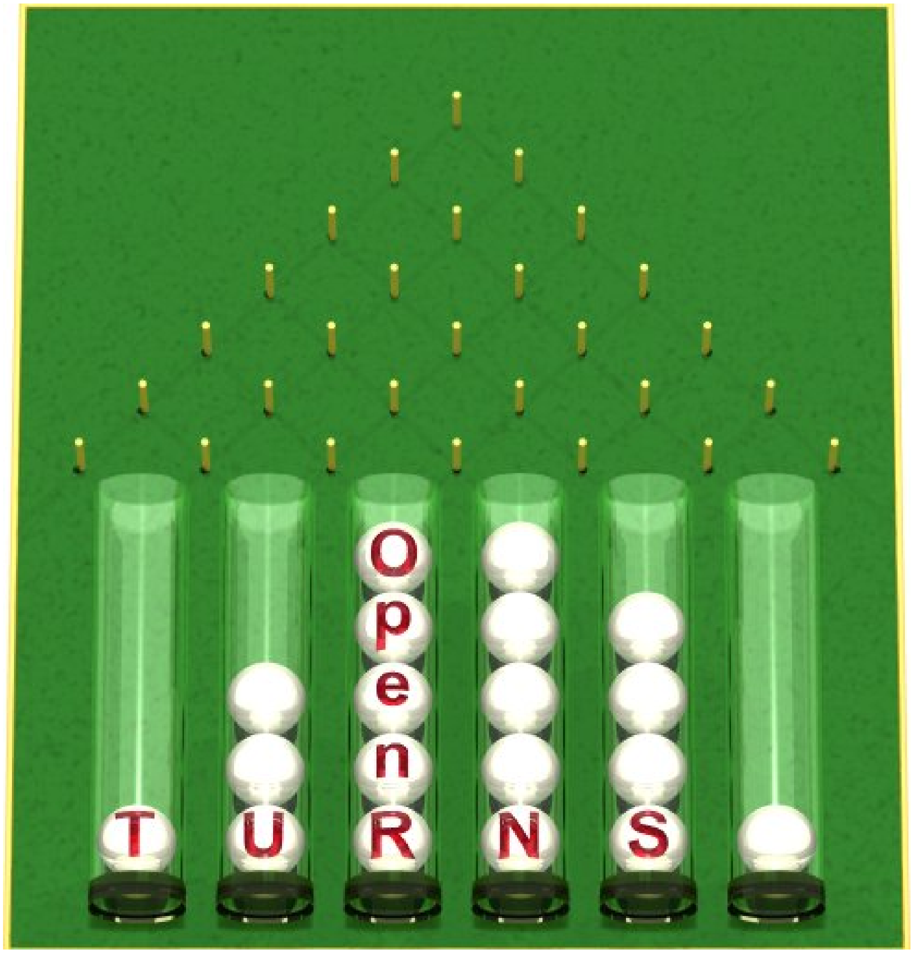
Wrappers guide

Open TURNS version 0.12.1

November 8, 2008



Contents

1 Introduction

This guide aims at facilitating the use of Open TURNS with an external code. This document is adressed more to a *super user* of this platform at the time of the deployement and the connection with domain's tools) than a classical user. Open TURN's is a non intrusive tool for external code. The connection with this tool must be done through a wrapper. This Wrapper is a C++ code with an interface specified by Open TURN'S.

This platform is suited for two kinds of use:

