OpenTEA Super-User Guide

A. Dauptain

December 16, 2014

Contents

1 OpenTEA

OpenTEA is a Graphical User Interface factory, developed at Cerfacs.

1.1 Description

The OpenTEA protocol is focused on scentific software with complex d.o.f. (degrees of freedom). This protocol is threefold:

1. Add to the software a set of metadata. These metadata are meant to ease the understanding, rise the possibilities and secure the setup of the software. These constraints often make the metadata largely different from the initial input data. This part is discussed in section 3.

2. Add to the software a set of successive setup processes. These processes are usually meant to translate the metadata into input file. Note that the same metadata can lead to several input files (for different versions of the same software).This part is discussed in section 4.

3. Add to the software execution a flexible encapsulation, to ease the porting and increase robustness. This part is discussed in section 5.

1.2 Licensing

The software is under the terms of Cecill-B licence, therefore Open Source , with any applications possibles, including commercial applications. The Cecill-B license is fully compatible with BSD-like licenses (BSD, X11, MIT) which have a strong attribution requirement (which goes much further than a simple copyright notice), a requirement normally not allowed by the GPL itself (which describes it as an advertising requirement). This license is also defined to make BSD-like and FSF's LGPL licenses enforceable internationally under WIPO rules

The explicit reference to the French law and a French court in the CeCILL licenses does not limit users, who can still choose a jurisdiction of their choice by mutual agreement to solve any litigation they may experience. The explicit reference to a French court will be used only if mutual agreement is not possible; this immediately solves the problem of competence of laws.

2 Installation, Getting started

2.1 Requirements: TclTk 8.5 , Python 2.6.6

OpenTEA is currently using a graphical engine written in Tcl/Tk 8.5. This limitation is motivated by use of the "Themed Tk" widgets (native appearance). Tcl/Tk 8.5 versions are almost always preinstalled on all flavours of Unix/Linux distributions. Using an older version of Tcl/TK would bring the following dialog at the start of the GUI:

```
wish8.4 opentea.tcl
Error in startup script: version conflict for package "Tcl": have 8.4,
need 8.5
    while executing
"package require Tcl 8.5"
    (file "opentea.tcl" line 4)
```

OpenTEA is currently using the XDR python library, a set of methods to give instant access to the memory of the GUI, either for reading or writing. This python library is customary for all OpenTEA subprocesses. The python versions must be over 2.6.6 , (one of the reasons is the `format` command which will fail on older versions. The python versions must not be 3 or higher, for they are not fully compatible with 2XX syntax. Using the wrong version of python would bring the following dialog in the log window:

```
 Saving Project As file /home/cfd1/dauptain/toto.xml
Script /home/rolex/c3smgit/library/lib_lite/scripts/process_tast.py
is completed.

Error in script
/home/rolex/c3smgit/library/lib_lite/scripts/process_tast.py :
Traceback (most recent call last):
  File "/home/rolex/c3smgit/library/lib_lite/scripts/process_tast.py",
  line 4, in ?
    import avbp
  File "/home/rolex/c3smgit/library/avbp/__init__.py",
  line 1, in ?
    from scripts import *
  File "/home/rolex/c3smgit/library/avbp/scripts/__init__.py",
  line 5, in ?
    from plugin_avbp import *
  File "/home/rolex/c3smgit/library/avbp/scripts/plugin_avbp.py",
  line 3, in ?
    import XDR
  File "/home/rolex/c3smgit/XDRpy/XDR.py",
  line 29, in ?
    raise Exception(
Exception: Must be run with python version at least 2.6.6,
 and not python 3
Your version is 2.4.3
```

☛ **Tip : Tcl/Tk errors reads top-to-bottom : first the actual problem, then the nesting of this error. On the other hand Python errors reads bottom-to-top : the nesting of this error down to the actual problem**

2.2 Installation

2.2.1 Default Installation

The usual installation of OpenTEA is a bundle of the graphical engine `opentea`, the XDR python library `XDR` and the repository of applications `library`. This last repository also carry the reserved `DATA` repository , with the "Plugins repositories" (see. Sec. 5)

```
$OPENTEA_HOME /
        /XDR
        /opentea
        /library
                /DATA
                        /pluginscripts
                        /toolPlugins
                        /codePlugins
```

In this case the startup script is similar to:

```
export PYTHONPATH="$OPENTEA_HOME/library/:$PYTHONPATH"
export PYTHONPATH="$OPENTEA_HOME/XDRpy/:$PYTHONPATH"
wish $OPENTEA_HOME/opentea/opentea.tcl
        -config $OPENTEA_HOME/../myconfig.xml
        "$@"
```

The PYTHONPATH environment variable must be explicitly set for both the library and the XDRpy folder. The `wish` command is finished by `"$@"` to enable additional keywords. The `wish` command must be the last of the script.

The `-config` keyword is compulsory in order to setup explicitly the actual config file of the user. Missing this argument rises the following console dialog, plus a popup window:

```
POPUP ERROR...
Error : keyword -config is compulsory to set explicitely your config file
```

On the other hand, a wrong path rises the following dialog on the console :

```
Error in startup script: can't read "configPath": no such variable
    while executing
"file copy -force $configPath $::env(HOME)"
    invoked from within
"if {[info exists key_config]} {
    set configPathtmp [file normalize [file join [pwd] $key_config]]
    if {[file exists $configPathtmp]} {
        set conf..."
    (file "/Volumes/Data/Users/.../opentea/opentea.tcl" line 166)
```

This tedious argument is motivated by numerous errors caused by implicit config files. The config file, usually named `myconfig.xml`, is similar to the following :

```
<?xml version="1.0" encoding="utf-8"?><dataset>

          <config value="">
                <gui value="">
                      <appearance value="">
                            <width value="1100"/>
                            <height value="1000"/>
                            <theme value="aqua"/>
                            <mode value="multicolumn"/>
                            <focusCorrection value="0"/>
                      </appearance>
                      <paths value="last">
                            <last value=""/>
                      </paths>
                      <python value="auto">
                            <auto value=""/>
                      </python>
                </gui>
                <id value="">
                      <user value="">
                            <name value="A. Dauptain"/>
                            <company value="cerfacs">
                                  <cerfacs value=""/>
                            </company>
                      </user>
                </id>
                <accounts value="">
                      <tool_plugins value="distant_unix">
                            <distant_unix value="">
                                  <login value="dauptain"/>
                                  <machine value="babylon"/>
                                  <directory value="/wkdir/cfd1/dauptain"/>
```

```
                                <nbprocs value="1"/>
                            </distant_unix>
                        </tool_plugins>
                    </accounts>
                </config>
            <meta/></dataset>
```

2.2.2 Custom Installations

A bit of flexibility is added when OpenTEA applications are used as production tools. They rely essentially on startup options :

- Explicit library position " -library " :To be used when several versions of the same library must coexist.
- Explicit plugins position " -plugins " :To be used when the plugins are NOT stored with the "library" folder.

