

Adding New File Types

File types are defined in "CommonSchemas/TableTypes.xsd" in the WorkflowComponents package. Any elements defined in a FileList must exist in the TableTypes.xsd file. Currently, adding a new type is straight-forward. Simply add it to one parent that best describes it, or create a new category entirely. As an example, let's add a new "tab-delimited" file type called "blackboard-quiz-scores". All we need to do is set the name and substitutionGroup.

```
...

<xs:element name="file" type="FileContainer" />

  <!-- All of the following are types of "file" elements.
       If the subheading you want doesn't exist, either create a new one with a unique name
       or use a more generic type, like "text" or "image". -->
    <xs:element name="text" substitutionGroup="file" />
      <xs:element name="tab-delimited" substitutionGroup="text" />
        <xs:element name="resource-use" substitutionGroup="tab-delimited" />
        <xs:element name="student-step" substitutionGroup="tab-delimited" />
        <xs:element name="blackboard-quiz-scores" substitutionGroup="tab-delimited" />

...
```

Appendix B

Lessons Learned

This appendix serves as a spot for documenting “lessons learned”. We ask that component developers update this section with notes about issues encountered during the course of developing new components. Corrections to the rest of the document should also be made, but if you find an issue you wish had been covered in the doc but isn’t, or debugging proved particularly tricky, please document those findings here so that others can benefit from your experience.

- The components should not write non-XML to standard output. The GUI interprets the component output as XML, assuming it is the component results. If the component writes to standard out, the GUI will fail to run the component and issue an XML parsing error.
- Many programming languages support writing to standard error. This stream is picked up by the workflows platform and will redirect all standard error streams to the user interface in the message window.