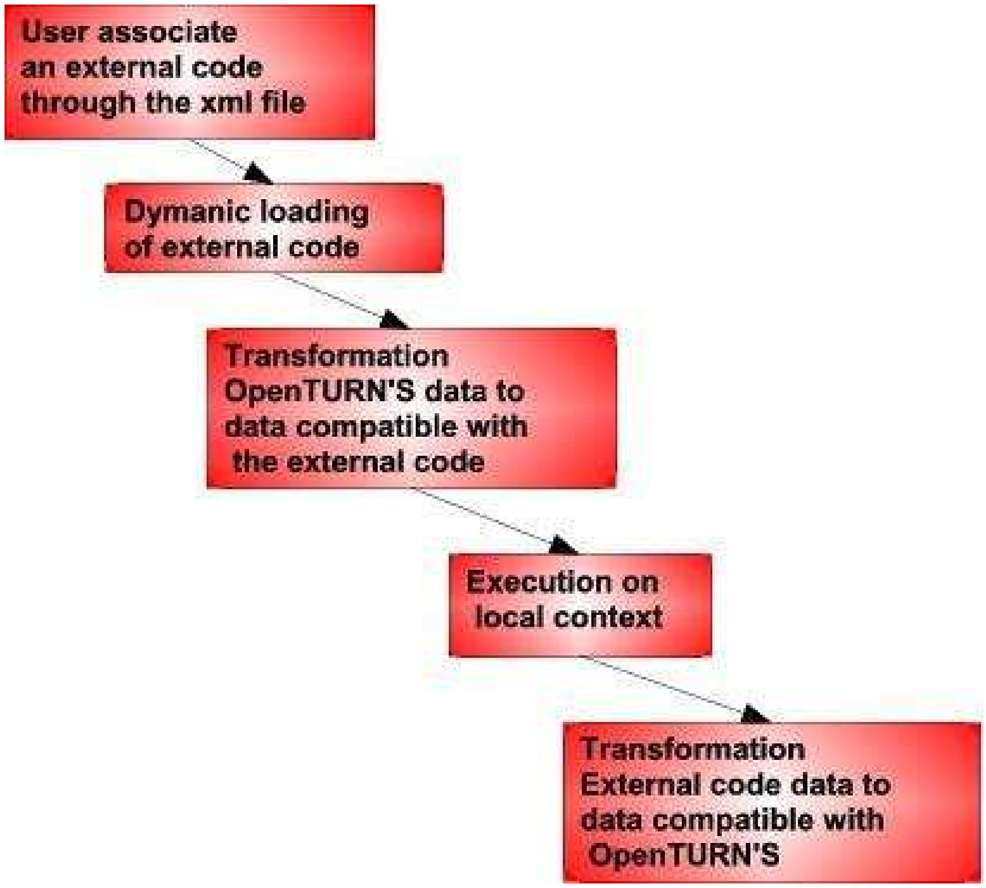
- **In a local context**:
  When the external code (domain software of interest) does not require lots of resources (in terms of memory or cpu), it is better to use your computer to run the computations.
  In this document, after the description of the principles on which the external code is plugged in the platform, you can find an example of this implementation for a toy case (one example called ”poutre” is distributed with the software).

- **In a distributed context**:
  We explain how it is possible to use a network to improve the performances of this software for ditributed computation. Some recommandations and restrictions are mandatory to use this function (especially the interface with the batch manager). One example of implementation can be found at the end of this chapter.
  In the last part, a short paragraph describes the use of external code with a parallel implementation. In this particular case, the goal is to run the computations with a grid of cluster (each distributed computation can be run with several processors).

2 Local Context

2.1 Principles

In OpenTURN's, you can use any software to compute the solution for one realisation of the set of parameters. If this computation does not require a lot of ressources (memory or cpu), you can execute this run on your own computer. In this context, your software must be seen by OpenTURN'S like a set of API in a dynamic libray. Two files are mandatory for OpenTURN'S: the first is a decription file in xml format, the second (referenced by the first one) is the dynamic library.


User associate an external code through the xml file
Dymanic loading of external code
Transformation OpenTURN'S data to data compatible with the external code
Execution on local context
Transformation External code data to data compatible with OpenTURN'S

The xml file gathers a set of informations:

1. the path of the dynamic library
2. the inputs necessary for a standard computation (for example a mesh file)
3. the association between the OpenTURNS world and the External code world for all potential uncertainty variables
4. the list of symbol defined in the dynamic library

WARNING:
By convention, the order of the description in xml file defines the same order of the uncertain variables in the input vector defined in Open TURNS

