Mobyle Service Description Guide

Mobyle 1.0

Contents

Chapter 1

Introduction

Mobyle is a system which provides an access to different software elements, in order to allow users to perform bioinformatics analyses. Such software elements are called **services**, and belong to three distinct categories:

- **programs** describe software that are executed on the server side "command line" tools, and return results in the form of files,
- **workflows** describe compositions of successive and/or parrallel calls to other server-side services (programs and/or subworkflows), and also return results as files,
- **viewers** describe browser-embedded visualization components, such as applets, that allow users to visualize data (results or bookmarks) stored in the Mobyle server.

To publish such services on a Mobyle server, you need to install and describe them so that Mobyle offers the adequate user interface and executes them when required. Mobyle stores the description of each service in an XML document, a **service description**.

The formal description of the format of these XML files is stored in `relax-ng` and `schematron` formats, in the `*.rnc`, `*.rng`, and `*.sch` files, located in the `Schema` folder. The goal of this document is to describe the elements of this format so that you can add new tools or modify existing ones. Even if you do not intend to use them, it is strongly recommended to download the service descriptions maintained and distributed by the Institut Pasteur, which are used to illustrate the most complex elements of the service description format.

The releases of service descriptions are available on the public ftp server of the Institut Pasteur: ftp://ftp.pasteur.fr/pub/gensoft/projects/mobyle/.

- Programs-2.0.tgz or higher.
- Workflows-??.tgz or higher.
- Viewers-??.tgz or higher.

The service descriptions are maintained in 3 subprojects of the Mobyle forge:

- https://projets.pasteur.fr/projects/show/pasteur-programs
- https://projets.pasteur.fr/projects/show/pasteur-workflows
- https://projets.pasteur.fr/projects/show/pasteur-viewers

The development versions of the service descriptions are accessible from the public SVN Mobyle repository. With a command-line subversion client, type these lines:

- `svn co https://projets.pasteur.fr/svn/pasteur-programs/`
- `svn co https://projets.pasteur.fr/svn/pasteur-workflows/`
- `svn co https://projets.pasteur.fr/svn/pasteur-viewers/`

Chapter 2

Writing a program description

A program is a software:

- running on the server side,
- callable in a shell command line,
- returning some results.

XML files describe these softwares and make them usable by Mobyle. Hereinafter, “program description” stands for “XML description of such a software”.

A program description aims at capturing different kinds of information:

- how to invoke the program (a program wrapper),
- how to display options in submission forms or job results (a UI description),
- how to control values or parameters compatibilities (a “semantic” description).

The root element of a program description is the **program** tag. A program description is divided into two parts:

- the **head** element, described in section 2.1, contains general information about the program,
- the **parameters** element, described in section 2.2, contains the different input data, options and results of the program.

2.1 head

- **name** (*required*): name of the interface you are describing.
  - does not have to be identical to the actual invoked program name (but highly recommended).
  - has to be identical to the XML file name (minus its extension). E.g.: for the file `blast2.xml`, the **name** tag has to be `<name>blast2</name>`.
- **version**: version of the program you are interfacing, not of the XML description itself.
- **package** (*new in 1.0*): description of the package the program belongs to. Can contain all the elements defined in **head** (`name`, `version`, `doc`. . .).
- **doc**: documentation of the program split among the following elements:
  - **description**: brief description of the software, in text or XHTML. This description appears at the top of the submission form.
  - **reference**: scientific reference officially used to cite the program, in text or XHTML. In addition, you can provide a hyperlink to the users by specifying a DOI or a direct URL, respectively in the `doi` and `url` attributes.
  - **authors**: program authors who must be cited.
  - **doclink**: any URL pointing to useful documentation regarding the program. If a **doclink** is specified, a button "Help Pages" is displayed at the top of the web form:
    - with only one **doclink** element, the documentation is be available both as the "Help Pages" button and as a link at the bottom of the web form,
    - with several **doclink** elements, the "Help Pages" button sets the focus to the bottom of web form where he can choose among the different links.
  - **sourcelink**: (*new in 1.0*) any URL where the program can be found. Essentially used as documentation by server administrators.
  - **homepagelink**: (*new in 1.0*) the URL to the homepage of the software project.
  - **comment**: text describing the program with more details. This comment is shown by clicking on the red question mark beside the title.
- **category**: path within the classification of the programs which determines where the program name will be displayed in the program tree on the left panel. Thus, the **category** must be chosen carefully. The colon is used as separator between nodes of the classification.
  - if multiple **category** elements are specified for a service, the program will appear in each branch of the tree,
  - if none is provided, it will appear at the "top level".
- **env**: used to specify an environment variable that will be defined prior to launching any job for the program. Usually implemented with entities or `XInclude`. Examples:
  - to specify the location of blast databases: `<env name=‘‘BLASTDB’’>/usr/local/ncbi/db</env>`
  - to give the path of a binary different from the usual path:
    `<env name=‘‘PATH’’>/path/to/my/binary</env>`

how to include package information (excerpt from **dnadist** description)

```
<program>
  <head>
    <name>dnadist</name>
    <package>
      <name>phylip</name>
      <version>3.67</version>
      <doc>
        <title>phylip</title>
        <description>
          <text lang=”en”>PHYLIP is a package of programs for inferring phylogenies.</text>
        </description>
        <homepagelink>http://evolution.gs.washington.edu/phylip.html</homepagelink>
        <sourcelink>http://evolution.gs.washington.edu/phylip/getme.html</sourcelink>
        <authors>Felsenstein Joe</authors>
        <reference>Felsenstein, J. 1993. PHYLIP (Phylogeny Inference Package)
        version 3.5c.
        Distributed by the author. Department of Genetics,
        University of Washington, Seattle.</reference>
        <reference>Felsenstein, J. 1989. PHYLIP -- Phylogeny Inference Package
        (Version 3.2).
        Cladistics 5: 164-166.</reference>
        <doclink>http://bioweb2.pasteur.fr/docs/phylip/phylip.html</doclink>
      </doc>
    </package>
    <doc>
      <title>dnadist</title>
      <description>
        <text lang=”en”>Compute distance matrix from nucleotide sequences</text>
      </description>
      <doclink>http://bioweb2.pasteur.fr/docs/phylip/doc/dnadist.html</doclink>
      <comment>
        <text lang=”en”>This program uses nucleotide sequences to compute
        a distance matrix, under four different models of nucleotide substitution.
        It can also compute a table of similarity between the nucleotide
        sequences. The distance for each pair of species estimates the total
        branch length between the two species, and can be used in the distance
        matrix programs FITCH, KITSCH or NEIGHBOR.
        </text>
      </comment>
    </doc>
    <category>phylogeny:distance</category>
  </head>
  [...]
```

2.1.1 XInclude to include external or common information

It is possible to include site-dependent information (such as the above-mentioned **env** tag) by including external pieces of XML code. Although XML entities can achieve the same goal, the recommended way is to use `XInclude` as it provides error-handling mechanisms not available with XML entities. XInclude can be used to:

- refactor some parts of a set of descriptions,
- specify site-dependent variables.

