How to write a new Mobyle service definition

Mobyle 1.0

Contents

Abstract

We use XML in different parts of Mobyle: to describe services (programs, workflows, viewers) to save informations about user workspaces - jobs and data (what we call session), and to store jobs execution informations. The grammar of these XML files is described in relax-ng format. The schema is stored in the *.rnc and *.rng files, located in the "Schema" folder of the project. The session and jobs XML files are generated automatically by Mobyle thus you won't have to modify them. On the other hand, the XML files describing the services are human made and are a central piece to integrate new services in Mobyle. In this document I will give some keys to build the XML service descriptions and understanding the "Mobyle philosophy". Even if you do not intend to use the XML distributed by Institut Pasteur, I recommend you download them: I will illustrate some tricky points using them.

The releases of services definitions are available on our public ftp server(ftp://ftp.pasteur.fr/pub/gensoft/projects/

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

There are 3 subprojects of Mobyle maintaining the services definitions:

- 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 definitions 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 1

Writing a Service defintion

There are three kinds of services in Mobyle: programs, workflows and viewers.

1.1 Programs

A program is a software which runs on the server side, is callable in a shell command line and returns some results. We used XML to describe these softwares and to be usable by Mobyle. Hereinafter, I will call program definition the XML definition of a such software.

A program definition aims to capture different kinds of information:

- how to invoke the program (a program wrapper),
- how to display options in submission forms or job results (a UI definition),
- some controls on value or parameters compatibilities (a "semantic" description).

The root element of a program definition is the *program* tag. A program definition is divided in two parts: the *head* which contains general information about the program, and the *parameters* which describes the different input data, options and results of the program, how to build the command line and so on.
In the rest of this document, I will not describe each schema element, but I will focus on the most important ones.

1.1.1 head

- **name** (*required*) is the name of the interface you describe. It does not have to be identical to the name of the actual program you call, but if you can we recommend it. However, it is mandatory that the value of this field is identical to the name of the XML file (minus its extension). e.g.: for the file `blast2.xml`, the name tag is the following: `<name>blast2</name>`.
- **version** is the version of the program you interface, not the version of the XML description itself.
- **package** (*new in 1.0*) describes the package the program belongs to. It can contain all of the elements defined in head (`name`, `version`, `doc` , ...).

how to include package information (extract of dnadist XML 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>
```

- **doc**
  - **description** a brief description of the software, in text or XHTML. This description will appear in the top of the submission form.
  - **reference** the scientific reference that has to be used to cite that program, in text or XHTML. You can also provide a DOI or a direct URL.
  - **authors** the authors of the program that have to be cited.
  - **doclink** any URL pointing to documentation useful to this program. If a doclink is specified, a button "Help Pages" will appear on top of the web form. If there is one doclink element the documentation will be available either directly by clicking on "Help Pages" button, or at the bottom of the web form as link. But if there are several doclink elements, the button sends the user to the bottom of the interface where he can choose between them.
  - **sourcelink** (*new in 1.0*) any URL where the software corresponding to these definition can be found. It is not used so far in the Mobyle Portal, and is used rather as documentation by server administrators.
  - **homepagelink** (*new in 1.0*) the homepage URL of the software project.
  - **comment** in the header, it is a text describing with more details the program. This comment is shown by clicking on the red question mark beside the title.
- The **category** element must be chosen carefully, because the program tree in the left panel is based on it. You can specify multiple "category" elements, each one specifying a path within the classification, composed of nested nodes separated by colons. If several categories are specified, the program appears in each branch of the tree. If no category is specified, the programs appears in the left panel at the top level.
- The element **env** allows you to specify an environment variable that will be defined prior to launching any job for this tool. Because this part is highly site-dependent, it is usually implemented with entities or XInclude. e.g:
  you can specify the location of your blast databases with this mechanism.
  `<env name=''BLASTDB''>/usr/local/ncbi/db</env>`
  or the path of the binary if it is not in the usual path
  `<env name=''PATH''>/path/to/my/binary</env>`

Entities and XInclude

It is possible to include site-dependent information (such as the above-mentioned env tag) by including external pieces of XML code. The preferred mechanism to do this is to use XInclude, as XInclude provides error-handling mechanisms which are not available with XML entities. They can be used to refactor some parts of a set of descriptions, or specify site-dependent variables. If you have for instance a set of interfaces which wraps different programs from the same package, you can include common package information in an entity that you include in the different XML files. All you need to do is to make sure the XInclude elements are correctly formatted and the XInclude namespace is declared somewhere.

Example: how to include databank (extract of golden XML 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 used also to share package information in the program definitions belonging to the same package.

Example: how to share package informations (extract of dnadist XML description)

```
<program>
  <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>
```

and in Entities folder (at the same level of Programs) we have the phylip_package.XML which is shared by all phylip programs (dnadist, dnapars, protdist, protpars ...)

Example: how to share package informations (extract of 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>
```

1.1.2 Parameters

The `parameters` element is a direct child of either the `program` or the `paragraph` element. In the first case it includes all of the defined parameters for the program, whereas in the latter it defines the parameters of a given paragraph. It can include parameters or nested paragraph elements.