2.2.3 Hiding OpenTEA, Adding private extensions

The "OpenTEA" banner and logo are a default appearance. It is possible to customize OPenTEA by redefining the logo, the name, and even add additional engine capabilities. In the case of "C3Sm", OpenTEA must be considered as a mere component powering the GUIs of the "C3Sm" software suite : the combustion community and Safran users do not know about openTEA Moreover, some extensions of the engine are "private", developed only for Safran group, and are not under the terms of Cecill-B licence.

In this case the GUIs are called with a simple variation of the initial script :

```
export PYTHONPATH="$OPENTEA_HOME/library/:$PYTHONPATH"
export PYTHONPATH="$OPENTEA_HOME/XDRpy/:$PYTHONPATH"
wish $OPENTEA_HOME/c3sm/c3sm.tcl
        -config $OPENTEA_HOME/../myconfig.xml
        "$@"
```

The Tcl/Tk pre-script `c3sm.tcl` is sourcing the core `opentea.tcl` after the surcharge of several variables and the declaration of one additional widget,`coolac` :

```
#  Ce programme depend des accords
l'ACCORD DE COOPERATION AVBP
(N. IFP 31.293).
Voir la fin du programme pour plus de details.

global additionalWidgets

set pathEngine [file normalize [file dirname [info script]]]

# Customization of C3Sm
image create photo icon_gui_small -file
        [file join $pathEngine IMAGES_PRIV logo_c3sm_small.gif]
image create photo icon_gui_tiny -file
```

```
        [file join $pathEngine IMAGES_PRIV logo_c3sm_tiny.gif]

# icones d'application
image create photo icon_hmg -file
        [file join $pathEngine IMAGES_PRIV logo_hmg.gif]
                [ . . . ]

# for the config and the footer of the app
# snecma
image create photo icon_snecma -file
        [file join $pathEngine IMAGES_PRIV logo_snecma.gif]
                [ . . . ]

#################################
# Additionnal widget must be of the form "create_mywidget.tcl",
#stored in folder SOURCES_PRIV
# The node "mywidget" will be automatically recognized
# widget for the coolant GUI
lappend additionalWidgets "coolac"

set banner {.         -------   ------     ------           .
.       (-------)(----- \  / -----)
.         -        -----) )( (----    ----
.       | |       (----- (   \---- \ |     \.
.       | |-----   -----) ) -----) )| | | |
.        \------)(------/ (------/ |-|-|-|}

source [file join $pathEngine "opentea.tcl"]
```

Using this pre-script, users know only about C3Sm instead of OpenTEA. The same setup can be done for any group of applications.

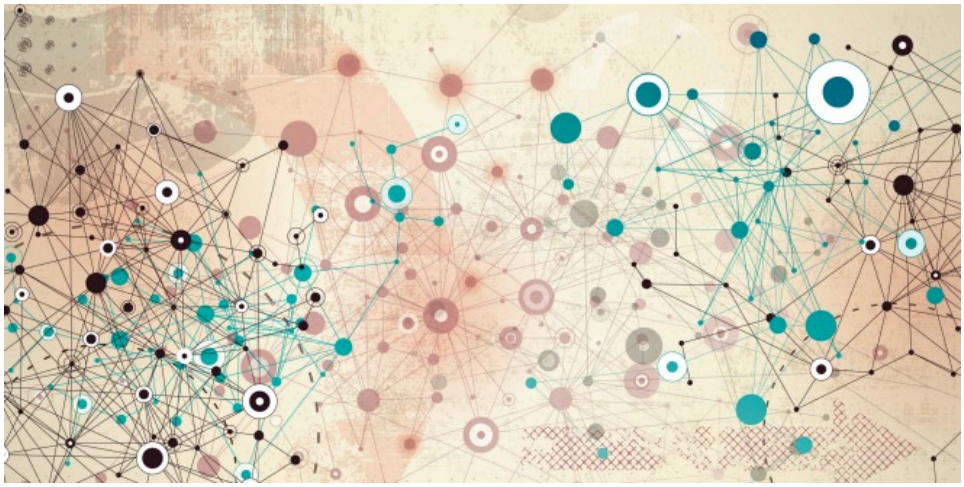
Installation hands-on

✍ **Basic install**

Use the material of Sec. 2 to install a distribution of OpenTEA on your computer.

✍ **Advanced install**

Create a custom library folder `mylibrary`, and several copies of the engine (`opentea` and `XDR` folders) to mimic several versions of the engine. Adapt the startup scrip in order to switch the engine versions while using the same library.



3 Data structure of OpenTEA projects

After a practical description of OpenTEA projects, a serie of hands on is proposed to get the following competencies :

- **load and save** From the trivial aspect of load and saving to the subtle handling of status.
- **investigate and debug** Handling ill-formed or corrupted OpenTEA projects.
- **off-GUI edition of projects** Creating a proper project without GUI.

This section ends with a detailed review of the principles that let to the actual OpenTEA data structure.

3.1 Getting started with OpenTEA Projects

OpenTEA project are an XML file `foo.xml` and a homonym folder `foo/` sharing the same location. The XML project is usually divided in two blocks within the `dataset` block : one block for the solver (e.g. `example`) and one block `meta` for the context of execution of the project. For a **full save** of a project, the XML file reads as :

```
<?xml version="1.0" encoding="UTF-8"?>
    <dataset>
        <exemple value="">
            <calculatrice value="">
                <basic value="">
                    <number_a value="30" />
                    <operator value="addition" />
                    <number_b value="12" />
                    <result value="42.0" />
                </basic>
            </calculatrice>
        </exemple>

        <meta>
            <solver>
                <name value="exemple" />
            </solver>
            <project>
                <name value="test_calc" />
                <address value="/Volumes/Data/Users/dauptain/
                        Documents/TEST_C3SM/test_calc.xml" />
                <username value="G. Hannebique" />
                <company value="cerfacs" />
            </project>
            <action>
                <callingAddress value="root.exemple.calculatrice" />
            </action>
            <temporary />
            <scriptSuccess value="1" />
            <engine>
                <name value="OpenTEA" />
                <launchCommand value="
                /System/Library/Frameworks/Tk.framework/Versions/8.5/Resources/
                        Wish.app/Contents/MacOS/Wish /Volumes/.../c3sm/c3sm.tcl
                        -config /Volumes/.../myconfig.xml
                        -plugins /Volumes/.../library/DATA/pluginscripts"
                />
                <pluginsPath value="/Volumes/.../library/DATA/pluginscripts" />
                <libraryPath value="/Volumes/.../library" />
                <configPath value="/Volumes/../myconfig.xml" />
            </engine>
        </meta>

    </dataset>
```

First note that some nodes do not have any attribute at all. Moreover the `example` block is only about the metadata content, nothing is known about the validity or the default value. The `meta` is essentially useful for debugging or datamining . It is also used in Python scripts for some introspection matters (see Sec. 4).