For example, if you have a set of interfaces which wrap different programs from the same package, you can include common package information in an entity that you include in the different XML files. Just make sure that:

1. the `XInclude` elements are correctly formatted
2. the `XInclude` namespace is declared somewhere.

Example: how to include databank (excerpt from `golden` description)

```
<parameters XMLns:xi=”http://www.w3.org/2001/XInclude”>
  <parameter ismandatory=”1” issimple=”1” ismaininput=”1”>
    <name>db</name>
    <prompt lang=”en”>Database</prompt>
    <type>
      <datatype>
        <class>Choice</class>
      </datatype>
    </type>
    <vdef>
      <value>null</value>
    </vdef>
    <xi:include href=”../../Local/Services/Programs/Entities/goldendb.xml”>
      <xi:fallback>
        <vlist>
          <velem undef=”1”>
            <value>null</value>
            <label>Choose a database</label>
          </velem>
        </vlist>
      </xi:fallback>
    </xi:include>
    <format>
      <code proglang=”perl”>” $db:”</code>
      <code proglang=”python”>” ” + db + ”:”</code>
    </format>
    <argpos>2</argpos>
  </parameter>
```

This mechanism is also used to share package information with other program descriptions belonging to the same package. In the following example, the `Entities` directory (same level as `Programs`) contains the `phylip_package.xml` file shared by all phylip programs (`dnadist`, `dnapars`, `protdist`, `protpars`...)

Example: how to share package information (excerpt from `dnadist` program description)

```
<head>
  <name>dnadist</name>
  <xi:include XMLns:xi="http://www.w3.org/2001/XInclude"
      href="Entities/phylip_package.xml"/>
  <doc>
    <title>dnadist</title>
    <description>
      <text lang="en">Compute distance matrix from nucleotide sequences</text>
    </description>
    <doclink>http://bioweb2.pasteur.fr/docs/phylip/doc/dnadist.html</doclink>
    <comment>
      <text lang="en">This program uses nucleotide sequences to compute a distance matrix,
       under four different models of nucleotide substitution. It can also
       compute a table of similarity between the nucleotide sequences.
       The distance for each pair of species estimates the total branch length
       between the two species, and can be used in the distance matrix programs
       FITCH, KITSCH or NEIGHBOR.</text>
    </comment>
  </doc>
  <category>phylogeny:distance</category>
</head>
```

Example: how to share package information (excerpt from phylip package description)

```
<package>
  <name>phylip</name>
  <version>3.67</version>
  <doc>
    <title>phylip</title>
    <description>
      <text lang="en">PHYLIP is a package of programs for inferring phylogenies.</text>
    </description>
    <homepagelink>http://evolution.gs.washington.edu/phylip.html</homepagelink>
    <sourcelink>http://evolution.gs.washington.edu/phylip/getme.html</sourcelink>
    <authors>Felsenstein Joe</authors>
    <reference>Felsenstein, J. 1993. PHYLIP (Phylogeny Inference Package) version 3.5c.
     Distributed by the author.
     Department of Genetics, University of Washington, Seattle.</reference>
    <reference>Felsenstein, J. 1989. PHYLIP -- Phylogeny Inference Package (Version 3.2).
     Cladistics 5: 164-166.</reference>
    <doclink>http://bioweb2.pasteur.fr/docs/phylip/phylip.html</doclink>
  </doc>
</package>
```

2.2 Parameters

The **parameters** element is a direct child of:

- either the **program** element: it includes all the defined **parameter**s for the program,
- or the **paragraph** element: it defines the **parameter**s of a given **paragraph**.

It can include:

- **parameter** elements,
- nested **paragraph** elements.

2.3 Paragraph

The **paragraph** element has two functions:

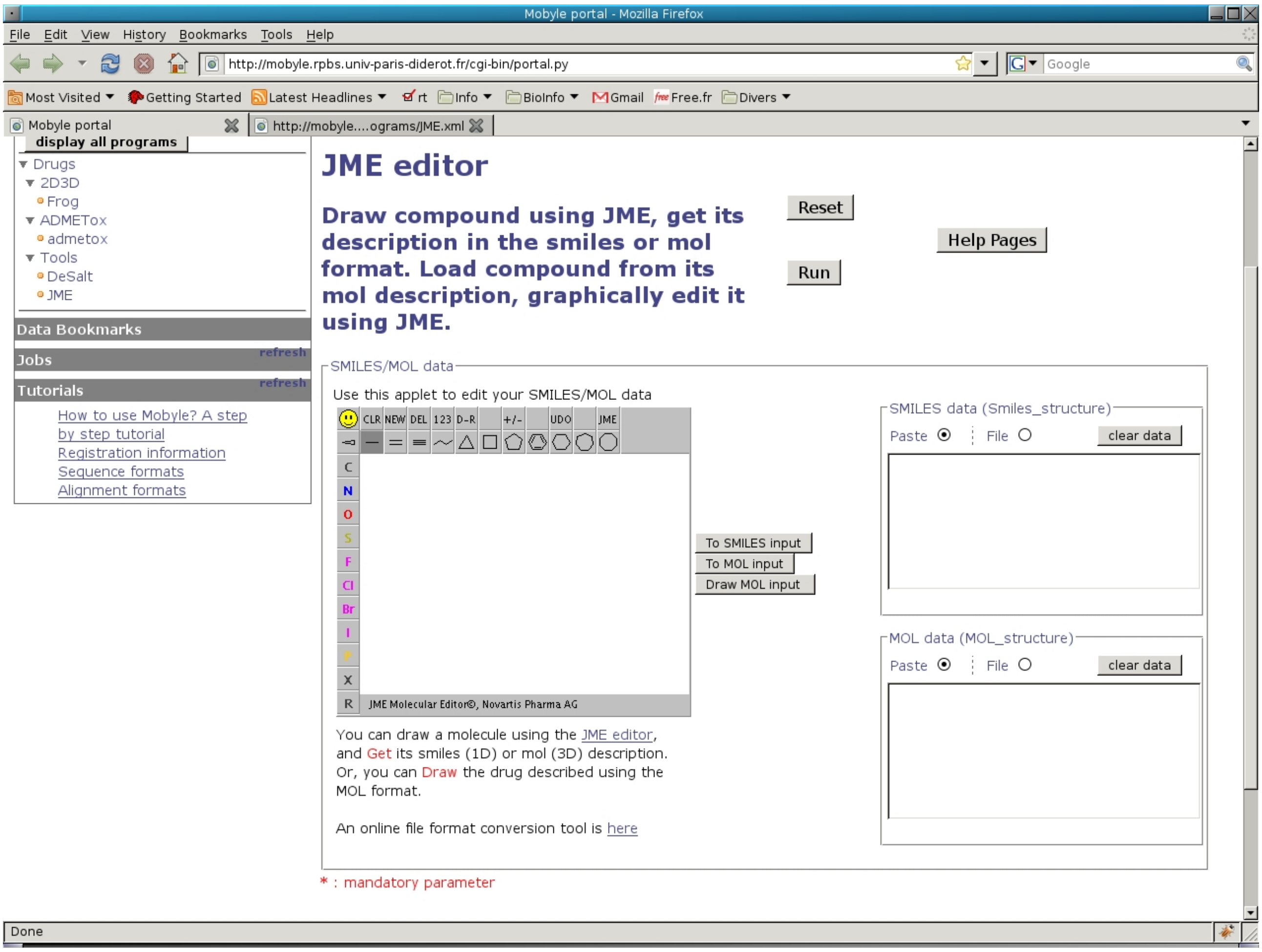
1. it can group the evaluation of different **parameters**, with respect to a set of **precond**itions for instance,