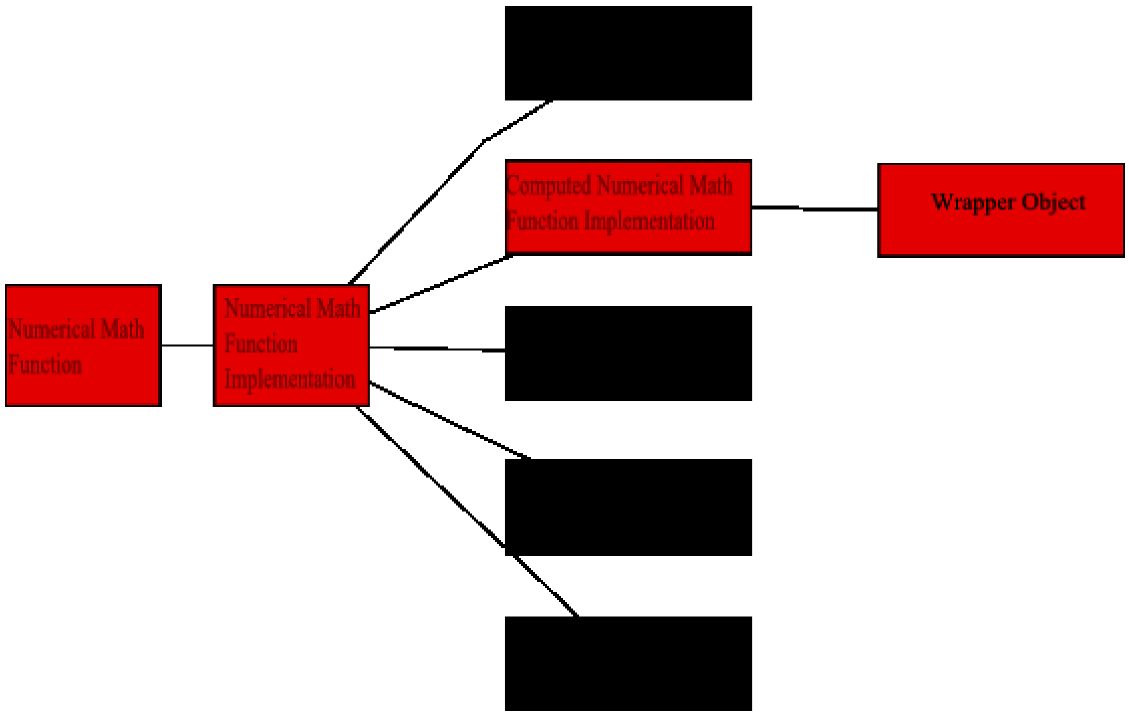
In the dynamic library we must find at least this API:

1. func_getinfo_*MySymbol*(): define the number of parameters (input and output), this definition must be coherent with the description in the xml file.
2. func_init_*MySymbol*(): with this API, you can define some operations necessary for a right execution of the external code (for example, directory creation)
3. func_exec_*MySymbol*(): This is the major API with three functions:
   - To translate OpenTURN'S data into External code data,
   - To run the external code (through an api, through a system call, ...),
   - To translate the External code data into OpenTURN'S data.
4. func_finalize_*MySymbol* (): The goal of this API is to finalize properly the execution (free memory, destroy temporary file, ...).

In this library, you can define two optional API:

1. func_createState_*MySymbol* () : Save a state between two sequential executions of two sets of input parameters,
2. func_deleteState_*MySymbol* () : Clear the previous state.

The interface between the Open TURN'S platform and an external code is done at the *ComputedMathFunctionNumericalImplementation* level as shown within the next figure:


Computed Numerical Math Function Implementation
Wrapper Object
Numerical Math Function
Numerical Math Function Implementation

