How to write a new Mobyle program interface

Mobyle 0.97

Contents

1 Overview

We use xml in different parts of Mobyle: to describe programs, to save informations about user workspaces - jobs and data (what we call session) -, and to store jobs execution informations. The grammar of these xml is described in relax-ng format. The schema is stored in the mobyle.rnc and mobyle.rng files, located at the root 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 programs are human made and are a central piece to integrate new programs in Mobyle. In this document I will give some keys to build the xml of programs 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 from these files.

2 Writing

A program is divided in two parts: the head which describes some generalities about the programs, and the parameters which describe the different options 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.

2.1 head

- **name** 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.
- **doc**
  - **reference** can be either text, xhtml or an uri which point to a doi.
  - **authors** can be either text or xhtml.
  - **doclink** any URI 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 is several doclink elements the button send the user to the bottom of the interface where the user can choose between them.
- The **category** element must be chosen carefully, because the program tree in the left panel is based on it. Each node of the classification is separated by a colon. If several categories are specified, the program will appear 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 customize the xml for your platform. It permits to specify an environment variable. Because this part is highly site-dependent, it is usually implemented with entities. e.g: you can specify the location of your blast databases with this mechanism.
  `<env name=''BLASTDB''>/usr/local/ncbi/db</env>`
  or path if the binary is not in usual path
  `<env name=''PATH''>/path/to/my/binary</env>`

2.1.1 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 factorize some parts of a set of descriptions, or specify site-dependent variables. If you have for instance 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. All you need to do is to make sure the XInclude elements are correctly formatted and the XInclude namespace is declared somewhere.

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/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>
```

2.2 Parameters

The parameters element can be located as a direct child of either the program tag, or of the paragraph tag. In the first case it includes all of the defined parameters for the program, whereas in the latter case it defines the parameters of a given paragraph. It can include itself parameters or paragraph elements.

2.3 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 attributes like:

- **argpos** that specify 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 take 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 allow 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 allow to factorize precondition code.
- **layout** the layout parameter 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>
```