2. it also groups visually these **parameter**s in the form.

Thus, a **paragraph** is both a set of **parameter**s – visually speaking – in the submission form and a way to share some properties such as:

- **argpos** elements (see 2.4.2) and **precond** elements (see 2.4.2).

- **layout** which allows to override the default display of the **parameter**s belonging to a **paragraph**. The default disposition of the **parameter**s is a vertical succession, where form **parameter**s and job results are laid out in the same order as in the program description. The **layout** tag allows to define horizontal groups (with the **hbox** tag) and vertical groups (**vbox** tag) in a recursive structure. Each group had a **box** element that refers to the corresponding **parameter** using its **name**. See the JME description below.



JME editor

Draw compound using JME, get its description in the smiles or mol format. Load compound from its mol description, graphically edit it using JME.

SMILES/MOL data

Use this applet to edit your SMILES/MOL data

You can draw a molecule using the JME editor, and Get its smiles (1D) or mol (3D) description. Or, you can Draw the drug described using the MOL format.

An online file format conversion tool is here

* : mandatory parameter

Figure 2.1: JME interface

excerpt from JME XML description

```
<paragraph>
  <name>inputs</name>
  <prompt>SMILES/MOL data</prompt>
  <parameters>
    <parameter>
      <name>jme_applet</name>
      <prompt lang="en">Use this applet to edit your SMILES/MOL data</prompt>
      <type>
        <datatype>
          <class>String</class>
        </datatype>
      </type>
      <interface>
         <!-- To improve the readability of the example,
              a part of the code has been removed.
              The complete code is available at
              http://mobyle.rpbs.univ-paris-diderot.fr/data/programs/JME.xml
         -->
      </interface>
    </parameter>
    <parameter>
      <name>smiles_input</name>
      <prompt lang="en">SMILES data</prompt>
      <type>
        <datatype>
          <class>Smiles_structure</class>
          <superclass>AbstractText</superclass>
        </datatype>
      </type>
   </parameter>
   <parameter>
     <name>mol_input</name>
     <prompt lang="en">MOL data</prompt>
     <type>
       <datatype>
         <class>MOL_structure</class>
         <superclass>AbstractText</superclass>
       </datatype>
     </type>
   </parameter>
 </parameters>
 <layout>
   <hbox>
     <box>jme_applet</box>
     <layout>
       <vbox>
         <box>smiles_input</box>
         <box>mol_input</box>
       </vbox>
     </layout>
   </hbox>
 </layout>
</paragraph>
```

2.4 Parameter

The **parameter** element is an essential piece of the program description. It describes and stores:

- the options,
- the inputs,
- and the outputs

of a program, used to build the command line and retrieve the results.

2.4.1 Parameter attributes

The attributes of the **parameter** element are:

- **ismandatory**: the **parameter** must be specified by the user, pointed out by a * next to its prompt in the interface. Special case: if the **parameter** is mandatory and if a **precond** is defined (see below), the **parameter** becomes mandatory only if the result of the **precond**ition is true. These parameters are indicated by a *.

- **isout**: the **parameter** is produced by the program. It is used to retrieve the results. Mobyle systematically generates 2 files:

  - `[program name].out`, corresponding to the standard output stream,
  - `[program name].err`, corresponding to the standard error stream

  These two implicit results are automatically shown to the user in the result page if they are not empty and are typed as “Text”.

- **isstdout**: to provide a more explicit or detailed description of the standard output, instead of using the **isout** attribut, define a **parameter** element with the desired type and pass it as the value of the attribute **isstdout**. **isstdout** and **isout** are mutually exclusive. See **parameter** golden_out in `golden.xml` below.

  standard output **parameter** from `golden.xml`

```
<parameter isstdout="1">
  <name>golden_out</name>
  <prompt lang="en">Sequence</prompt>
  <type>
    <biotype>Protein</biotype>
    <biotype>Nucleic</biotype>
    <datatype>
      <class>Sequence</class>
    </datatype>
  </type>
  <filenames>
    <code proglang="perl">"golden.out"</code>
    <code proglang="python">"golden.out"</code>
  </filenames>
</parameter>
```

- **ishidden**: hides the **parameter** from the user, as opposed to the input parameters shown to the user in the submission form, or to the results captured by the output parameters shown in the job results. When adding a **parameter** just to control another one, conceal it from the user by setting the attribute **ishidden** to 1. The value of hidden parameters can't be set by the user. See **parameter** rateAll in `seqgen.xml` below.

hidden **parameter**

```
<parameter ishidden="1">
  <name>rateAll</name>
  <prompt lang="en">Rates</prompt>
  <type>
    <datatype>
      <class>Float</class>
    </datatype>
  </type>
  <precond>
    <code proglang="perl">$rate1  and $rate2 and $rate</code>
    <code proglang="python">rate1 is not None and rate2 is not None and rate3 is not
        None</code>
  </precond>
  <format>
    <code proglang="perl">" -c $rate1 $rate2 $rate3"</code>
    <code proglang="python">" -c %f %f %f " %(rate1,rate2,rate3)</code>
  </format>
</parameter>
```