2.2 example: poutre

This example is located in the distribution at: *../OpenTURNS/test*

2.2.1 The XML FILE DESCRIPTION: poutre.xml

```xml
<wrapper>
  <library>

    <!-- The path of the shared object -->
    <path>testwrapper_3.so</path>

    <!-- This section describes all exchanges data between the wrapper and the
       platform -->
    <description>

      <!-- Those data are external to the platform (input files, etc.)-->
      <data>

  <!-- An input file -->
        <file id="file-1" type="in">
          <name>StdIn</name>
          <path>./fort.5</path>
        </file>

  <!-- Another input file -->
        <file id="file-2" type="in">
          <name>StdTruc</name>
          <path>/tmp/fort.55</path>
        </file>

  <!-- An output file -->
        <file id="file-3" type="out">
          <name>StdOut</name>
          <path>/tmp/out/fort.6</path>
        </file>

      </data>

      <!-- Those variables are substituted in the files above -->
      <!-- The order of variables is the order of the arguments of the function
        -->
      <variable-list>

        <!-- The definition of a variable -->
        <variable id="F" type="in">
          <comment>May the force be with you</comment>
          <unit>Newton</unit>
          <regexp>F=.*</regexp>
          <format>F=%10.5g</format>
        </variable>
      <!-- The definition of a variable -->
      <variable id="L" type="in">
        <comment>How long are you ?</comment>
        <unit>Meter</unit>
        <regexp>L=[0-9]+\.[0-9]* *m</regexp>
        <format>L=%8.4f</format>
      </variable>
    </variable-list>

    <!-- The function that we try to execute through the wrapper -->
    <function>
      <symbol>fonction1</symbol>
    </function>

    <!-- The gradient of the function is defined into the wrapper -->
    <gradient>
      <symbol>gradient1</symbol>
    </gradient>

    <!-- But the hessian is NOT defined, so it does NOT appear in this file --
       >

  </description>

  </library>
</wrapper>
```

2.2.2 The DYNAMIC LIBRARY: poutre.cxx

Firstly, you must define the size of uncertain input and output variables for the study:

```
/* Wrapper information */
  static struct WrapperInformation info_compute_deviation = {/* inSize_  = */ 4,
       /* outSize_ = */ 1};
```

WARNING:
If you don't read this information in xml file, you must have coherency between these two informations

```


    enum WrapperErrorCode func_exec_compute_deviation(void * p_state, const
        struct point * inPoint, struct point * outPoint)
  {
    internalStateIncrement(static_cast<struct WrapperInternals::internalState
        *>(p_state));

#if DEBUG_3
    {
      std::cerr << "<<< in void func_exec_compute_deviation(void * p_state=" <<
         p_state << ", struct point {size=" << inPoint->size_ << ", data=[";
      const char * separator = "";
      for(unsigned int i=0; i < inPoint->size_; i++, separator=",") std::cerr <<
          separator << inPoint->data_[i];
      std::cerr << "]}, ";
    }
#endif
```

```

    /* Transformation of Data OPEN TURNS => External Code */
    double E, F, L, I;

    if (inPoint->size_ == info_compute_deviation.inSize_) {
      E = inPoint->data_[0];
      F = inPoint->data_[1];
      L = inPoint->data_[2];
      I = inPoint->data_[3];
    } else return WRAPPER_WRONG_ARGUMENT;
  /* Be careful, It's here you assume the coherency between the definition
  of uncertain vector and parameters for external code*/
```

```
  /* In this trivial case, External code is in the same function */
    double d = -( F * L*L*L ) / ( 6 * E * I );
```

```
    /* Inverse ransformation of Data External Code => OPEN TURNS  */
  if (outPoint->size_ == info_compute_deviation.outSize_) {
      outPoint->data_[0] = d;
    } else return WRAPPER_WRONG_ARGUMENT;
```