1.1.3 Paragraph

The paragraph has a double meaning: it can group the evaluation of different parameters, with respect to a set of preconditions for instance, and it also groups visually these parameters in the interface. Thus, a paragraph is both a set of parameter visually in the submission form and mean to share some properties like:

- **argpos** that specifies the position of the argument on the command line (by convention the command argpos is 0). If argpos is not specified at the parameter level, it takes the argpos of the immediately upper level (paragraph), and so on (if the parameter order doesn't matter, it's not mandatory to specify one). This mechanism allows to set easily the relative position in the command line for a set of parameter.
- the **precond** element allows to specify in which condition the parameter is evaluated. All preconditions are evaluated beginning by the condition nearest the root to the condition of the parameter. This mechanism allows to refactor precondition code.
- **layout** is a structure that allows to override the default display of the parameters belonging to a paragraph. The default disposition of the parameters is a vertical succession, where parameters in the form and job results in the job are layed out in the same order as in the program description. This can be overriden using the layout tag, which allows to define horizontal groups (with the hbox tag) and vertical groups (vbox tag) in a recursive structure, which links to each parameter using its name.

extract of 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 was 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>
```

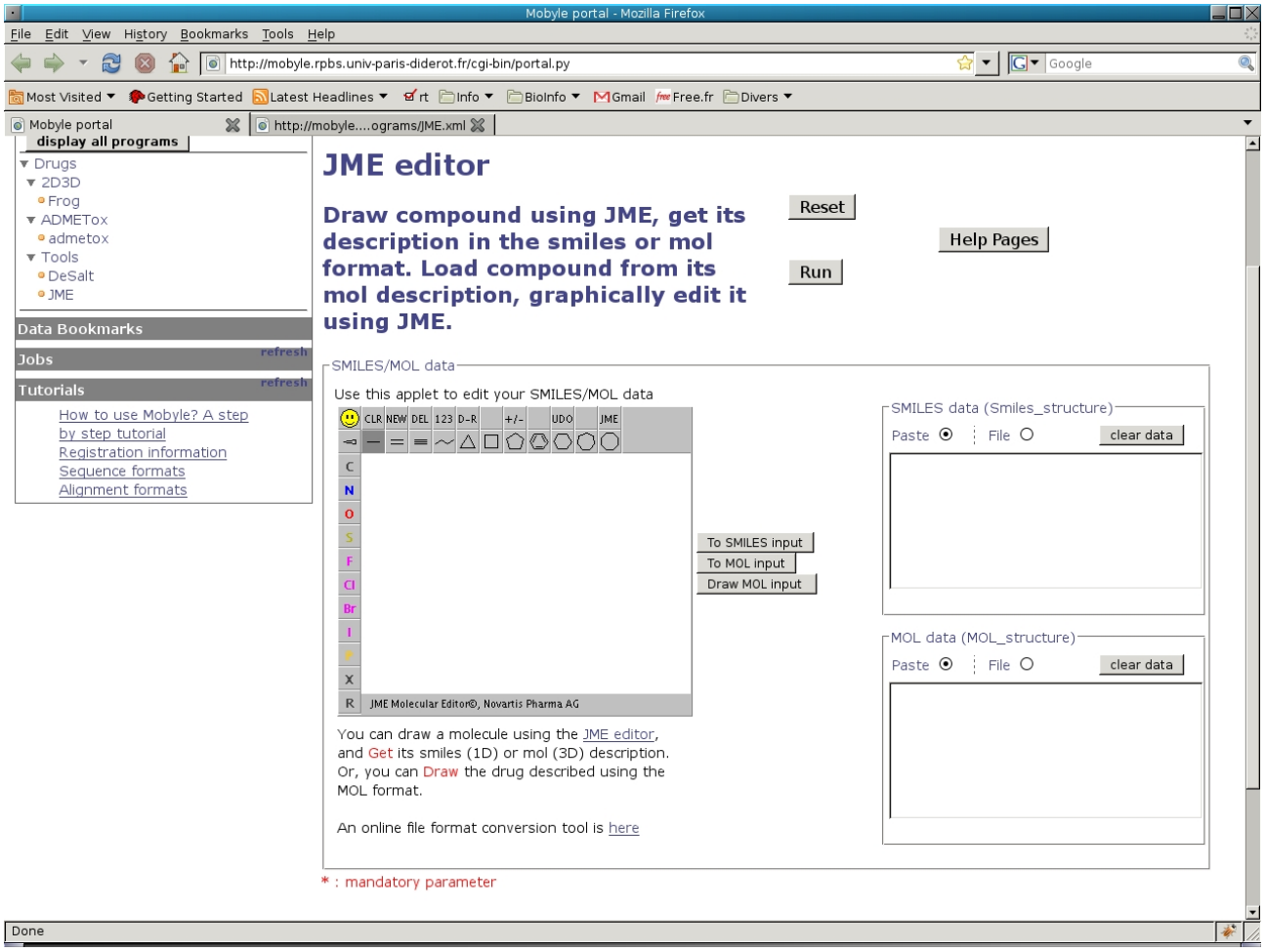

Figure 1.1: JME interface

1.1.4 Parameter

The `parameter` element describes the options, the inputs and outputs of a program. The information to build the command line is stored in it. It's an essential piece of the program description.
Its attributes are:

- **ismandatory** means that the parameter must be specified by the user. This kind of parameter is represented with a red star next to their prompt in the interface. There is a special case, when the parameter is mandatory but a precond is also defined (see below). In this case the parameter becomes mandatory only if the result of the precondition is true (these parameters are not indicated by a red star).

- **isout** means the parameter is produced by the program. It is used to retrieve the results. Mobyle always generates 2 files (`[program name].out` and `[program name].err`), corresponding to the standard output and error streams. These two implicit results are automatically shown to the user in the results page (if they are not empty) and are typed as Text. If you want to provide a more explicit or detailed description of the standard output, you can define a `parameter` element with the desired type and we declare that this parameter corresponds to the standard output with the attribute **isstdout**. For an example see parameter golden_out in golden.XML.

standart outpout 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>
```