2.4.2 Parameter subelements

- **name** (*required*): mainly used inside Mobyle to refer to this parameter but not seen by the user in the interface. The **name**:
  - has to be unique among all parameters and paragraphs,
  - must be a valid python variable name (cannot begin by a number . . .),
  - cannot be the name of a javascript property of the HTMLFormElement object, to avoid unexpected behaviors in the portal. Such names include **name**, **nodeName**, **length**, etc. For a complete listing of the forbidden values, please refer to the `mobyle.sch` file, located in the `Schema` folder.
- **prompt**: the human-readable label of the parameter that the user can see in the form
- **argpos** that specifies the position of the parameter in the command line.
  - by convention the **command** has its **argpos** set to 0,
  - if **argpos** is not specified at the **parameter** level, takes the **argpos** of the immediate upper level (**paragraph**),
  - required only if the order of parameters matters.
- a **precond** which allows to specify under which conditions the **parameter** must be evaluated. All **precond**itions are evaluated starting with the **precond** of the outermost **paragraph** up to the **precond** of the innermost nested **paragraph**, and finally ending with the **precond** of the **parameter**. This mechanism allows to refactor **precond**ition code.
- **format**: the inside code is evaluated to generate the command line. Mobyle uses python code but playMoby (`http://lipm-bioinfo.toulouse.inra.fr/biomoby/playmoby/`) uses perl code. By default, the result of the code evaluation is used as part of the command line. To write it in a file, in particular to simulate the interactive behavior of some programs such as some from the Phylip suite, use the element **paramfile**. See `pars.xml`, `protdist.xml`, `fitch.xml`, . . .
- **ctrl**: used to specify additional constraints on values provided by the user such as being an interger, being odd, not excessing a maximum . . . . Upon execution, the **code** is evaluated and the corresponding error **message** is displayed to the user if the result is False. See parameter identity in `cons.xml` below.
- **vdef**: the default value for the **parameter**, mandatory for **vlist** and **flist** (see 2.5.1).

2.4.3 Retrieving results

A result from a program is a parameter with:

- the **isout/isstdout** attribute set to 1,
- a **filenames** element.

Unix masks are defined in the **filenames** element to create the mapping between one or more file names and the parameter. Use them to represent filenames that cannot be known statically or when the parameter must be mapped with several files.

For example, the toppred program can take a file with several FASTA protein sequences. In this case, toppred will produce one hydrophobicity file per sequence appearing in the input file. The name of each file will be the name of the corresponding sequence, suffixed with the `.hydro` extension. To retrieve these files, the Unix mask is : “*.hydro” (e.g. parameter hydrophobicity_files in `toppred.xml`).

control a parameter value

```
<parameter>
  <name>identity</name>
  <prompt lang=”en”>Required number of identities at a position (value greater than or
      equal to 0)</prompt>
  <type>
    <datatype>
      <class>Integer</class>
    </datatype>
  </type>
  <vdef>
    <value>0</value>
  </vdef>
  <format>
    <code proglang=”python”>(””, ” −identity=” + str(value))[value is not None and
        value!=vdef]</code>
  </format>
  <ctrl>
    <message>
      <text lang=”en”>Value greater than or equal to 0 is required</text>
    </message>
    <code proglang=”python”>value &gt;= 0</code>
  </ctrl>
  <argpos>4</argpos>
  <comment>
    <text lang=”en”>Provides the facility of setting the required number of identities at a
        site...</text>
  </comment>
</parameter>
```

Unix mask are used as follow:

- a Unix mask is defined in a **code** element,
- only 1 mask can be defined per **code** element,
- **filenames** element may have several children **code** elements

The code is evaluated before being used as Unix mask.

how to retrieve results

```
<parameter isout="1">
 <name>graphicfiles</name>
 <prompt lang="en">Graphic output files</prompt>
 <type>
    <datatype>
      <class>Picture</class>
      <superclass>Binary</superclass>
    </datatype>
    <card>0,n</card>
  </type>
  <precond>
    <code proglang="perl">$graph_output</code>
    <code proglang="python">graph_output == 1</code>
  </precond>
    <filenames>
      <code proglang="perl">*.$profile_format</code>
      <code proglang="python">'*.' + profile_format</code>
    </filenames>
</parameter>
```

The **comment** and **example** elements allow to generate in line help and example data for the parameter or paragraph, very useful as pieces of documentation to the user.

- When an help **comment** is available, a clickable "?" appears beside the prompt.
- When an **example** is available, a "use example data" link appears beside the parameter. By clicking on it, the user fills automatically the corresponding databox with this value.

2.5 Typing

Choosing the right type for a parameter is an essential point in program description authoring, as a lot of features are based on types. The typing influences:

- the interface display,
- the controls on user values assigned to the input parameters,
- the chaining possibilities between programs and data reusability.

In Mobyle, typing is "multidimensional", meaning that it's based on several fields:

- the **card** represents the cardinality, *i.e.* the number of distinct values that can be set for the parameter.
- the **biotype** describes the biological object (Nucleic , Protein, Drug . . .). It is not required as some parameters do not represent any biological object. The **biotype** values are free text labels, but to chain the data appropriately, they must be recognized so take special care on the spelling and case.
- the **datatype** describes the computing object.
- the **dataFormat** (*new in 1.0*) specifies the format of the data for this parameter. If the parameter is an input parameter, it lists the formats accepted by the program, hence the formats into which Mobyle must convert (if possible) an input parameter value. For instance, the data formats accepted by a Sequence datatype can be FASTA, EMBL, . . . Several **dataFormat** elements can be specified. If the parameter is an output one (**isout**="1" or **isstdout**="1") then the **dataFormat** element specifies

the format for the result. This information can be very useful in order to suggest chaining possibilities, so it's an important information. Sometimes the data format of a result cannot be predetermined but can be computed at runtime based on the values of other input parameters set by the user. There are two ways to express this:

1. to test the value of another parameter,
2. to reference directly a parameter in **dataFormat**,

The two following examples illustrate these possibilities:

dataFormat by testing the value of another parameter (excerpt from `muscle.xml`)