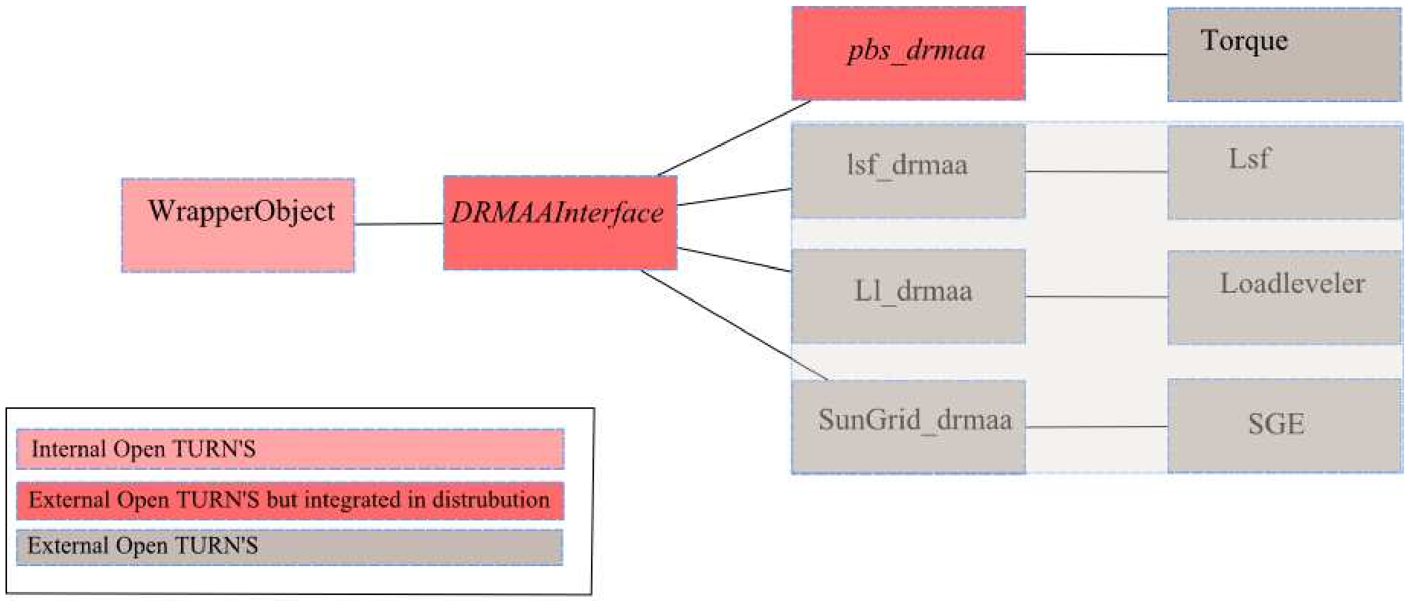
```


#if DEBUG_3
    {
      std::cerr << "struct point {size=" << outPoint->size_ << ", data=[";
      const char * separator = "";
      for(unsigned int i=0; i < outPoint->size_; i++, separator=",") std::cerr
         << separator << outPoint->data_[i];
      std::cerr << "]} ) NumberOfCalls = "
    << WrapperInternals::internalStateGetNumberOfCalls(static_cast<struct
       WrapperInternals::internalState *>(p_state))
    << " >>>" << std::endl;
    }
#endif

    return WRAPPER_OK;
  }
```

3 Distributed Context

3.1 Principle

Some recommandations are pre requisite for the use of a network of computers. Firstly, you must have a batch server in front of your grid, and today, the implementation in Open TURN'S is realised only with **PBS**.
The test is made with an open source implementation of PBS: Torque (`http://www.clusterresources.com/downloads/torque/`) version 2.1.7 and 2.2

The next figure shows the list of package useful for ditribution:



The extension to another batch server is not a problem, the missing package is only batchServer_drmaa. (see `http://drmaa.org/wiki/`)
You must have also a communication protocol between your local computer and your batch server. Actually, we test only with ssh.