- The input parameters are shown to the user in the submission form, the output parameters in the results form (if a result corresponding to this parameter is really produced). Sometimes it's necessary to add a parameter to control others but we don't want it appear on the interface. To do that we set the attribute **ishidden** to true (the value of hidden parameter can't be set by the user). For example see parameter rateAll in seqgen.XML.

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>
```

Each parameter must have a **name**. The name must be unique among all parameters and paragraphs. This name must be a valid python variable name (it cannot begin by a number ...). The name is used

mainly inside Mobyle to refer to this parameter but the user does not see it on the interface. He sees the **prompt** which is a human-readable label for the parameter. Some parameters depend on other parameters. For example the parameter "A" has meaning only if parameter B is set. To specify this kind of constraints we can use the **precond** element. The parameter which has a precond will be evaluated only if this last is true.
The code contained in the **format** element 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 printed on the command line. But sometimes we need to write it in a file. We do that with the element **paramfile**. We used this mechanism to simulate the interactive behavior of some programs such as those of the Phylip suite (see pars.XML, protdist.XML, fitch.XML, ...).

We may need to control finely the values provided by the users. The ctrl element allows to specify additional constraints, declaring that a given parameter's value has to be less than a maximum value, be odd .... 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).

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)</
  <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]</cod
  </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...<
  </comment>
</parameter>
```

We discussed mainly about the input parameters but an important step is to define how to retrieve the results produced by a programs. Such results are described using parameter elements that specify the isout/isstdout attribute, and the element **filenames**. In this element we define unix masks which are used to map filenames with this parameter. Thus one parameter can map easily 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 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" (eg. parameter hydrophobicity_files in toppred.XML). The unix mask is defined in a code element thus the code is evaluated before to use as unix mask. We use it to generate a mask which cannot be known statically (e.g. parameter .treefile in bionj.XML). Only 1 mask can be defined per code element. But the element "filenames" may have several children elements "code" (parameter in .XML).