1.
```
<parameter>
  <name>outformat</name>
  <prompt lang=”en”>output format</prompt>
  <type>
    <datatype>
      <class>Choice</class>
    </datatype>
  </type>
  <vdef>
    <value>fasta</value>
  </vdef>
  <flist>
    <felem>
      <value>fasta</value>
      <label>fasta</label>
      <code proglang=”perl”>””</code>
      <code proglang=”python”>””</code>
    </felem>
    <felem>
      <value>html</value>
      <label>html</label>
      <code proglang=”perl”>” −html ”</code>
      <code proglang=”python”>” −html ”</code>
    </felem>
    <felem>
      <value>msf</value>
      <label>msf</label>
      <code proglang=”perl”>” −msf ”</code>
      <code proglang=”python”>” −msf ”</code>
    </felem>
    <felem>
      <value>phyi</value>
      <label>phylip</label>
      <code proglang=”perl”>” −phyi ”</code>
      <code proglang=”python”>” −phyi ”</code>
    </felem>
    <!−− it’s a clustalw with a muscle header which is not supported by squizz
       −−>
    <felem>
      <value>clw</value>
      <label>muscle format</label>
      <code proglang=”perl”>” −clw ”</code>
      <code proglang=”python”>” −clw ”</code>
    </felem>
    <felem>
      <value>clwstrict</value>
      <label>clustalw 1.81</label>
      <code proglang=”perl”>” −clwstrict ”</code>
      <code proglang=”python”>” −clwstrict ”</code>
    </felem>
  </flist>
  <comment>
    <text lang=”en”>fasta : Write output in Fasta format</text>
    <text lang=”en”>html  : Write output in HTML format</text>
    <text lang=”en”>msf   : Write output in GCG MSF format</text>
```

```
      <text lang="en">phylip    : Write output in Phylip (interleaved)
          format</text>
      <text lang="en">muscle    : Write output in CLUSTALW format with muscle
          header</text>
      <text lang="en">clustalw : Write output in CLUSTALW format with CLUSTALW
          1.81</text>
    </comment>
  </parameter>

  <parameter isstdout="1">
    <name>alignmentout</name>
    <prompt lang="en">Alignment</prompt>
    <type>
      <datatype>
        <class>Alignment</class>
      </datatype>
      <dataFormat>
        <test param="outformat" eq="fasta">FASTA</test>
        <test param="outformat" eq="msf">MSF</test>
        <test param="outformat" eq="phyi">PHYLIPI</test>
        <test param="outformat" eq="clwstrict">CLUSTAL</test>
        <test param="outformat" eq="clw">MUSCLE</test>
      </dataFormat>
    </type>
    <precond>
      <code proglang="perl">$outformat =~ /^(fasta|msf|phyi|clwstrict|clw)$/
          </code>
      <code proglang="python">outformat in [ 'fasta' , 'msf' , 'phyi' ,
          'clwstrict' , 'clw'] and outfile is None</code>
    </precond>
    <filenames>
      <code proglang="perl">"muscle.out"</code>
      <code proglang="python">"muscle.out"</code>
    </filenames>
  </parameter>
```

The user specifies the format `muscle` must use to output the alignement result with the first parameter “outformat”. The second parameter “alignmentout” captures the alignment result. The **dataFormat** of “alignmentout” is set by testing the value of the parameter “outformat”. To test a value we have a set of operators:

- **eq**: equal.
- **ne**: not equal.
- **lt**: less than.
- **gt**: greater than.
- **le**: less or equal than.
- **ge**: greater or equal than.

dataFormat by referencing another parameter (excerpt from `clustalw-multialign.xml`)

```
2.    <parameter>
        <name>outputformat</name>
        <prompt lang="en">Output format (-output)</prompt>
        <type>
          <datatype>
            <class>Choice</class>
          </datatype>
        </type>
        <vdef>
          <value>null</value>
        </vdef>
        <vlist>
          <velem undef="1">
            <value>null</value>
            <label>CLUSTAL</label>
          </velem>
          <velem>
```

```
      <value>FASTA</value>
      <label>FASTA</label>
    </velem>
    <velem>
      <value>GCG</value>
      <label>GCG</label>
    </velem>
    <velem>
      <value>GDE</value>
      <label>GDE</label>
    </velem>
    <velem>
      <value>PHYLIP</value>
      <label>PHYLIP</label>
    </velem>
    <velem>
      <value>PIR</value>
      <label>PIR</label>
    </velem>
    <velem>
      <value>NEXUS</value>
      <label>NEXUS</label>
    </velem>
  </vlist>
  <format>
    <code proglang="perl">(defined $value ) ? " -output=$value" : ""</code>
    <code proglang="python">( "" , " -output=" + str( value) )[ value is not
        None ]</code>
  </format>
</parameter>

<parameter isout="1">
  <name>aligfile </name>
  <prompt>Alignment file </prompt>
  <type>
    <datatype>
      <class>Alignment</class>
    </datatype>
    <dataFormat>
      <ref param="outputformat"/>
    </dataFormat>
  </type>
  <precond>
    <code proglang="perl">$outputformat =~ /^(NEXUS|GCG|PHYLIP|FASTA)$/</code>
    <code proglang="python">outputformat in [ "FASTA", "NEXUS", "GCG",
        "PHYLIP"]</code>
  </precond>
  <filenames>
    <code proglang="perl">(defined $outfile)? ( $outputformat eq 'GCG' )? (
        $outputformat eq 'PHYLIP' )?"$outfile":"*.msf" : "*.phy" :
        "*.nxs"</code>
    <code proglang="python">{ "FASTA":"*.fasta", "NEXUS": "*.nxs", "PHYLIP":
        "*.phy" , 'GCG': '*.msf' }[ outputformat ]</code>
  </filenames>
</parameter>
```

The user specifies the format in which `clustalw` will produce the result with the first parameter “outputformat”. The value of this parameter also becomes the **dataFormat** of the second parameter “aligfile” which is the paramater to capture the alignment.

2.5.1 Mobyle DataTypes Tour

Simple datatypes Simple datatypes characterize the parameters which are used to configure the jobs, and are used to generate the command line (or the paramfiles). They do not represent *per se* scientifically meaningful data, and cannot be bookmarked and reused. Parameters which have a “simple“ datatype are displayed as “simple“ form controls (text input, select box, radio button, etc...).

Boolean Represents boolean values. None value special case: for boolean, a None value means False whereas for all other types it means undefined.

Integer Represents integer values.

Float Represents float values.

String Represents string values. Since these strings will be on the command line, for security reasons some values are not allowed. The acceptable characters are alphanumerical words, spaces, quotes, plus and minus, single dots. 2 or more dots in a row are forbidden.

This restriction could be problematic for some programs e.g: `fuzznuc`, `fuzzpro`, `fuzztran`, ... Indeed, these programs, from the EMBOSS suite, allow to search patterns in sequences using a grammar using characters such as `[`, `]` or `*` forbidden in Mobyle. If the parameter used to specify the pattern is typed as string, many patterns won't be accepted. One possibility, when available, is to specify this kind of value in a file. The special characters will not appear on the command line and are thus allowed in Text parameter.

Choice Appears as a select list in the interface. The list can be:

- a **vlist** containing predefined entries,
- a **flist** containing entries computed on the fly.

A **vlist** element contains 2 or more **velem** elements. Each of them allows to define a new entry consisting of:

1. a **value** that will be assigned to the variable `$value`, unless **undef**='1',
2. a **label** meaningful to the user.

The **undef** attribute is used to indicate that the **value** of the **velem** element must be considered as undefined. In the description of `clustalw-multialign` (see 2), the block

```
<velem undef="1">
  <value>null</value>
  <label>CLUSTAL</label>
</velem>
```

means that if "CLUSTAL" is chosen, the variable `$value` is not defined. In that case, given the **code** element:

```
<code proglang="python">( "" , " -output=" + str( value) )[ value is not None ]</code>
```

the `-output=` won't appear in the command line.