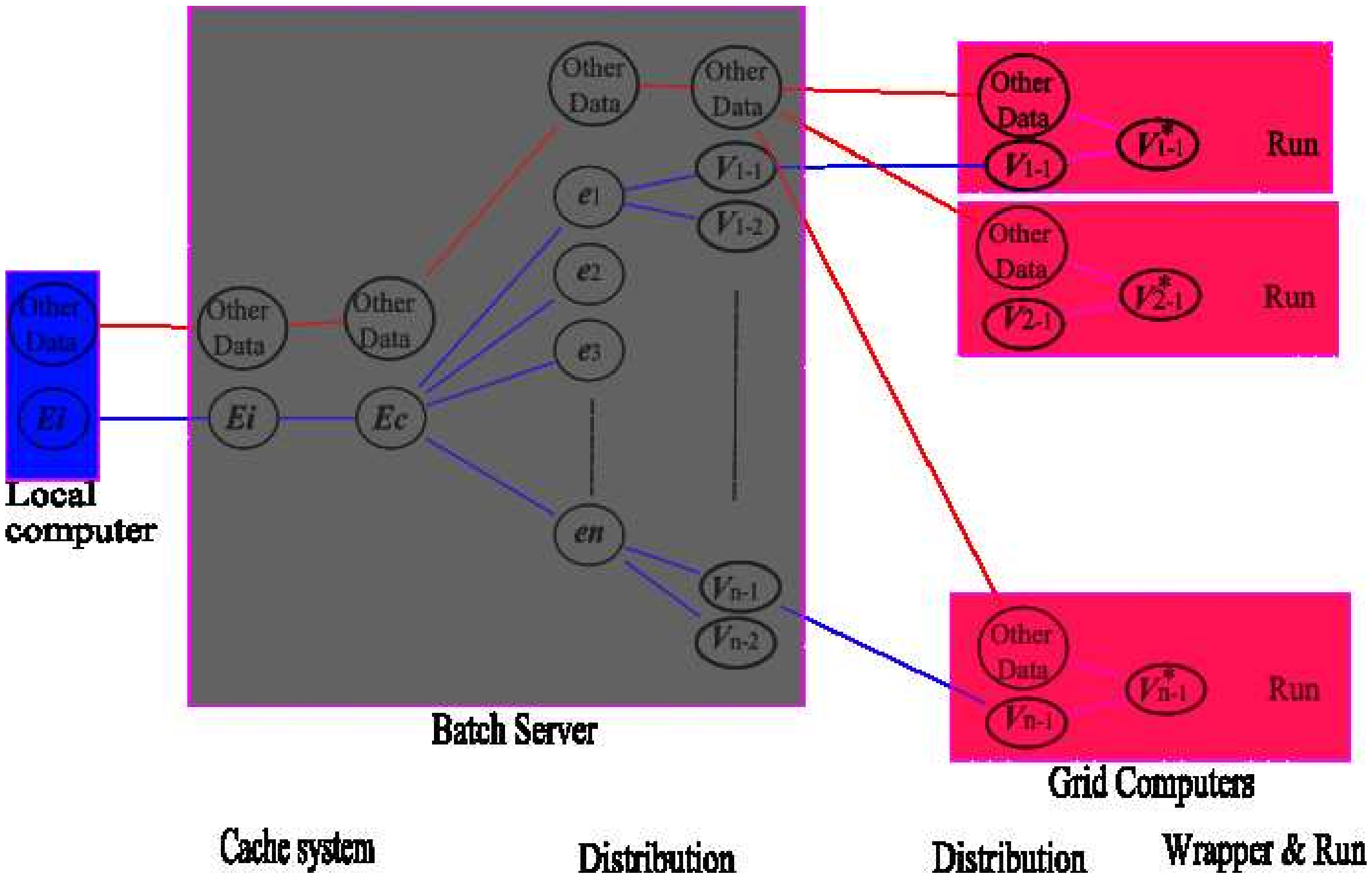
Be careful, Torque and DrmaaInterface need to have direct communications in both directions without password. With the ssh-keygen,you can resolve this constraint (see manpage ssh-keygen)

From the point of view of the tasks, the ideal workflow break up is thus:

1. Restrict the initial sample to a sample about which all parameter were never computed

2. Split the sample in sub samples with the optimal granularity (in point of view of algorithm)
3. Put on Server Batch all of generic data (called Other Data)
4. Split the sub samples on unitary uncertain vector and put it on Batch server
5. Execute the first data transformation for right execution of external code
6. Execute external code
7. Execute the last transformation of data for a right interpretation for Open TURN'S
8. Put this data on local computer

The figure below give an atoher view of this processus:



Today, the cache system (task 1) is not yet implemented in Open TURNS.

Two kinds of additional informations are mandatory to use the distribution, the first one, concerning the set of servers are defined in servers.xml, and the second, concerning the external code, are in the xml file describe int the chapter one (a new tag nammed distribution is defined)

3.2 examples

3.2.1 Servers Descritpion File:

In this file, you can define a set of possible servers available in your grid. For each server you must describe system's information:

```

  <server name="hal"> <!-- My favorite name -->
    <data>
      <protocol>scp</protocol><!-- Communication protocol -->
      <remote>/gridspace/</remote><!-- an existing remote directory where you
         want to push your data -->
      <host>hal</host><!-- the name define in hosts table -->
    </data>
  <!-- for dynamic using, you must define the Pathlib... Warning Today,
     pbs_drmaa is bugged for this function -->
    <exec gbatch="pbs">
       <drmaa>/home/ubuntu/pbs_drmaa-0.1.1/src/.libs/libdrmaa.so</drmaa>
       <lbatch>/home/ubuntu/torque-2.1.6/src/lib/Libpbs/.libs/libtorque.so</
          lbatch>
    </exec>
  </server>


<server>
```