The XML file given to OpenTEA to create this application is :

```
<model name="basic" title="Simple Operation" >
    <param name="number_a" title="First number" type="double" default="30" />
    <choice name="operator" title="Operator" type="choice" default="addition">
        <option value="addition" title="+"/>
        <option value="soustraction" title="-"/>
        <option value="multiply" title="x"/>
```

```
    </choice>
    <param name="number_b" title="Second Number" type="double" default="12" />
    <info name="result" title="Result" type="double"  />
    <desc>
        This dialog triggers a simple computation
    </desc>
    <docu>
        How does it work ?
        [section=Graphical User Interface] The XML file content is read by
        C3Sm engine, and interpreted for the graphical display.[] The XML
         content being the one and only data source, it IS the source code
          of the application. It can be seen as a high-level, declarative
           programming language.
        [section=Execution] A python script is associated to this tab, with
        three steps:
        [item= It reads the data stored in the GUI.]
         [item= It make some operations.]
         [item= Il send back new data to the interface.]
         The first and last step are already part of the C3Sm engine.
         An application developper only focuses on the middle step.
    </docu>
</model>
```

One can see that all the specifications needed to build the GUI are present in this file. In particular parameters `number_a`,`operator`, `number_b` are all set by this file.

3.2 About statuses

During the GUI execution, the project is loaded in the memory in a Tcl array named "tmpTree", readting to all the user actions. Therefore, the "tmpTree" is the exact real-time image of the GUI content. When the user is is processing a TAB in the GUI, the valid "tmpTree" branches are dumped into an other array "DSTree", working as a saving point, filled only with consistent informations.

☛ **Tip : Both tmpTree and DS tree arrays can be browsed using the Debug>tmpTree (resp.Debug>DSTree) main menu.**

While looking into the tmpTree, one can see both visibility and status are also provided for each node. These information are never stored for the following reason : Neither the status, nor the visibility of a node are related to the *content* of a project. On the contrary, these two element must be recomputed at each startup. This is the only way to ensure that status are updated even when the application specifications have changed,.

The statuses at the startup of an existing project depends on the way it was saved:

- **Save All** The default saving of OpenTEA is a direct dump of the memory tmpTree. All informations are stored, even default values , spurious data and incorrect values option
- **Save Only Green** The cleanup Saving of OpenTEA : only the data related to green tabs are stored, red and orange tabs are ignored.

  ☛ **It is a good habit to use regularly the Save Only Green option during the lifetime of a project, to get rid of unnecessary data.**

XML projects hands-on

✍ **Project edition**

Open th `exemple` application `opentea -code exemple`. Process the tab and check the result $(30 + 12 = 42)$. Then quit using "Save all" option. Edit the XML project on the operator by replacing "addition" by "multiply". Re-open the XML project within opentea `opentea -code exemple -file test_exemple.xml`. The radio button must have changed.

✍ **Simplest application edition**

Edit the XML file of the application `exemple` named `calc.xml`, add the fourth operator "division". Edit the script file associated to the tab `process_calc.py` in order to handle the division operator. Re-open the XML project within opentea `opentea -code exemple -file test_exemple.xml` and test the extension to the division. Finally, modify the XML named `calc.xml` to prevent a division by zero.

✍ **Status modifications**

Open the library `lib_lite`. Process the first tab then exit using "Save Only Green" , making a file `dummy.xml` Check the content of the project : Only the data related to the first tab `xor` is stored. Reopen the project and check statuses : The first tab is green, others are still orange.

Exit again using the "Save All" and check the content of the project. There is also data for the last tab `defaults` Reopen the project : since data is filled and valid for the last tab, the status is green.

Exit again. Edit the `type` attribute of the parameter `dafaults/simple/real1` from `type=double_gt0` to `type=double_lt0`. Reopen the project and chaco that the status went to red at startup.

Have a break, stay calm and drink coffee.

✍ **Corrupted project salvage**

Open the library `lib_lite`, set the first XOR to the `beta` option,

process, save and quit. Edit the library `lib_lite/xor/xor` by renaming the model `beta` into `newbeta`. Reopen the former project. The engine will raise a popup error of the type :

```
POPUP ERROR...
"The option beta is no longer available in the XOR root lib_lite xor xor.
 Please remove this item from the XML"
Edit the project and remove the references to \texttt{beta} , then reopen the project.
```

3.3 Further reading... The graph theory behind OpenTEA

From the user point of view, the input parameters of a solver are intuitively clustered in groups and subgroups, However, the interdependencies between all parameters do not necessarily respect a hierarchical segregation. Both hierarchical vision and verification of dependencies are compulsory for an industrialized software: the user will use the first one to navigate through parameters, and the second one to know the implications of his actions. The best approach to describe hierarchical vision and verification of dependencies is to use graph theories.

First, a *parameter* is a variable to specify to the solver. A parameter can have many different natures: integer, real, boolean, choice, filename, coordinates. The ensemble of parameters, or *parametrization* defines a unique instance of the solver.

By definition [5], graphs describe the connectedness of systems and can help to create a formal model of parameter setups. A *general graph* is a set $\mathcal{V}$ of vertices with a set $\mathcal{E}$ of 2-subsets of $\mathcal{V}$ called edges. If the edges have an orientation so that they go from one vertex to another, they are *directed edges*, and the graph is a *directed graph*. In the present context, the vertices are linked to the parameters, and the directed edges to their dependencies. More precisely, each vertex includes a boolean information about the validity of the parameter. If the validity of parameter B needs to be tested when parameter A changes, the associated graph is $A \to B$. To ease the discussion, in the relation $A \to B$, A is the *child* of B and B is the *father* of A. Furthermore, validity is recessive, i.e. if $A_1 \to B$ and $A_2 \to B$ then B can be true only if both A_1 and A_2 are true. In other words, a parameter can be valid only if all its children parameters are valid.