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>
```

To complete this outline, you may find the following 2 elements useful to provide help and documentation to the users. The **comment** and **example** allow to generate in line help and example data for the parameter or paragraph. When help is available, a clickable red question mark appears beside the prompt. When an "example" is available, a "use example data" link appears beside the parameter. By clicking on it, the user fill automatically the corresponding databox with this value.

1.1.5 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 (for the input parameters), the chaining possibilities between programs and data reusability. In Mobyle, typing is "multidimentional", meaning it is based on several fields:

- the **biotype** describes the biological object (Nucleic , Protein , Drug ,...). It is not mandatory as some parameters do not represent biological objects. The biotype values are free text labels, but to chain the data appropriately, we must agree on the used labels (take special care on the spelling and case).
- **type** describes the computing object.
- **DataFormat** (*new in 1.0*) specifies the data format 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 in output (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 dataformat of a result cannot be predetermined, but can be computed it at runtime based on the values of other inputs by the user. There are two ways to express this: (1) to reference directly a parameter in DataFormat or (2) to test the value of another parameter. The two following examples illustrate these possibilities:

dataFormat by referencing an other parameter (extract of clustalw-multialign.XML)

```
1.  <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 ]
    </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"]</c
    </precond>
    <filenames>
      <code proglang="perl">(defined $outfile)? ( $outputformat eq 'GCG' )? ( $output
      <code proglang="python">{ "FASTA":"*.fasta", "NEXUS": "*.nxs", "PHYLIP": "*.phy
    </filenames>
  </parameter>
```

The first parameter "outputformat" allow the user to specify in which format clustalw will produce the result. The second parameter aligfile which is the parameter to capture the alignment, have for dataFormat the value of parameter "outputformat".

dataFormat by a value (extract of muscle.XML)

2.
```
  <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 lang="en">clustalw : Write output in CLUSTALW format with CLUSTAL W (1.81
    </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 first parameter "outformat" allows the user to specify in which format muscle will produce the result. The second parameter "alignmentout" allows to capture the alignment result. We define the format 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.

- **card** represents the cardinality, i.e. the number of distinct values that can be set for the parameter.

The Mobyle data types are based on python classes (in Src/Mobyle/Classes). This system offers some powerful features as inheritance ..., but has some limitations. Indeed, we have based the chaining between two parameters on the type, we should write a python class 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. This is the reason why we provide a mechanism to define a new datatype on the fly, called "XML typing". The element `class` does not refer to a mobyle python class but the new datatype we need, and we add an element `superclass` which refers to a Mobyle python datatype class. This mean that it is 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, same "XML data type" in different parameters cannot be defined as refering to different Mobyle python classes.

Mobyle DataTypes Tour.

Boolean Represents boolean values. Special case of None value: for all other types a None value mean: undefined. But for boolean it means False.

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 unavailable. The value must be composed of words , spaces, quotes , minus , plus, dots (if not followed by another dot) and eventually surrounded by commas.

This restriction could be problematic for some programs eg: fuzznuc, fuzzpro, fuzztran, ...: these softwares, from the EMBOSS suite, allow to search patterns in sequences using a grammar with following forbidden characters: [ ] * ,.... If the parameter to specify the pattern is typed as string lot of patterns could not be used. One possibility, when available, is to specify this kind of value in a file. The specials characters will not appear on the command line and are thus allowed in Text parameter.

Choice It appears (by default) in the interface as radio button if there is less than 3 choices and as a select box if there is more than 3 choices. It allows the user to choose a value among a range of predefined values. The value is treated as a literal and of course to validate the user value must be in the range of predefined values.

MultipleChoice It is similar to the Choice DataType, but always represented as a select box where several values can be selected at once. The selected values appear on the command line separated by the value of element **separator**.

AbstractText This type has been created to avoid unwanted inheritances. It should only appear as the superclass of an actual type.

Text This type is displayed as a "data box" in the job submission form. It handles text files. The data provided by the user will write on disk (in the session space). Before to write the data, its contents is cleaned up (the windows end of line $\backslash n \backslash r$ are replaced by unix $\backslash n$), we rename the file, and some characters (# " ' < > & * ; $ ` | ( ) [ ] { } ?) are substituted for security reasons. It is very important to realize that Text is not a very expressive type, and if an input parameter is typed as Text a lot of outputs will be potentially chained to this parameter. If possible, please use another more specific type.

Binary This type is identical to the "Text" type, but handles binary data. Of course the data contents is not "cleaned".

Filename This one is used to specify a file name. Some programs offer the possibility to specify the name of the results file with an option. For security reasons, the user values # " ' < > & * ; $ ` | ( ) [ ] { } ? are not allowed.