3.2.2 Description File

In the file description, to use this function you must add the distribution field:

```

   <distribution>
  <computer id="hal"> <!-- the batch server you want to use for this study -->
          <executable>
             <name>/home/xxx/First </name><!-- the complete path for  first in
                your remote computer -->
             <arg> un </arg><!-- targument for the code, one argument per line
                (optionnal tag) -->
             <arg> deux </arg>
          </executable>
          <executable>
             <name>/home/xxxx/Second </name>
          </executable>
          <executable>
             <name>/home/xxxx/Third </name>
          </executable>
  </computer>
     </distribution>

<server>
```

Warning:
As you see it, the executable mut be accesible from your batch server (through NFS, physically present,...)
In the list of executables, you must describe your first converter, your executable and your last converter. Note that you can have a list of executables if you want.

3.3 How run an external code with a command line (MPI example)?

If you add the section Command, you cand launch your code with a commandline, and so you can use several processors for one execution.
For example, if you write:

```
                <executable>
                     <name>/home/xxxx/compute   </name>
                     <command> mpirun -np 2 </command>
                 </executable>
```

compute run on two processors if the soft is compiled with MPI.
Warning:
You must define the number of procesors in your command line (It's not necessarly the best place for this definition, but for the current version of OpenTURN'S, it is mandatory)

3.4 Package DrmaaInterface

3.4.1 list of API

1. **int ReadServerDescription (char * FileName, ServerInfo_ * pServerInfo)**
   Read the server description file, initialize the ServerInfo_ struct, return a non zero value if there is an error.

2. **int ReadFileDescription (char * FileName, RemoteExecutionInfo_ * pRemoteExecutionInfo)**
   Read the distribution field in the file describing the external code, initialize the RemoteExecutionInfo_ struct, return a non zero value if there is an error.

3. **int ConnectServer (DataServer_ DataServer)**
   Connect the batch server, return a non zero value if there is an error.

4. **int PutData (RemoteExecutionInfo_* RemoteExecutionInfo, DataServer_ ServerInfo, int size, JobStatus_ * jobS)**

Put in a unique directory, the necessary files to execute external code, return a non zero value if there is an error.

5. **int PutDataInTempFile (RemoteExecutionInfo_ * RemoteExecutionInfo, DataServer_ ServerInfo, const double * inPoint, int size, JobStatus_ * job)**
   Same as PutData, but put in remote directory a list of double through a temporary file, return a non zero value if there is an error.

6. **int AllocateJobTemplate (drmaa_job_template_t **jt)**
   Allocate a job template, return a non zero value if there is an error.

7. **int InitJobs (JobStatus_* jobS, RemoteExecutionInfo_ RemoteExecutionInfo , int size)**
   Initialise the job template, return a non zero value if there is an error.

8. **int CreateJobTemplate (drmaa_job_template_t *jt, RemoteExecutionInfo_ RemoteExecutionInfo, JobStatus_ job)**
   Create a job (drmaa_job_template_t) and initialise the JobStatus_ structure.

9. **int RunJob (drmaa_job_template_t *jt, JobStatus_ *job)**
   Launch the job,return a non zero value if there is an error.

10. **int StatusJob (JobStatus_ *job, RemoteExecutionInfo_ RemoteExecutionInfo)**
    Get the current status for the job, return a non zero value if there is an error.