The parametrization of an arbitrary CFD solver is shown as a directed graph in Fig. 1, taking into account the hierarchical dependencies only. In the hierarchical graph of Fig. 1, one can show that the n vertices are connected by exactly $n - 1$ edges by construction, since all parameters are grafted either to the root vertex (CFD solver) or to a pre-existing vertex. By theorem [5] this graph is a tree, i.e. a connected graph without circuits. This particular graph allows a very efficient data storage [2] used by all filesystem browsers.

A second graph sketched in Fig. 2 includes the cross-dependencies between parameters of different kinds. By construction, the n vertices are connected by more than $n - 1$ edges, excluding this graph from the tree family [5]. The descriptions of coupled parameters, like the choice between a tetrahedron-based numerical scheme and a hexahedron-based one are rep-

resented with a circuit (cf. example mesh file $\rightleftharpoons$ scheme of Fig. 2). [1]. These circuits rise the complexity of a graph.

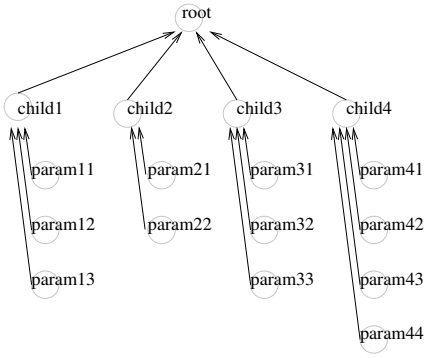


Figure 1: CFD solver information shown as a directed graph,17 vertices for 16 edges.

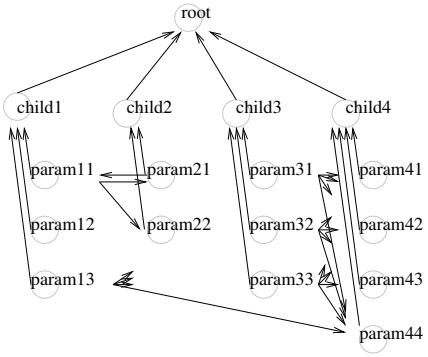


Figure 2: CFD solver information stored in as a directed graph including the cross-dependencies . 17 vertices for 35 edges.

The graph theory yields two conclusions:

1. The hierarchical part of the setup, or "the way the user comprehend the setup", can be implicitly modeled by the structure of a directed tree.

2. The cross-dependencies can link any parameters, and cannot be implicitly modeled by the structure of a directed tree. If the tree structure is used, these dependencies must be explicitly declared.

Note that the implicit modeling of hierarchy makes native the "error tracking": according to the graph of Fig. 1, setting the integer parameter "Dimensions" to 1.3 makes the vertex "false". The path:

Dimensions→Domain→CFD solver

is recursively set to false. The user can quickly track down the parameter blocking the whole setup using this highlighted path. This process is illustrated in Fig.3.

[1] Coupled parameters are common in scientific solvers because they gather the different aspects of the same approach. For example, the wall modeling and the sub-grid scale modeling is a reccurent couple in Large Eddy Simulation solvers.

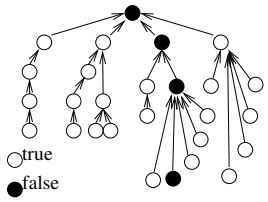


Figure 3: Path toward a invalid parameter using the directed tree structure. A non-validity is propagated to the ancestors. The search for the non-valid parameter among 27 possibilities is highlighted in 4 steps from the root.

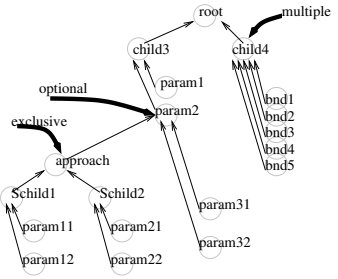


Figure 4: Three dynamic vertices: a multiple vertex for the boundaries, an optional vertex for single/two phase computations, and an exclusive vertex whose children are mutually exclusive.

The tree model considered until now is static, in the sense that no part of the graph can appear or vanish. The actual setup of a solver is more dynamic: the number of boundaries is not known in advance, some equations are optional, and some are mutually exclusive. Consequently, a supplementary property must be added to some vertices in order to allow the variety of setups, illustrated in Fig. 4. The exact property to add is discussed in the next section.

Operations on the tree model

3.3.1 Operations on the tree model

Once a tree model is set, its is possible to manipulate safely the solvers setups. Two major operations are defined:

1. the *tree grafting* operation is the addition of a tree B to a tree A by setting one node of A dependent of the root of graph B. This operation is needed for the setup of cycled (Fig. 5) and coupled (Fig. 11) computations. Indeed, a cycling simulation need some inputs specific to the cycling procedure in the left branch of Fig. 5, plus several instances of the same solver shown in the right branch of Fig. 5. The coupling configuration is identical excepted that the global setup need the instances of different solvers (right branch of Fig. 11).

2. The *tree pruning* operation is the removal of a vertex with all its children from the tree. A particular use of pruning is found when comparing two trees by *tree substraction*. The *tree substraction* $A - B$ is the pruning of vertices of A which are in B and do not have differing children. Figure 7 illustrates the concept, and enhance the noncommutativity of the operation. The *tree substraction* is useful to compare exhaustively two arbitrary setups of the same computation, and to solve frontward/backward compatibility issues.

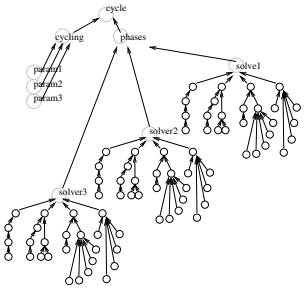


Figure 5: Grafting operation for a cycling setup: phases corresponding to several setups of the same CFD solver

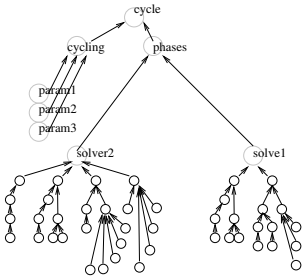


Figure 6: Grafting operation for a coupling setup: a CFD solver coupled with a thermal solver

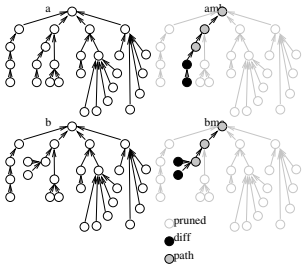


Figure 7: Pruning operation for tree comparisons. The operator "substraction" is noncommutative. Some vertices must be kept to find the path to the differences.