Sequence The Sequence input parameters appear as "data boxes". We provide with Mobyle a sequence converter based on squizz as a plugin. Unlike the previous Mobyle versions, we does not provide the readseq support anymore. If you prefer to use readseq instead of squizz, it's allways possible but you must write the converter python wrapper needed by Mobyle to plug readseq in Mobyle system (see TODO). To use squizz as Sequence converter you must specify that in your configuration file. Like this:

```
DATA_CONVERTER={
      'Sequence': [ squizz_sequence('/local/gensoft/bin/squizz') ] ,
      'Alignment': [ squizz_alignment('/local/gensoft/bin/squizz')]
                }
```

Usually the parameter of Sequence datatype defined also several elements dataFormat which are the format understood by the program. Mobyle will try to convert the sequence in the first dataformat found. If it's not possible try the second and so on.

Alignment Handled identically to the Sequence type; but represents an Alignment, and has its own formats.

Tree This class represents a phylogenetic tree. It does not implement any specific processing for now.

Report This class is designed to handle text programs results which are not handle by a more specific Type.

XML data types

As we explained before, this mechanism allow to defined easily a new data type in Mobyle. But the new data type must be chosen carefully because the chaining is based on it. We provide a script (Tools/mobtypes) which analyses XML and generates a report, in 3 sections:

- the mobyle DataType based on python class.
- the data types defined by XML.
- the biotypes used.

By default this script analyse all installed XML programs definitions. It's help the Mobyle administrator to keep coherent a set of types for the portal. This kind of repository is programs specific, so we distribute it with the XML programs (ftp://ftp.pasteur.fr/pub/gensoft/projects/mobyle/).

chaining

The mobyle system provides a suggestion mechanisms that allows users to use data in a defined set of programs, wether by proposing in the program form user workspace data that are compatible with the parameter, or by letting users interactively chain the result of a job to another program form. The selection of the programs and input parameters which can accept a result (or any bookmarked data) is based on type compatibility: the datatype of the target input has to have the same datatype as the source or a superclass of it. Besides, if biotypes have been defined in the output and input, one of the source biotypes has to be included in the target biotypes.

extending mobyle types

You can extend the Mobyle python data type by coding new classes. Your new classes must inherit from DataType or another class which inherits from DataType and must implement at least 2 public methods "convert" and "validate" following the same api defined in DataTypeTemplate (in Core.py). To avoid to be erased during a further update, your new modules must be located in Local/CustomClasses, and your new classes must be added in the __init__.py (see PhylipTree example in Example/Local/CustomClasses). These new datatypes can then be used as those in Src/Mobyle/Classes in your XML.

1.1.6 the Output

Mobyle can handle results only if they are stored as files. Once a job is finished, the different output parameters are evaluated using the **filename** masks on the job directory to store the corresponding filenames. The mapping between the output parameters (with attribute isout/isstdout="1") allows to organize the results for the user and type them, which is essential for the chaining and the data reusability.

1.1.7 Parameter display customization

The display of a parameter is by default deducted from a number of characteristics, as we stated before: type, value range, etc. This default display can be overriden to specify custom HTML code which will be used to layout parameters, either inputs or outputs. This is done using the interface tag. It contains HTML code, but it's use is rather tricky because it should conform some rules that the Mobyle portal respects (CSS class names, javascript libraries which cannot be dynamically loaded. When using this mechanism to customize the layout of an output parameter, you will 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.

overriden 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>
```

1.2 Workflows (*new in 1.0*)

A workflow is na ordered sequence of connected task. Each step can be a program or a workflow. The root element of a workflow is the "workflow" tag. The workflows definitions are divided in three parts:

- **head** which describes some generalities about this workflow.
- **flow** which describe the different tasks and how to connect them.
- **parameters** which desribed the parameters of this workflow. This is the parameters that the user can/must specify in input of this workflow and the results of this workflow.

1.2.1 head

It's contain the same informations as programs head (see 1.1.1).

1.2.2 flow

Is divided in two parts:

task

A task element have the following attributes:

- **id** (*required*) It's an uniq label which permit to identify this task in the workflow.
- **service**(*required*) point out the name of a program or workflow. **suspend** Two values "true" or "false" are allowed for this attribute. If suspend is true that mean the workflow will be suspend at the end of this task and need an action by the user to resume it. It allow the user to check intermediate results and decided if he want to continue or cancel the workflow.
- **description** It's short description (one sentence describing this task).
- **inputValue** Allow to specify some values for some parameters of this task (program or workflow). This mechanism is used to specify a value which is mandatory for this program or to specify a value different from the programs's parameter default value (vdef). the name of the parameter is specify whith the attribure "name". to illustrate this point I willl take a workflow of phylogeny with 3 steps:
  - a mutiple alignment.
  - a ditance matrix computing.
  - a contruction tree based on neighbor joining ethod.

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>
```