11. **int InitSymbolDrmaa (DataServer_ DataServer)**
    If you want to use drmaa with a dynamic loader, you must use this function (today, pbs_drmaa habe a bug in a dynamic use)

3.4.2 exemple of use

```
#include "drmaaInterface.h"

enum WrapperErrorCode func_exec_sample_equation (void * p_state, const struct sample *
    inSample, struct sample * outSample)
{

  //Declaration
  //...
  //
  //Read File Description and Initialisation
  ReadFileDescription   ("equation.xml", &RemoteExecutionInfo);
  ReadServerDescription ("servers.xml" , &ServerInfo);

  //Choose the good Server
  for (int i=0; i < ServerInfo.nbServer; i++) {
    if ( strcmp(ServerInfo.Server[i].name, RemoteExecutionInfo.BatchServer) == 0 ) {
      DataServer = &(ServerInfo.Server[i]); break;
    }
  }

  //Put all Data (files defined in equations.xml) in remote directory
  ierr = PutData (&RemoteExecutionInfo, (*DataServer),size, (JobStatus_*) jobS);if (
      ierr != 0) return(WRAPPER_INITIALIZATION_ERROR);

  //Connecting to Batch Server
  ierr = ConnectServer(*DataServer); if (ierr != 0) return(WRAPPER_INITIALIZATION_ERROR
      );

  InitJobs (jobS, RemoteExecutionInfo, size);
  int GlobalComputation = isNotDone;
  int sizeComputed      = 0;
  int first             = 0;

  // Loop on Sample and submit batch
  while (GlobalComputation != isDone) {
    sizeComputed = 0;
    if (first != 0) sleep (_sleepDelay_);
    for (int i=0; i<size; i++) {

      if (first == 0) {
    inPoint  = &(inSample->data_ [i]);
    outPoint = &(outSample->data_[i]);
    //Put on remote computer the data (inpoint) through a temporary file
    ierr = PutDataInTempFile   (&RemoteExecutionInfo, (*DataServer), inPoint->data_,
                 inPoint->size_,  &jobS[i]              ); if (ierr != 0) return(
                   WRAPPER_INITIALIZATION_ERROR);
    //Create a Job template
    ierr = AllocateJobTemplate (&(jt[i]))                           ; if (ierr !=
        0) return(WRAPPER_INITIALIZATION_ERROR) ;
    ierr = CreateJobTemplate   (jt[i], RemoteExecutionInfo,  jobS[i]   ); if (ierr !=
        0) return(WRAPPER_INITIALIZATION_ERROR);
    //Launch job
    ierr = RunJob              (jt[i], &(jobS[i]))                  ; if (ierr !=
        0) return(WRAPPER_INITIALIZATION_ERROR);
    //Get Status of current job
    ierr = StatusJob           (&(jobS[i]), RemoteExecutionInfo)    ; if (ierr !=
        0) return(WRAPPER_INITIALIZATION_ERROR);

      }
      else {
    ierr = StatusJob           (&(jobS[i]), RemoteExecutionInfo)        ; if (ierr !=
         0) return(WRAPPER_INITIALIZATION_ERROR);
    if (jobS[i].status == isDone) sizeComputed ++;
    //launch a new job if the previous on the same computer is finished
    else  if (jobS[i].status == Next) {
      ierr = CreateJobTemplate   (jt[i], RemoteExecutionInfo,  jobS[i]   ); if (ierr !=
           0) return(WRAPPER_INITIALIZATION_ERROR);
      ierr = RunJob                    (jt[i], &(jobS[i]))                        ; if (ierr !=
         0) return(WRAPPER_INITIALIZATION_ERROR);
      }
        }
      }
      if (sizeComputed == size) GlobalComputation = isDone;
      first = 1;
   }
}
```

4 analyticalMathFunctionCompiler

This tool enables to create automatically the wrapper for analytical functions : $\mathbb{R}^n \rightarrow \mathbb{R}$.
Its execution is the following one :

analyticalMathFunctionCompiler <way> <name> <formula> <v1> <v2> ... <vn>

where :

- *way* : indicates the adress where the files created are put,
- *name* : indicates the name of the file.xml,
- *formula* : indicates the formula of the analytical function,
- *<v1> <v2> ... <vn>* : indicates the description of the input data (conatins at least the list of the input data of the function which intervene in the formula).

The tool creates the files **name.xml**, **name.cxx** and **libname.so** at the adress *way*.

All the suffixes *tata, titi, toto, tutu* are fixed to *name*.

There is no implementation for the gradient and the hessian of the function automatically generated.
For example :

analyticalMathFunctionCompiler NULL sum 'x+y' x y

The tool creates **sum.xml**, **sum.cxx** and **sum.so** where the tool command is executed.