3.3.2 The Feature Modeling

The requirements or research softwares being known by the graph theory, are they compatible with the FM approach? Can a research software be regarded as Software Product Line (SPL) , i.e. a family of related programs. The basic Feature Model notation includes relationships between a parent feature and its child features:

- *Mandatory* child feature is required. Can be extended to multiple.
- *Optional* child feature is optional.
- *Or* one or more sub-features must be selected.
- *Alternative (xor)* only one of the sub-features must be selected

In addition to the parental relationships between features, cross-tree constraints are allowed. The most common are:

- A *requires* B The selection of A in a product implies the selection of B.
- A *excludes* B A and B cannot be part of the same product.

This basic model can be extended [6] by describing multiplicities of some mandatory (resp. optional) features: mandatory multiple means A must have 1 or more B children (resp. optional multiple means A can have 0, 1 or more B children). These six notations on a directed tree are sufficient to describe the parametrization of a research software. The parametrization of the Computational Fluid Dynamics Large Eddy Simulation code AVBP is showed thought a Feature Modeling diagram in Fig. 8. For the sake of clarity, only five out of the sixty boundary conditions available in AVBP 6.2β and only major parameters are shown.

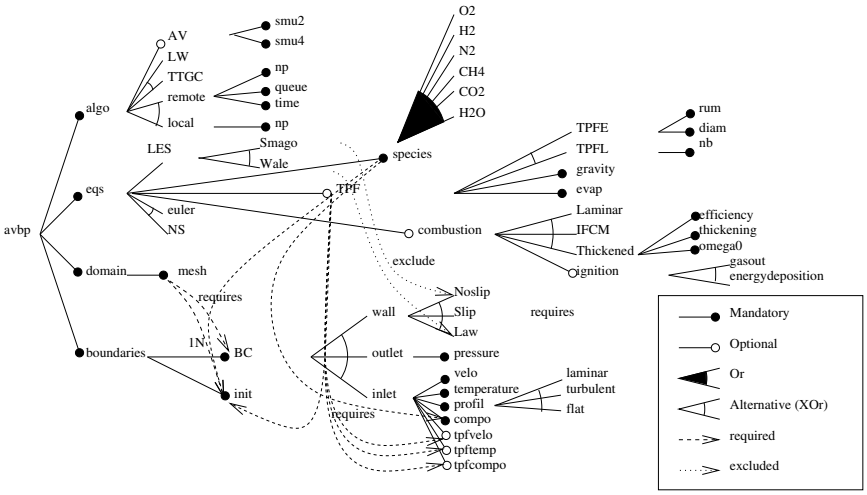


Figure 8: Feature Diagram of the CFD LES code AVBP 6.2β using the notation of Kang [7]. The diagram is simplified: more than sixty boundary conditions are available, and only the major model parameters are shown.

3.3.3 Practical implementation

On the way to simplify the communication between solvers teams and industrialization teams, Boucher *et al.* [3] suggests a text-based approach to

describe the softwares and shows an application of the concept to a family of printer drivers. This reduces the knowledge overlap to an exchange of text files. As the language needed to model research software will be used only reluctantly by solver teams, there is a strong constraint: this language must be "research-oriented" i.e. as explicit as possible, handled by the classical academic tools (editing with a *vi/emacs/xedit* console, grafting/pruning with a console *cp/mv/rm* or a browser, management with a CVS/SVN-like file manager). The perception of this language by researchers is the cardinal point which will condition the ease of solver teams to reach the GUI-compliant state.

A Domain Specific Language is therefore the best option, with a tree-shaped data structure, and the six notations of FM to enrich the nodes. The present section explains why an eXtensive Markup Language (XML) is a good candidate to this purpose, what vocabulary is necessary for markups, and why a scattered cloud of XML files is more suited to the present context. Afterwards, a quick overview on the GUI engine completes the picture of the methodology.

3.3.4 XML files with explicit Feature Model notations

XML is a set of rules [4] to write data structures. Each file is composed of structured elements. An *element* begins with a start-tag and end with an end-tag. Attributes can be attached to start-tags to describe the element and content is what is between the start-tag and the end-tag. If no content is written, the element can just be an only tag named empty-element tag.

XML is not primarily made to handle cycled graph and, as mentioned before, only the hierarchical part of the graph is stored in the XML file. The introduction of FM notations in the XML structure is a matter of vocabulary. A possible example is shown in Fig. 9. The six notations (mandatory /optional, or/Xor, require/exclude) are explicitly shown. In this example, both CFD and boundaries conditions are mandatory, but several nodes "boundary conditions" can exists. A verbose mode supported only in Euler resolution is secured by the required parameter. One can note that any choice (or/Xor) implies the creation of an intermediate node, which helps to store the choice result.

3.3.5 Scattering files

The description of a full solver in its entire complexity in one file is possible but heavy. A alternative is scattering the solver description into smaller files.

```
<model name="MySolver"  >
        <param name="CFL" type="mandatory" \>
        <param name="verbose" type ="optional" require="equations Euler" \>
        <param name="B.C." type="mandatorymultiple" \>
        <param name="passive scalar" type="optionalmultiple" exclude="species empty"\>
        <param name="species" type="or" >
                <choice name="hydrogen"\>
                <choice name="oxygen"\>
                <choice name="water vapor"\>
                <choice name="nitrogen"\>
        </param>
        <param name="equations" type="Xor" >
                <choice name="Euler"\>
                <choice name="Navier-Stokes"\>
        </param>
<\model>
```

Figure 9: Example of an XML file with the Feature Modeling (FM) notations

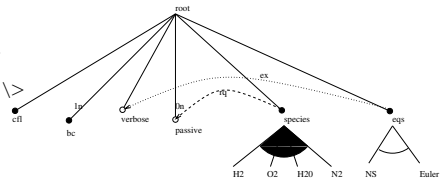


Figure 10: FM diagram associated to the XML-FM file of Fig. 9

As files are stored in a computer within an arborescence, the tree structure of the arborescence can be used for in the description of the global tree using the "grafting" operation. The advantage of storing the description of each model into a specific file is twofold:

1. Like any source code subroutine in academia, these small files can be edited with lightweight editors (e.g. vi), managed by release managers (e.g. SVN/ CVS), and installed/uninstalled from a console or a file browser. Adding simply one file to the arborescence gives more autonomy in a trial-and-error attempt. In all aspects, researchers can interact with these files like they do with source code.

2. Contractually, each of these files becomes the deliverable entities associated to the industrialization of the associated model. Each file has its own traceability, and its own confidentiality properties. The deliverable is paid to the solver team, redirecting the industrialization funding towards the scientific team.