To force the user to consciously choose an entry:

1. set the **ismandatory** attribute to 1 for the **parameter**,
2. set the **vdef** to "null",
3. put as first **velem** of the **vlist** something like:

   ```
   <velem undef="1">
     <value>null</value>
     <label>Choose a value</label>
   </velem>
   ```

An **flist** element contains **felem** subelements instead of **velem** elements. It obeys the same mechanisms as the **vlist**, but instead of specifying the command line (or **paramfile**) code in a **format** element, the evaluated code is specified for each corresponding felem item in a **code** element. Its main purpose is to facilitate the construction of the command line, as the choice between the different codes requires nested *if* conditions if the more than two values are possible. Hence, the **flist** and **format** elements are mutually exclusive in a **parameter** element.

flist evaluates three different python expressions to generate the command line

```
<flist>
  <felem undef=”1”>
    <value>null</value>
    <label>no cutoffs; winner−takes−all (default)</label>
    <code proglang=”perl”>’’</code>
    <code proglang=”python”>’’</code>
  </felem>
  <felem>
    <value>cutoff_95</value>
    <label>specificity &gt;0.95</label>
    <code proglang=”perl”>( $type eq ’p’)? ” −p 0.73 −t 0.86 −s 0.43 −o 0.84 ” :
        ” −t 0.78 −s 0.00 −o 0.73 ”</code>
    <code proglang=”python”>( ” −t 0.78 −s 0.00 −o 0.73 ” , ” −p 0.73 −t 0.86 −s
        0.43 −o 0.84 ” )[ type == ’p’ ]</code>
  </felem>
  <felem>
    <value>cutoff_90</value>
    <label>specificity &gt;0.90</label>
    <code proglang=”perl”>( $type eq ’p’)? ” −p 0.62 −t 0.76 −s 0.00 −o 0.53 ” :
        ” −t 0.65 −s 0.00 −o 0.52 ”</code>
    <code proglang=”python”>( ” −t 0.65 −s 0.00 −o 0.52 ” , ” −p 0.62 −t 0.76 −s
        0.00 −o 0.53 ” )[ type == ’p’ ]</code>
  </felem>
</flist>
```

MultipleChoice Similar to the Choice datatype but represented as a select box where several values can be selected at once by holding the appropriate key. The selected values appear on the command line separated by the value of the **separator** element.

Filename Generally used to specify a result file name when the program offers the corresponding option. For security reasons, the user values `# " ' < > & * ; $ ‘ | ( ) [ ] { } ?` are not allowed.

File-based datatypes File-based datatypes characterize all the parameters and user data that are stored as files. Such data are very diverse, spanning from simple analysis reports to sequence data. They are displayed in the job submission forms using a specific component, the databox, which facilitates the load and reuse of data across analyses.

AbstractText Should not be used directly as a datatype class in Mobyle, it only represents any text file that should not be included in chaining options that are associated with .Created to avoid unwanted inheritances. Should only appear as the value of the **superclass** element of an actual type.

Text Handles “generic“ text files. Before writing the data, the content is cleaned up, *i.e.* the windows end-of-line \n\r are replaced by Unix \n. Moreover, some characters of the file name (`# " ' < > & * ; $ ‘ | ( ) [ ] { } ?`) are replaced by _ for security reasons. If an input parameter is typed as **Text** a lot of outputs will be potentially chained to this parameter. Please use a more specific type if possible.

Binary Handles ”binary” data files. A good example of the use of binary files as input data is in the **abiview** description: the input of EMBOSS abiview is an ABI trace. Of course the content is not “cleaned”.

Sequence Handles any text-based sequence format, such as FASTA, Uniprot, . . . A sequence converter based on `squizz` is already provided as a plug-in (but of course requires that you install squizz). Unlike in some previous Mobyle versions, the `readseq` support is no longer provided. In case you wish to use `readseq` or any other format conversion tool instead of `squizz`, you can write a python wrapper to handle it. For more information about format converters, please refer to the “Data converter management“ section of the Mobyle configuration guide. Usually, the parameter of “Sequence” datatype also defines several **dataFormat** corresponding to the formats handled natively by the program. If the format of the provided data is detected and does not match any of the formats handled by the program, Mobyle will try to convert the data into one of them, by sequentially testing the capabilities of the converters to convert the data.

Alignment Handles its data almost identically to the “Sequence” type, but represents an sequence alignment and has its own formats.

Tree Represents a phylogenetic tree. Does not implement any specific processing for now.

Report Designed to handle generic programs text results, if those are not handled by a more specific type.

2.5.2 XML DataTypes

The Mobyle DataTypes are based on python classes (in `Src/Mobyle/Classes)`. This system offers some powerful features such as inheritance, but has some limitations. Indeed, given that the chaining between two parameters is based on the type, a python class should be written for each type of data to avoid irrelevant chaining. In the bioinformatics field, the number of datatypes is too large to manage such a list. That’s why a mechanism to define a new datatype on the fly, called “XML typing”, is also available. When “XML typing”, a **class** element and a `superclass` element must be added in the program description. Beware:

- the **class** element is the new desired datatype,
- the **superclass** element refers to a Mobyle python DataType class

The desired datatype is thus considered as a subtype, for features such as programs chaining (appearing as a new datatype), but it is handled like its parent Mobyle python class for conversion and validation steps. For consistency reasons, when a same “XML data type” is defined in different parameters, it can’t refer to different Mobyle python classes. In other words, if the **class** element is identical among 2 parameters, the `superclass` element must be identical too.

2.5.3 Chaining

The Mobyle system provides a suggestion mechanism allowing users to use data in a defined set of programs:

- either by proposing in the form the data from the user’s workspace that are compatible with the parameter,
- or by letting the user interactively chains the result of a job to another program form.

In the following, “source” stands for a result or any bookmarked data. The selection of the programs and input parameters that can accept a source is based on type compatibility: the datatype of the input of the target parameter has to be the same as the one of the source or a superclass of it. Besides, if biotypes (see 2.5) have been defined in the output and input parameters, one of the source’s biotypes has to be included in the biotypes of the target parameter.

2.5.4 Extending Mobyle DataTypes

The Mobyle python DataType can be extended by coding new classes. Any new class must :

1. inherit from the DataType class or from another class which inherits from DataType,
2. implement at least 2 public methods, **“convert”** and **“validate”**, following the API defined in DataTypeTemplate (see `Core.py`).

To be able to use the new datatypes in program descriptions the same way as those provided in `Src/Mobyle/Classes`, don’t forget:

1. to put the new modules in `Local/CustomClasses` to prevent them from being erased during further updates,
2. to add the new classes in the `__init__.py`.

2.6 Output

Mobyle can handle results only if they are stored as files. Once a job is completed, the different output **parameter**s are evaluated by applying the **filename** masks (see 2.4.3) on the job directory to list the corresponding result files. Given that the output **parameter**s have a **datatype**, the mapping between output **parameter**s and result files is a way to type the results. As said in 2.5, typing is essential for chaining and data reusability.