This task with id "1" refer to the "clustalw-multialign" program and we specify that the parameter "outputformat" have "PHYLIP" as value. You must have an "inputvalue" element by parameter to specify.

link

This is an important part of the workflow design. It describe how to connect the different tasks. All connections must be explicit. Each link is directional and join two points, a source and a destination. The source must be either an input parameter of the workflow (see 1.2.3) or a parameter of a task. The destination must be either an output parameter (see 1.2.3) of the workflow or a parameter of a task.
The source is unambigously point out by **fromParameter** attribute with the id of this workflow parameter (if it's an input of this workflow), or by the combination of the **fromParameter** attribute with the name of a parameter and the **fromTask** attribute with the id of this task.
The destination unambigously point out by **toParameter** attribute with the id of this workflow parameter (if it's a result of this workflow), or by the combination of the **toParameter** attribute with the name of a parameter and the **toTask** attribute with the id of this task.

example of workflow connection

```
<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 indicate that the parameter with id="1" from the workflow will be send into the parameter parameter names "infile" from the task with id ="1".
- The second link element indicate that the parameter named "aligfile from the task with id="1" will be send into the parameter named "infile" of the task with id="2".
- The last link element inidicate that the parameter named "treefile" of the task with id="3" will be send into the parameter with id="2" of this workflow. this means that it's a result of this workflow.

1.2.3 parameters

This section describe the input parameters that the user must/can specify, and the results of this workflow. Of course each steps remains accessible and all intermediates results are easily accessible. By results of the workflow, we mean the results which will be presented to the user as final results. The parameters element contain the same informations as describe in programs section (see 1.1.2), however some elements are irrelevant in workflow parameter context: format, flfist, argpos, paramfile. The data typing and dataformat information are very important to be suitable reused or visualized with a viewer (for more expalnation see 1.3) in Mobyle. For instance if the "archeopteryx" viewer has been deployed, the result of this workflow could be visualized with this applet 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="true">
    <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" represents the result of this workflow which will show to the user. Of couse, all files produced by the clustalw-multialign, protdist and neighbor-joining steps are accessible.

1.3 Viewers (*new in 1.0*)

Viewers are a way to embed type-dependant visualization components for the data displayed in the Mobyle Portal. The limitations of the basic text-based data previews offered by Mobyle can easily be overcomed by defining Viewers. These XML files provide a way to incorporate custom interface code that will display data of a given type in the browser. Such custom interfaces can incorporate HTML-embeddable components such as Java or Flash applets of Javascript code. For instance, using viewers, we can automate the inclusion of the Jalview component to visualize multiple alignments wherever it is relevant in the portal, such as the results of multiple alignment programs like ClustalW or Muscle.

A viewer is composed of two elements:

- a definition, an XML file which is in many points similar to program definitions described above,
- and a set of dependencies, which are client-required files stored in a folder (e.g., jarfiles for a Java applet).

The root element of a viewer definition 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 describe on what kind of data Mobyle can apply the viewer.

1.3.1 head

It contains the same information as a program (see 1.1.1) or workflow head, minus invocation-specific elements such as command, env, etc... In addition to this, the author has to provide a template of the XHTML code that displays the data in a viewer element with 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 further in the viewer example we provide later.

1.3.2 parameters

The parameters section contains all the input parameters that will be used by the component to, generate the visualization, once again similar to the `program` parameters section, minus the server-side invocation details.

1.3.3 Example: Jalview

Jalview is a component that allows to display, explore, and analyse graphically sequence multiple alignments. The aim of this viewer example (the jalview.xml file of the pasteur-viewers definitions package) is to enable users to visualize any compatible multiple alignment data (by compatible, we mean that the data format is compatible with those accepted by Jalview).

jalview.xml file