3.3.6 Toward multi-physics applications

A multi-physics application can be addressed either by a multi-physics solver (monolithic approach) or by several dedicated solvers that exchange boundary conditions (coupled approach). Industrialization of a monolithic approach is straightforward with the present methodology, but a coupled approach rises new issues. Conjugate Heat Transfer (CHT) problems treated by coupled legacy codes are a good illustration of these issues. This solution has the advantage of using existing state-of-the-art codes to solve fluid

and solid equations and of being able to exchange one solver with another easily. The main drawback of this coupling methodology is that an adapted CHT framework is requested for the simulations especially on parallel machines. The performances of such a coupling framework are linked to (1) the strategy to couple the solvers in an accurate and stable fashion and to (2) the exchange of information between the solvers in an efficient and scalable fashion when using a large number of processors.

Point (1) imposes to be able to extract and to impose information in the legacy codes during the computation at given times. This work is done by collection of empty routines, or *User Defined Functions*, i.e. called at strategic places. The success of point (2) relies on a coupling library able to:

- efficiently connect coupled geometric interfaces (meshes or sub part of meshes) of parallel solvers distributed on a large number of processors,
- produce high quality interpolations of exchanged data.

The OpenPALM coupler [1] co-developed by CERFACS and ONERA tackles these issues. It is used to control AVBP for the resolution of the fluid part and AVTP[2] for the resolution of the conduction in solids. In addition to the AVBP and AVTP parameters, a set of *coupling parameters* has to be specified : the frequency of meeting points for data exchange, the number of meeting points, the location where information need to be exchanged, the type of information to exchange and some parameters for the interpolation.

The present industrialization methodology can be extended to the coupling of legacy code by *grafting* the solvers tree to an application-specific coupling tree, as illustrated in Fig. 11. Note that some exclusion/requirements will be necessary: impose temperature from fluid to solid and also temperature from solid to fluid for example will lead to exchange always the same quantity and ... no convergence.

A snapshot of the corresponding GUI is given on Fig. 12.

[2]Parallel thermal solver developed at CERFACS.

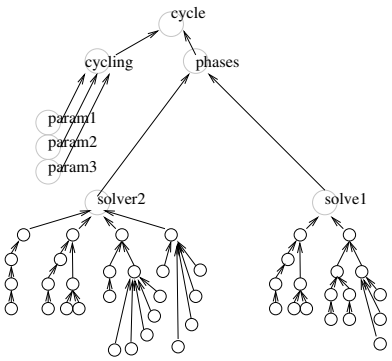


Figure 11: Grafting operation for a CHT coupling setup: a CFD solver coupled with a thermal solver

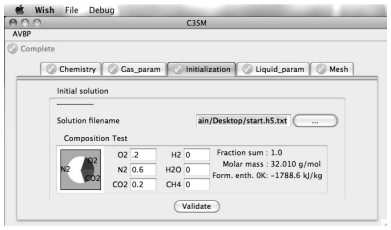


Figure 12: Snapshot of the C3S**M** graphical user interface, which must able to setup AVBP ,YALES2, AVSP simulations as well as AVBP/AVTP coupling with a reduced industrialization weight on academia with respect to the C3S project.

4 Processes for the setup: Python scripts

The XML specifications must be completed with actions. In OpenTEA these actions are done through Python >2.6.6 scripts. The Numpy extension being now almost systematically associated to Python distributions, on can use it safely within OpenTEA. However, if an OpenTEA application require other packages such as Scipy, the developer must mention explicitly this dependency, and include an explicit error handling about this topic in his scripts.

4.1 The link between the GUI and the Python scripts

Scripts are associated to the application on tabs :

```
<tab name="xor" order="1" title="Test XOR" script="process_xor.py"/>
```

The script must be present at the startup, else the following error is raised.

```
Error in startup script: File not found :
/Volumes/.../library/lib_lite/scripts/process_multiple.py
```

The script can be written in a standalone version:

```
from XDR import *

init()
xor =  getValue("xorchoice")
if xor == "alpha":
    value = getValue("temperature")
    print "temperature",value
if xor == "beta":
    value = getValue("pressure")
    print "pressure",value
finish()
```

The following structure is compulsory :

```
from XDR import *
init()
 [ actual actions ]

finish()
```

- `from XDR import *` to import the OpenTEA memory module.
- `init()` to download the memory of the GUI (tmpTree) in the python context
- `finish()` to upload the altered memory of python context into the GUI

The actual actions in the script are handled by classical python actions. Note the memory access using the XDR function `getValue(-node-,-optional disambiguation-)`. The memory alteration is done using the reverse XDR function `setValue(-value-,-node-,-optional disambiguation-)`.

4.2 Batch execution of scripts

The reusable scripts are written this way :

```
from XDR import *

def process_xor(ds):
    xor =  ds.getValue("xorchoice")
    if xor == "alpha":
        value = ds.getValue("temperature")
        print "temperature",value
    if xor == "beta":
        value = ds.getValue("pressure")
        print "pressure",value

if __name__ == '__main__':
    init()
    process_xor(getDsout())
    finish()
```

In this pattern, the function `process_xor` is callable ether by the GUI, or by a separate python script, which could be read as follow :

```
from XDR import *

import process_xor
import process_multiple
import process_defaults

dsin,dsout=init("./dummy.xml")

process_xor.process_xor(dsout)
process_multiple.process_multiple(dsout)
process_defaults.process_defaults(dsout)
```

Python scripts hands-on

✍ Errors while getting/setting values

The `lib_lite` application is voluntarily ill-formed, with tree parameters named `real1` : one in the `multiple` tab, two nested in the `defaults` tab. This redundancy will illustrate how XDR find the correct values in the memory of the application, even if other share the same names.

In the `lib_lite` application sources, open the python script `process_default.py` . The file should be similar to :

```
from XDR import *
def process_defaults(ds):
    print "Processing defaults..."
    value = float(ds.getValue("real1","dataset","lib_lite",
        "defaults","simple","add"))
    print "Real 1 is",value
    ds.setValue(value+3.1416,"real1","dataset","lib_lite",
        "defaults","simple","add")
    #ds.setValue(value+3.1416,"real1")
    pass
if __name__ == '__main__':
    init()
    process_defaults(getDsout())
    finish()
```

Remove the optional arguments on the `ds.getValue` until the process crashes. Try the same exercise on the `ds.setValue`.

The same exercise can be done while changing the target address in the XML specification. For example, rename the `real1`parameter of the tab `defaults` into `real1_dummy`. The process should yield to explicit error messages.

✍ Using OpenTEA python scripts in batch mode

Create a python script with a python loop able to relaunch the 3 scripts of `lib_lite` 100 times, changing the prameter `temperature` each time. This can be used either to setup and launch parametric runs, or to automatically regenerate the input files from an existing project without the GUI.