2.7 Parameter display customization

The default display of a parameter is deducted from a number of characteristics: type, value range . . . This default display can be overridden to specify custom HTML code which will be used to layout **parameter**s, either inputs or outputs. This is done using the **interface** tag. Its use is rather tricky because the HTML content must conform to some rules that the Mobyle portal respects (CSS class names, javascript libraries that cannot be dynamically loaded). When using this mechanism to customize the layout of an output **parameter**, use the `$resultfile` string to tell the system where the name of the result file should be placed. The example below is taken from the `webmol` visualizer from the RPBS lab (`http://mobyle.rpbs.univ-paris-diderot.fr/programs/webmol_example.xml`). It specifies how to display a PDB-formatted result structure in a visualization applet and a text frame.

overridden the default display with interface element

```
<interface>
  <table width=”100%” XMLns=”http://www.w3.org/1999/xhtml”>
    <tr>
      <td width=”50%”>
        <applet code=”proteinViewer.class” codebase=”/applets/webmol/” width=”100%”
            height=”450px”>
          <param name=”PROTEIN” value=”$resultfile” />
          <param name=”PATH” value=”./” />
          <param name=”PDB_STRING” value=”” />
          <param name=”URL” value=”” />
          <param name=”EXT” value=”” />
        </applet>
      </td>
      <td width=”50%”>
       <object type=”text/plain” data=”$resultfile” height=”250px”>
       </object>
     </td>
   </tr>
  </table>
</interface>
```

Chapter 3

Writing a workflow description (*new in 1.0*)

A workflow is an ordered sequence of connected **task**s. Each **task** can be a program or a workflow. The root element of a workflow is the **workflow** tag. A workflow description is divided into 3 parts:

- **head** describes some generalities about the workflow.
- **flow** describes the different tasks and how to connect them.
- **parameters** describes the parameters of the workflow, that is to say the parameters that the user can/must specify as inputs of the workflow and the results of the workflow.

The following phylogeny workflow:

1. performs a multiple alignment,
2. used to compute a distance matrix,
3. to finally build a tree using the neighbor joining method.

It is used as an example to illustrate this section.

3.1 Head

Contains the same information as programs **head** (see 2.1), minus all the command-line execution-specific elements (**command** and **env**).

3.2 Parameters

This section describes the input parameters that the user must/can specify, and the results of a workflow. Of course each **task** remains accessible and all intermediate results are easily accessible. The "results" of the workflow indicates which outputs are presented to the user as final results. Workflow **parameter** elements contain the same information as program **parameter** elements (see 2.2). However, some elements are irrelevant in a workflow context, i.e, all the command-line execution-specific elements: **format**, **flist**, **argpos**, **paramfile**. Nevertheless, **datatype** and **dataFormat** specification remains very important to be able to reuse a result or visualize it with a viewer (see 4) in Mobyle. For instance, if the `archeopteryx` viewer has been deployed, it can be used to visualize the result of the phylogeny workflow defined above since the **parameter** 2 is typed as "Tree".

parameters part of a phylogeny workflow

```
<parameters>
  <parameter id="1">
    <name>sequences</name>
    <prompt>Input sequences</prompt>
    <type>
      <datatype>
```

```
        <class>Sequence</class>
      </datatype>
      <biotype>Protein</biotype>
    </type>
  </parameter>
  <parameter id="2" isout="1">
    <name>tree</name>
    <prompt>Phylogenetic Tree</prompt>
    <type>
      <datatype>
        <class>Tree</class>
      </datatype>
      <biotype>Protein</biotype>
    </type>
  </parameter>
</parameters>
```

The parameter with **id**="1" is the input of the workflow, whereas the parameter with **id**="2" with its attribute **isout**="1" represents the result of the workflow.

3.3 Flow

Is divided in two parts:

- **task**,
- **link**.

3.3.1 Task

A **task** element has the following attributes:

- **id** (*required*): a unique label allowing to identify each **task** in the workflow
- **service** (*required*): the name of a program or workflow
- **suspend**: an attribut with 2 allowed values "true" and "false". If **suspend** is true, the workflow will be suspended at the end of the task and will wait for an action from the user to resume. This allows the user to check intermediate results and decide if he wants to resume or cancel the workflow.
- **description**: a one-sentence-description of the task.
- **inputValue**: allows to specify some values for some **parameter**s of the **task** (program or workflow). This tag is used to specify a value:
  - because the **parameter**'s attribute **ismandatory**="1"
  - to replace the default value **vdef** of the **parameter**.

  The **name** attribute is used to refer to the **parameter**. Thus use a new **inputValue** element for each **parameter** to set.

  tasks section of phylogeny workflow

  ```
  <task id="1" service="clustalw-multialign" suspend="false">
    <description>Align the sequences using Clustal-W</description>
    <inputValue name="outputformat">PHYLIP</inputValue>
  </task>
  <task id="2" service="protdist" suspend="true">
    <description>Compute my distance matrix</description>
  </task>
  <task id="3" service="neighbor" suspend="false">
    <description>Perform neighbor-joining</description>
  </task>
  ```

  The **task** with **id** "1" refers to the `clustalw-multialign` program which **parameter** "outputformat" has "PHYLIP" as value. Of course, all files produced by the `clustalw-multialign`, `protdist` and `neighbor-joining` **task**s remain accessible.

3.3.2 Link

This is an important part of the workflow design. It describes how to connect the different **task**s. All connections must be explicit. Each **link** is directional and joins two points, a source and a destination.

The source must be:

case 1 an input **parameter** of the workflow (see 3.2)

case 2 a **parameter** of a **task**.

The source is unambiguously pointed out:

in case 1 by a workflow's input **parameter**'s **id** assigned to the **fromParameter** attribute,

in case 2 by the combination of:

1. a **parameter**'s **name** assigned to the **fromParameter** attribute,
2. a **task**'s **id** assigned to the **fromTask** attribute.

The destination must be:

case 1 an output **parameter** of the workflow (see 3.2)

case 2 a parameter of a **task**.

The destination is unambiguously pointed out:

in case 1 by a workflow's output **parameter**'s **id** assigned to the **toParameter** attribute,

in case 2 by the combination of:

1. a **parameter**'s **name** assigned to the **toParameter** attribute
2. a **task**'s **id** assigned to the **toTask** attribute.