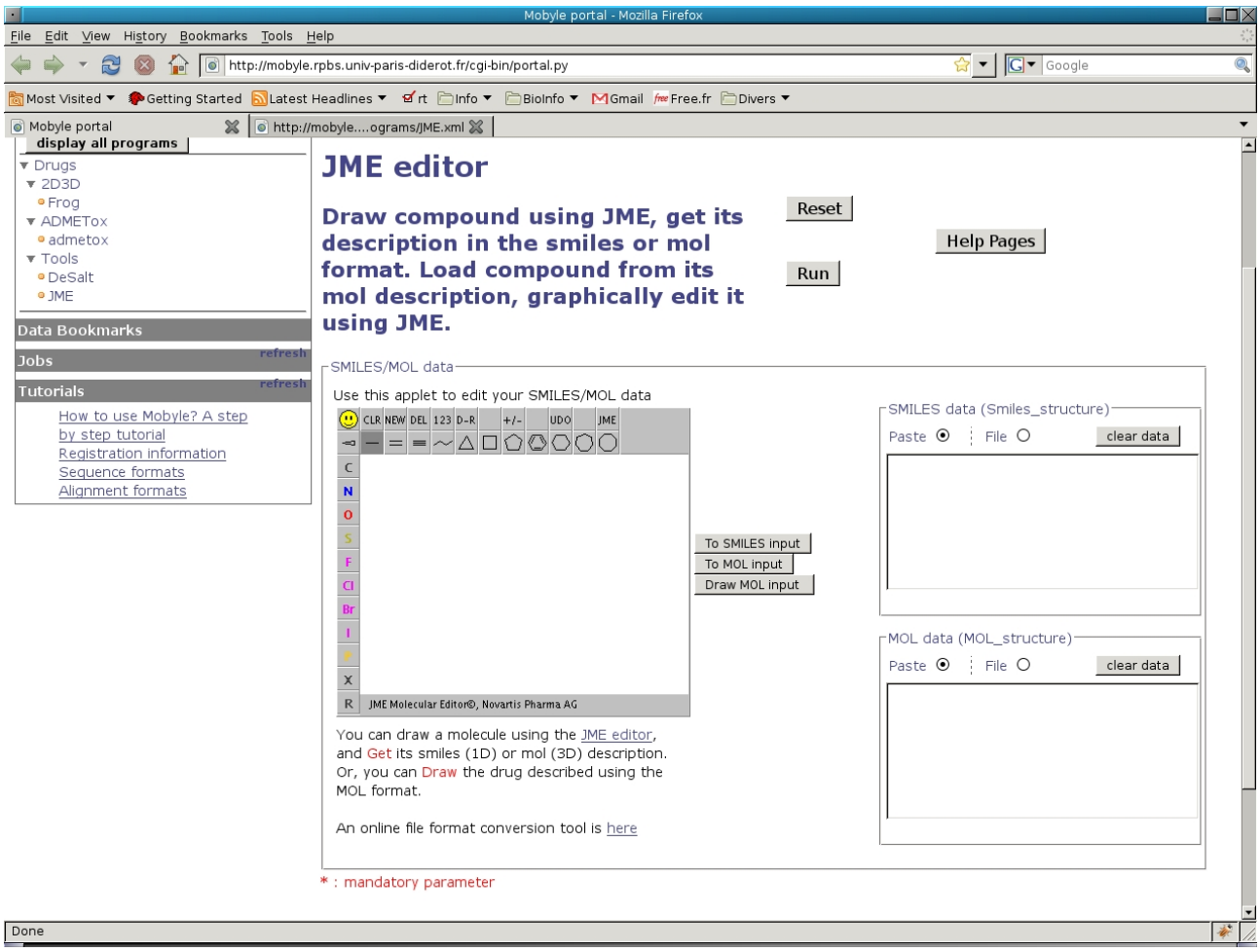

Figure 1: JME interface

2.4 Parameter

This is the element which allows us to describe the options, the inputs and outputs of a program. The information to build the command line is stored in this element. It's an essential piece of the program description.
The "parameter" element can have some attributes.

- **ismandatory** means that the parameter must be specified by the user. This kind of parameter is indicated by 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. This two implicit results are automatically shown to the user in the results page (if they are not empty) and are typed as Text. Some times we want to change the description of the standard output. To do this, we 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 isstdout attribute allows to change the type of the standard output from Text to the type of your choice. Current software limitations impose that the name of the output is as follows: {program_name}.out. Futhermore the file {program_name}.out is automaticaly captured from the standard output by Mobyle so you must not capture or rename this stream in any way in the command line (if you declare this parameter with the "isstdout" attribute). Examples like this one:

```
<format proglang=''python''>golden &gt; golden.out</format>
```

can cause some system-dependent side-effects, which will for instance prevent the index.xml file from listing this output in the job results (the result file will be in the job directory and seems correct at the end of the job but will not appear in the xml job summary (index.xml) and in the web interface.)

- 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 gradually generate the command line. Mobyle use python code but playMoby (http://lipm-bioinfo.toulouse.inra.fr/biomoby/playmoby/) uses perl code. By default the value resulting 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 Phylip suite (see pars.xml , protdist.xml, fitch.xml, ...). We may need to control finely the values provided by the users. The values must be less than a maximum value, be odd .... We can specify this with the **ctrl** element which contains evaluated code and a message which will display to the user if the code is evaluated to 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 defined how to retrieve the results produced by a programs. We use parameters with 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 file of hydrophobicity per sequence in the input file. the name of these file will be the name of sequence with '.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 paramfile can contains several code elements(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>
```

At last 2 elements which are not essentials to the command line building but which are very helpful for the users. **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.

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

- the **biotype** describe the biological object (Nucleic , Protein , Drug ,...). biotype 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 (specially on the spelling and case).
- **type** describes the computing object.
- **acceptedDataFormat** specifies the data format accepted by the program, hence the format in which Mobyle must convert (if possible) an input parameter value. For sequence it could be FASTA, EMBL ...any format managed by Mobyle. Several **acceptedDataFormat** elements could be specify.
- **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". We call it "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 refer 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 the 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.

2.5.1 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 (see typing paragraph).

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". The Sequences are analyzed by the tool from the SEQCONVERTER variable defined in Config.py. We recommend to use squizz instead of readseq as it may cause many troubles in sequence detection and conversion. The supported formats are those supported by squizz and/or readseq. Usually the parameter of Sequence datatype defined also elements acceptedDataFormats. Mobyle will try to convert the sequence in the first acceptedDataformated 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.

2.5.2 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/).

2.5.3 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.

2.5.4 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.

2.6 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.

2.7 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>
```

3 Installing and debuging

3.1 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.

3.2 Installation, Programs vs Local/Programs

The program installation process is described in the README file, located in the Tools subfolder of the Mobyle directory.

3.3 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.

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