4.3 Further reading... Behind the scene, the execution process

The execution of scripts is done in the context of a Tcl pipe (file `open pipe`). With `$address` being the tree address of the node (usually the corresponding tab in the GUI) and `$execCommand` a command usually of the form `python my_script.py`, the pipe is started with the following lines :

```
set widgetInfo($address-actionChan) [open "| $execCommand" "r+"]
 fileevent $widgetInfo($address-actionChan) readable "readPipe $win $address"
```

The action Channel is then read in real time by :

```
proc readPipe {win address} {
    global widgetInfo DStree metaTree
    if {[gets $widgetInfo($address-actionChan) line] >= 0} {
       [....]
```

The python script is executed in the context of this channel. Any standard output like `print "hello world"` will be redirected to the log of the GUI. The key bind "Esc" send the signal to close the current channel. On some architecture, this is enough to stop the process, but brute force os better:

☛ **As the process is a child of openTea, an external stop of the child process such as `kill -9 pid` in unix will result in a simple error execution on OpenTEA, giving back the control on the GUI. Do not be afraid to kill zombies...**

5 Code execution: Plugins scripts

OpenTEA handles code execution through a peculiar layer of scripts : the pluginsScripts. After a brief look on the `XDR.execute` , which should replace any use of python `execute` inside OpenTEA scripts, this section focuses on the global approach of pluginsScripts.

5.1 The bare XDR.execute command

This command replaces the classical Python `execute` with a proper analysis of the command given in argument, and an adjustable level of verbosity. Note the use of `subprocess.Popen` and not `execute`, which gives a realtime output channel.

```
def execute(command, always_print_err=False, silent=True):
    """
    This procedure searches for the specified executable in the script directory,
    if not, it tries to execute the command itself.
    The command is then executed and its output is printed in standard output
    """
    ### Need some work to handle long run and reading of the output on the fly !
    if (os.path.exists(os.path.join(scriptDir,command))):
        command=os.path.join(scriptDir,command)
    print "command "+ command
    command=shlex.split(command)
    if silent == False :
        print "Executing " + repr(command) + ' in ' +
              repr(os.getcwd()) + ':\n' + 50*'-' + '\n'
    read_from = None
    if "<" in command :
        read_from = command[-1]
        command = command[:-2]
    p=subprocess.Popen(command,stdin=subprocess.PIPE, stdout=subprocess.PIPE,
        stderr=subprocess.PIPE)
    if read_from:
        p.stdin.write(open(read_from, "r").read())

    stdout_data = []
    if silent == False :
        print "\nXDRExecute =============StdOut================\n"
    while True:
        line = p.stdout.readline()
        if not line:
            break
        if silent == False :
            print 'XDRExecute ' + line.rstrip()
        sys.stdout.flush()
        stdout_data.append(line)
    returncode = p.wait()
    stderr_data = p.stderr.read()

    if ( always_print_err and not returncode ):
        if silent == False :
            print "\nXDRExecute =============StdErr================\n"
            print 'XDRExecute ' + "\nXDRExecute ".join(stderr_data.split('\n'))
    # if traite "None" "0" False" "" Comme des retours negatif
    if returncode:
        error("Problem while running command :"+" ".join(command)+
              "\n=============StdErr================\n"+stderr_data)
    return "".join(stdout_data)
```

5.2 pluginScripts contents

A pluginScript is a pythonScript dedicated to the use of a specific resource by OpenTEA. The same plugin is used by all users for all applications on this resource.

A typical plugin scripts shows the following structure

1. The initialisation part
   - getting values in the GUI for the selected plugin
   - checking the connexion
   - checking the distant folder
2. The declaration of supported application, using the decorator line `XDR.supported_applications`
3. The distantCommand script
   - global initializations
   - 3 lines of "############"
   - application-specific commands
   - 3 lines of "############"
   - sending the directory gathering the necessary files
   - execution of the command via `ssh`
4. The retrieveDirectory script
5. The removeDirectory script

This structure is the most general. When the plugin with a local execution on the same resource as OpenTEA, many simplifications can be done:

1. The initialisation par
   - getting values in the GUI for the selected plugin

2. The declaration of supported application, using the decorator line `XDR.supported_applications`
3. The distantCommand script
    - global initializations
    - 3 lines of "############"
    - application-specific commands
    - 3 lines of "############"
    - execution of the command
4. The retrieveDirectory script (void)
5. The removeDirectory script (void)

An actual plugin for a distant execution will look like :

```
class myplugin(XDR, Plugin)
    def __init__(self,typePlugin)
        [ initialisation by getting the data from the GUI,
        testing the connexion,
        and creating the distant folder]

   @XDR.supported_applications(['tool_avsp52','tool_avbp621'])
   def executeDistantCommand(self,command,execDirectory,appli,flags=[]):
                    print "Plugin : Running executeDistantCommand "+command+"
                in "+execDirectory+"("+appli+")"
        ########
        # INITS
        #########
        hostname = "KALI"
        pythonexec= "/usr/bin/python"
##################################################
##################################################
##################################################
        #########
        # AVSP #
        #########
        if appli == "tool_avsp52":
            avsp_home =  "/home/rolex/QUIET_5.3/AVSP_HOME"
            # Temporary, bug with axisym duplication in HIP v1.41.0
            hip_cur_version = "/home/rolex/HIP/1.40.1/hip-1.40.1-"+hostname
            avsptool = plugin_avsp(avsp_home,hostname,hip_cur_version,
                pythonexec,execDirectory)
            command_exe = avsptool.switch_avsp_tools(appli,command)
        ########
        # AVBP #
        #########
        if appli == "tool_avbp621":
            avbp_home = "/home/rolex/AVBP_V6.X/AVBP_D6.2.1"
            avbptool = plugin_avbp(avbp_home,hostname,hip_cur_version,
                pythonexec,execDirectory)
            command_exe = avbptool.switch_avbp_tools(appli,command)
##################################################
##################################################
##################################################
        if command_exe.startswith("-c3sm_auto_"):
            XDR.error("command was not understood: "+command_exe )
        ######################
        # SENDING DIRECTORY #
        ######################
        print "Final composition of c3sm_archive:"
        print self.dir2send
        XDR.ssh_send(self.machine,self.login, self.distantDirectory,
                self.dir2send, options="")
        ######################
        # EXECUTING COMMAND #
        ######################
        sshCommand= "cd "+self.distantDirectory+"/"+execDirectory+"; "+command_exe
        # NB : -X allows here an interactive action
        output = XDR.ssh(self.machine,self.login, sshCommand, options="-X")
        # back to the initial directory
        os.chdir(local_directory)
        return output

    def retrieveDirectory(self,directory):
        local_directory = os.path.abspath(directory)
        dist_directory = os.path.basename(local_directory)
        # Retrieve Directory
        if os.path.exists(local_directory):
            shutil.rmtree(local_directory)
        scpCommand = "rsync -a "+self.login+"@"+self.machine+":"+
                self.distantDirectory+"/"+dist_directory+" "+
                os.path.dirname(local_directory)
        XDR.execute(scpCommand)

    def removeDirectory(self,directory):
        sshCommand = "/bin/rm -rf "+self.distantDirectory+"/"+directory
        output = XDR.ssh(self.machine,self.login, sshCommand, options="-X")
```

Several commands from the XDR library help to keep commands as simple as possible. Note the XDR.execute, XDR.ssh_send (resp. XDR.ssh_retrieve) commands, which are all higher level commands than they look like.