Example of workflow connections

```
<link toTask="1" fromParameter="1" toParameter="infile"/>
<link fromTask="1" toTask="2" fromParameter="aligfile" toParameter="infile"/>
<link fromTask="2" toTask="3" fromParameter="outfile" toParameter="infile"/>
<link fromTask="3" fromParameter="treefile" toParameter="2"/>
```

- The first **link** element indicates that the **parameter** with **id**=“1” from the workflow will be sent into the **parameter** named “infile” from the **task** with **id** =“1”.
- The second **link** element indicates that the **parameter** named “aligfile” from the **task** with **id**=“1” will be sent into the **parameter** named “infile” of the **task** with **id**=“2”.
- The last **link** element indicates that the **parameter** named “treefile” of the **task** with **id**=“3” will be sent into the **parameter** with **id**=“2” of the workflow. This means that it's a result of the workflow.

Chapter 4

Writing a viewer description (*new in 1.0*)

Viewers are a way to embed type-dependent visualization components for the data displayed in the Mobyle portal. Viewers easily overcome the limitations of the basic text-based data previews offered by Mobyle. These XML files provide a way to incorporate custom interface code to display data of a given type in the browser. Such custom interfaces can incorporate HTML-embeddable components such as Java applets, Flash applets or Javascript code. For instance, using viewers, the Jalview component can automatically be included wherever it is relevant, so that the user can immediately visualize the results of multiple alignment programs such as ClustalW or Muscle in the portal.

A viewer is composed of two elements:

1. an XML description in many points similar to program descriptions 2
2. a set of dependencies, which are client-required files stored in a directory (e.g., jar files for a Java applet).

The root element of a viewer description is the **viewer** tag (instead of the previously-cited **program** or **workflow**). It is divided in two parts:

- the **head** (*required*) which describes some generalities about the viewer and how to display it.
- the **parameters** (*required*) which describes the kind of data Mobyle can apply the viewer to.

4.1 Head

Contains the same information as a program (see 2.1), minus invocation-specific elements such as **command**, **env**, etc...In addition to this, the author has to fill the **viewer** element with a template of the XHTML code that displays the data when these data have a "viewer" type. This template is used by Mobyle to generate on the fly the client content required to visualize the data. This template is described in 4.3.

4.2 Parameters

The **parameters** section contains all the input parameters that will be used by the component to, generate the visualization. It's very similar to the **program parameters** section 2.2, minus the server-side invocation details.

4.3 Example: Jalview

Jalview is a component allowing to graphically display, explore and analyze multiple sequence alignments. The aim of this viewer is to enable users to visualize any compatible multiple alignment data. "Compatible" means that the file format of the output alignment is compatible with those accepted by the `Jalview` applet. The `jalview.xml` file is provided in the `pasteur-viewers` descriptions package.

`jalview.xml` file

```
[...]
<viewer>
  <head>
    <name>jalview</name>
    <version>2.0.11</version>
[...]
    <interface type=”viewer”>
      <center xmlns=”http://www.w3.org/1999/xhtml” style=”height: 100%”>
        <applet codebase=”/data/viewers/jalview” code=”jalview.bin.JalviewLite”
            archive=”jalviewApplet.jar” width=”100%” height=”100%”>
          <param data-parametername=”aligfile” name=”file” value=”data-url”/>
          <param name=”embedded” value=”true”/>
          <param name=”linkUrl_1” value=””/>
          <param name=”srsServer” value=””/>
        </applet>
      </center>
    </interface>
  </head>

  <parameters>
     <parameter>
       <name>aligfile</name>
       <prompt>Alignment file</prompt>
       <type>
         <datatype>
           <class>Alignment</class>
         </datatype>
         <dataFormat>FASTA</dataFormat>
       </type>
     </parameter>
  </parameters>
</viewer>
```

4.3.1 Chaining

Given that in this example:

- the **datatype** of the input **parameter** “aligfile” is “Alignment”
- the **dataFormat** of the input **parameter** “aligfile” is “FASTA”

the portal will offer to visualize any FASTA alignment file with Jalview by providing an additional “Jalview” button in the web interface. Clicking on that button will trigger the display of the Jalview alignment in a modal dialog box.

4.3.2 Layout

The **applet** and **param** elements are the same as any regular applet embedding. But it's necessary to provide the applet with the url of the alignment file. Thanks to the **data-parametername** attribut telling that the mapping must be done with the “aligfile” **parameter**, Mobyle replace“data-url” with the real url of the input file corresponding to the “aligfile” **parameter** to generate the final HTML contents (see below). This will enable the Jalview applet to load this file.

Example: jalview generated HTML code

```
[...]
  <applet width=”100%” height=”100%” archive=”jalviewApplet.jar”
      code=”jalview.bin.JalviewLite” codebase=”/data/viewers/jalview”>
    <param value=”http://mobyle.example.org/data/jobs/muscle/M30387337085962/muscle.out” name=”file”
        data-parametername=”aligfile”>
    <param value=”true” name=”embedded”>
    <param value=”” name=”linkUrl_1”>
    <param value=”” name=”srsServer”>
  </applet>
[...]
```

Chapter 5

Validation

When creating or modifying a service description, validating the XML code is highly recommended to detect mistakes which can cause problems, in particular at display or job execution time. To validate a service description, use the mobvalid script located in the $MOBYLEHOME/Tools directory[1].

As the typing system is central to the chaining process, the consistency between the types is crucial. A same "XML data type" appearing in different service descriptions must always refer to the same python superclass. Some typographical mistakes in the XML class or in a **biotype** can prevent the desired chaining or lead to an unexpected chaining. The mobtypes script located in $MOBYLEHOME/Tools scans service descriptions and python classes to create a repository of all types used in your portal. This tool helps the Mobyle administrator to maintain the consistency of his portal or the service description author to choose the right types. The usage of mobtypes script is explained in the associated README file.

When the Tools/mobtypes script analyzes a service description, it generates a report with 3 sections:

- the Mobyle **datatype** based on python class,
- the DataTypes defined by "XML typing",
- the **biotypes** (see 2.5) used.

By default this script analyzes all installed service descriptions. It helps the Mobyle administrator to keep a consistent set of types for the portal.

To be used in the portal, the services must be deployed and debugged if necessary. Those operations are described in how_to_configure_mobyle.pdf.

[1] This tool will simply automate the validation of the XML file, according to its grammar based in the *.rnc/rng files and a set of additional schematron rules. However, it should not be necessary to be familiar with these files to be able to write a service description.

Chapter 6

Upgrade from version 0.97

Upgrading service definitions from previous versions to v1.0 should be reasonably straightforward. The only two incompatibilities are:

- in the **dataFormat** element: whereas in versions 0.9x **dataFormat** elements are sometimes placed in a container **acceptedDataFormats** element, this container has now been removed and dataFormat elements should be direct children of the **type** element.
- the **interface** element has to have a **type** attribute: the details of this mechanism which extends the customization possibilities are described in the detailed documentation, but legacy 0.97 XML **interface** elements should be updated with a **type="form"** attribute for inputs and a **type="job_output"** attribute for results.

The other evolutions of the grammar are additions are annotated with the "*new in 1.0*" label.