```
[...]
<viewer>
  <head>
    <name>jalview</name>
    <version>2.0.11</version>
    <doc>
      <title>Jalview</title>
[...]
    <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>
```

Chaining Based on the type of the Alignment input file, "aligfile", the portal will offer to visualize any FASTA alignment file with Jalview by providing an additional "Jalview" button in the web interface. Clicking on it will trigger the display of the Jalview alignment in a modal dialog.

Layout In this example, we define an aligfile input parameter. This input parameter will be used to generate the final HTML contents by substituting the value attribute's value with the url of the input file corresponding to the aligfile parameter (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 2

Deploying and Debuging

2.1 Validation

When creating or modifying a program description, validating the XML code is highly recommended as it can detect problems which can cause sometimes obscure problem during the use of the description, at display or job execution time for instance. To do so, use the mobvalid script which is located in the Tools subfolder of the Mobyle directory[1]. As the typing system is central to the process chaining, the consistency between the types is crucial. All "XML data type" types must refer to the same python class, some typographical mistakes in this XML class or biotypes could prevent or lead to unexpected chaining. So we provide a tool: "mobtypes" which scan all types and make a kind of repository of all types used in your portal. This tool helps the Mobyle administrators to maintains the consistency of his portal or the XML writer to choose the right types. mobtypes is located in the Tools subfolder and the usage is explain in the associated README file.

2.2 Deploying Services

2.2.1 Overview

Before deployment the services definitions are located in 2 parallele directories structures. The first one $MOBYLEHOME/Services/ is intend to sore services definitions provide by us or ??? . The second one $MOBYLEHOME/Local/Services/ is intend to store the services definition you write.

in $MOBYLEHOME/Services/ you have following subfolder:

- **Programs** to store programs definitions.
- **Workflows** to store workflows definiotions.
- **Viewers** to store viewers definitions and dependencies.
- **Entities** to store pieces of XML shared by several XML. like package informations.

$MOBYLEHOME/Local/Services/ you have following subfolder:

- **Programs** to store programs definitions.
- **Workflows** to store workflows definiotions.
- **Viewers** to store viewers definitions and dependencies.
- **Entities** to store pieces of XML shared by several XML, like packages or nucleic databanks available.
- **Env** to store some environment variables needed by some programs (BLASTDB . . .).

The deployment is a chainig of several steps:

[1] This tool will simply automate the validation of the XML, according to its grammar which is 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 program description.

1. get the XML (on your computer or the distant server for imported service) and the resolution of the entities.
2. service definition validation.
3. transform the definition to be ready for Mobyle use (to lower the burden of the portal).
4. web publication.

2.2.2 Configuration

The tool used to manage the services deployment is "mobdeploy". It allow to deploy or undeploy a service. To work, mobdeploy need some informations from the Mobyle configuration file (Config.py),

- which programs to deploy?
- have some informations about imported services?

So I will detail the Configuration directives related to the deployment. During the deployment we consider 2 kind of services the "local" services which correspond to some services executed on this server and the imported services which apear in your portal but are executed in an other Mobyle "server".

locals services

The locals services definitions are store in 2 repository (as describe above 2.2.1). But you can configure Mobyle to deploy only a subset of the definitions stored in these 2 repositories. This obtain by specifying 2 directives LOCAL_DEPLOY_INCLUDE which indicate which definitions must be deployed and LOCAL_DEPLOY_EXCLUDE which indicate which definitions must not be deployed. Mobyle evaluate LOCAL_DEPLOY_INCLUDE first followed by LOCAL_DEPLOY_EXCLUDE. These 2 directives are python dictionnaries with the 3 following keys: 'programs', 'workflows','viewers' and the values are lists of corresponding services to deploy or not. Joker can be used to defined a set of services. By default all definitions found are deployed.

```
LOCAL_DEPLOY_INCLUDE = { 'programs' : [ '*' ] ,
                         'workflows': [ '*' ] ,
                         'viewers'  : [ '*' ]
                       }

LOCAL_DEPLOY_EXCLUDE = { 'programs' : [ '' ] ,
                         'workflows': [ '' ] ,
                         'viewers'  : [ '' ]
                       }
```

Let me show how to configure Mobyle if I want to have a portal which proposes 'hmmalign', 'hmmconvert', 'hmmemit', 'hmmfetch' and 'muscle' as programs. I know it's a stupid example but it's just to just illustrate how to work this two directives. In my repository, among all definitions from pasteur_programs package I have: 'hmmalign', 'hmmconvert', 'hmmemit', 'hmmfetch', 'hmmscan', 'hmmsim', 'hmmstat'. Then I will write something like this:

```
LOCAL_DEPLOY_INCLUDE = { 'programs' : [ 'hmm*' , 'muscle' ] ,# will deploy all
                                                             # programs begining
                                                             # by hmm plus
                                                             # muscle
                         'workflows': [ '*' ] ,
                         'viewers'  : [ '*' ] }

LOCAL_DEPLOY_EXCLUDE = { 'programs' : [ 'hmms*' ] , # all programs begining by
                                                    # hmms will be remove from
                                                    # the list of program to
                                                    # deploy so 'hmmscan',
                                                    #'hmmsim', 'hmmstat' will
                                                    # not appear in my portal
                         'workflows': [ '' ] ,
                         'viewers'  : [ '' ]  }
```

When you deploy services, if 2 services have the same name, one in $MOBYLEHOME/Local/Services and the other in $MOBYLEHOME/Services, only the version in Local/Services will be deployed.

imported services

Instead of the local services there is no implicit mechanism to deploy imported services. All imported services which must be deployed must be explicitly specified. But we need more informations, for instance the name of the portal, where to find the definitions (repository) .... I give you an example of a configuration for an hypothetic Mobyle server called portal1 and if you want to import services from Institut Pasteur.