Note also that in this plugin, the path to the actual executables are totally explicit. Il is also possible to rely on environment variables, however the debugging with become a bit harder (the actual path is stored elsewhere) and the environment variables of the python context of a subprocess can become extremely hard to control.

5.2.1 More about XDR.ssh_send

This command encapsulated the action "send this directory there using `ssh`" . Note that the actual transfer is done with a `tar` command before and after. There is no temporary files, the output of the `tar` is redirected to the `ssh` via a pipe.

```
def ssh_send(host, login, distant_directory, local_directories_list, options=""):
    """ Uploads files to a server using ssh.
        This creates a tar archive, and pipes it through
        ssh to a 'tar xf' on the distant side.
        In short, the command that we run is:
        tar cf - directory1 directory2 | ssh distant_server
         "tar xf - -C distant_directory"
        (with some additional safety)
    """
    # TODO: check that we are on a platform where tar, scp, ssh actually exist
    # TODO: Check that local_directories_list is actually a list.
    # People will try with string, and it does bad things with strings.

    full_host = ssh_host(host, login)

    if (len(local_directories_list) == 0):
        print "No directories to be sent"
        return
   print "Sending and extracting archive..."
    command = """bash -c "tar cvf - """ + (" ".join(local_directories_list))
        + """ | ssh """ + full_host + """ \\\"tar xf - -C """
        + distant_directory + """\\\" " """
    #print "ssh_send command ",command
    execute(command, always_print_err=True)
```

The basic requirement of this strategy is to have an access via `ssh` without typing keyword, i.e. with a RSA or DSA authentication key. Keywords could be handled in OpenTEA using Expect, but this would open an extremely dangerous security weakness on all the resources.

☛ **If the user distant profile (`.profile` or the likes) is set in a way that some text is sent to the Standard Output as soon as the ssh is called, this text will be treated as an error code and will eventually crash for any `ssh` attempt from OpenTEA**

5.3 pluginScripts use

The pluginsScript are used in OpenTEA in the following pattern:

```
[...]
temp_path_name = "tmp_track"
temp_path = ensureDirectory([temp_path_name],clean=True)
[...]
plugin = loadToolPlugin()
plugin.sendDirectory(temp_path_name)
plugin.executeDistantCommand("-c3sm_auto_track-",temp_path_name,"tool_avbp621")
plugin.retrieveDirectory(temp_path_name)
plugin.removeDirectory(temp_path_name)
[...]
```

In other words, a temporary folder is created , gathering all the necessary data. The plugin is loaded accordingly to the user setup (see `myconfig.xml`). The file is sent, executed, retrieved and cleaned.

Note the two keywords `-c3sm_auto_track-` and `tool_avbp621` in the arguments of `executeDistantCommand`. These keywords must meet their counterparts in the plugins. ☛ **in the case of a local execution, the script is exactly the same, but the commands `plugin.sendDirectory` and `plugin.retrievedDirectory` are void,**

A slightly more complex pattern is

```
[...]
temp_path_name = "tmp_track"
temp_path = ensureDirectory([temp_path_name],clean=True)
[...]
plugin = loadToolPlugin()

dum1,XDR_dir_path,dum2 = imp.find_module("XDR")
shutil.copy(XDR_dir_path,temp_path)
shutil.copy(os.path.join(getScriptDir(),"script_track.py"),temp_path)

plugin.sendDirectory(temp_path_name)
plugin.executeDistantCommand("-c3sm_auto_track-",temp_path_name,"tool_avbp621")
plugin.retrieveDirectory(temp_path_name)
plugin.removeDirectory(temp_path_name)
[...]
```

In this last pattern, an OpenTEA script `script_track.py` is sent together with the XDR python library to take care of the execution. This is useful when the distant action is a sequential execution of several FORTRAN executables : data is send once for all a the beginning, and the debugging is far easier .

☛ **to debug a distant script, rename the file `out_dateset.xml` into `dataset.xml`, then re-interpret the script using `python script_track.py`. This way, you can execute interactvely the distant script on the final resource several times in a row.**

pluginScripts hands-on

✍ **The first FORTRAN execution**

Execute the following fortran program from the application `exemple` script `process_calc.py`, assuming the file "squared.choices" is already in the working directory :

```
program squared
open(10,file="squared.choices")
read(10,*) value
close(10)
result = value**2
open(20,file="squared.out")
write(20,*) result
close(20)
end program
```

The use of the command `XDR.execute` is compulsory.

✍ **FORTRAN giving data to the GUI**

In the same script, read the data from `squared.out` and fill the `result` GUI field with this data.

✍ **FORTRAN getting data from the GUI**

In the same script, write the file `squared.choices` with the content of `number_a`. The GUI is now operational for a FORTRAN square computation.

✍ **Creating your own Plugin**

Use the templates PluginsScripts stored in folder`DATA` to create your own pluginScript adapted to your resource. The usage can be local or distant. Do not forget to create the associated XML and to set your config file on this Plugin Adapt your square computation in order to use the script. The use of the command `XDR.executeDistantCommand` is compulsory.

References

[1] A. Thévenin A. Piacentini, T. Morel and F. Duchaine. O-palm : An open source dynamic parallel coupler. In *IV International Conference on Computational Methods for Coupled Problems in Science and Engineering - Coupled Problems 2011*, Kos Island, Greece, June 2011.

[2] A.V. Aho, J.E. Hopcroft, and J. Ullman. *Data structures and algorithms*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[3] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing tvl, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January*, pages 27–29.

[4] T. Bray, J. Paoli, and CM Sperberg-McQueen. Extensible markup language (xml) 1.0. 1999.

[5] P.J. Cameron. *Combinatorics: topics, techniques, algorithms*. Cambridge Univ Pr, 1994.

[6] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. *Software Product Lines*, pages 162–164, 2004.

[7] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis feasibility study. *Software Engineering Institute, Pittsburgh CMU/SEI-90-TR-21*, 1990.