```
PORTALS={
        'portal1': {                              # this is a free label which
                                                  # will apear in your portal
            'url': 'http://mydomain.ext/cgi-bin', # the url where to find the
                                                  # portal.py
            'help' : 'user@mydomain.ext',         # where to have help about
                                                  # jobs.
            'repository':'http://rita.sis.pasteur.fr/', # where are located the
                                                        # sessions, jobs and
                                                        # services.
            'services':{
                  'programs': ['program1' , 'program2' , ... ], # the programs
                                                                # list to import
                                                                # from portal1
                  'workflows': [ 'workfA' , 'workfB' , ... ]    # the workflows
                                                                # list to import
                                                                # from portal1
                       } #the viewers cannot be imported.
             } ,
             # for instance if you want to import from pasteur Mobyle
             'pasteur': {
                       'url': 'http://mobyle.pasteur.fr/cgi-bin',
                       'help': 'mobyle@pasteur.fr',
                       'repository': 'http://mobyle.pasteur.fr',
                       'services': { 'programs' : [ 'pars', 'protopars', 'mix']}
        }
}
```

A more complete documentation about Mobyle betwork is available in mobyle_network_overview.pdf

mobdeploy usage

Before to run mobdeploy, you should make sure that your current user has the permissions to read and write the files that belong to the Mobyle user (e.g., apache user).
Usage: mobdeploy [options] cmd
Here are the available commands:

- **deploy**: deploy services available options [-s –server, -p –programs , -w –workflows , -v –viewers , -f –force , -V –verbose]
- **clean**: clean the repository available options [-s –server, -p –programs , -w –workflows , -v –viewers , -f –force , -V –verbose]
- **index**: generate indexes available options [-V –verbose]

And options:

- -s –server specify the names (comma separated list) of the servers on which the command is applied. The keyword 'all' mean all servers as defined PORTALS in Config.py
- -p –programs specify the names (comma separated list) of the programs on which the command is applied. The keyword 'all' mean all programs as defined in Config.py
- -w –workflows specify the names (comma separated list) of the workflows on which the command is applied. The keyword 'all' mean all workflows as defined in Config.py
- -v –viewers specify the names (comma separated list) of the viewers on which the command is applied. The keyword 'all' mean all viewers as defined in Config.py

- -V –verbose increase the verbosity (there is 3 level of verbosity: warning , info , debug . default is warning)

After deployement the services are store in a directory hierachy which have the following schema: The direcory root is : $MOBYLE_ROOT_URL/HTDOC_PREFIX/data/services/ in this directory we find one folder by server (your mobyle is called "local") all other folders are named as it's specified in PORTALS Config.py. In each server folder you will find subfolders: programs, workflows and viewers (only for local server), if some programs, workflows and viewers have been deployed for this server. And in these folders there are the definitions of the corresponding services.
A full description of mobdeploy is available in Tools/README.

2.3 Debug

There are 4 levels of debug for a program. The debug level is set in Local/Config/Config.py either for all programs (with DEBUG variable) or for only one program (with PARTICULAR_DEBUG). Set the debug level for your program to a value up to 0. With a debug level up to 1, all steps of commandline building: value received, value conversion, controls, and code evaluation, are logged in build_log file. Unfortunately this log file receives all logs from all programs and executions simultaneously. Hence, it can become quickly unreadable, and I recommend to set the debug level up to 1 only for one program at a time and hide this new program to the other users (if mobyle is accessible from multiple users) with the AUTHORIZED_SERVICES fields. When you set the debug level to 2, you can test the command line building but it's not executed. Thus, you can debug the XML even if the program is not installed on your platform or if its execution time is too long. Don't forget to reinstall the XML each time you modify it (python mobdeploy.py -s local -p myNewProgram deploy). Once the debuging phase completed you'll just need to remove the restricted access and set the debug level to 0.

Chapter 3

Upgrade

TODO
incompatibilite entre version pre 1.0 et 1.0 comment upgrader un XML version pre 1.0 en 1.0 compatible
dataformat interface