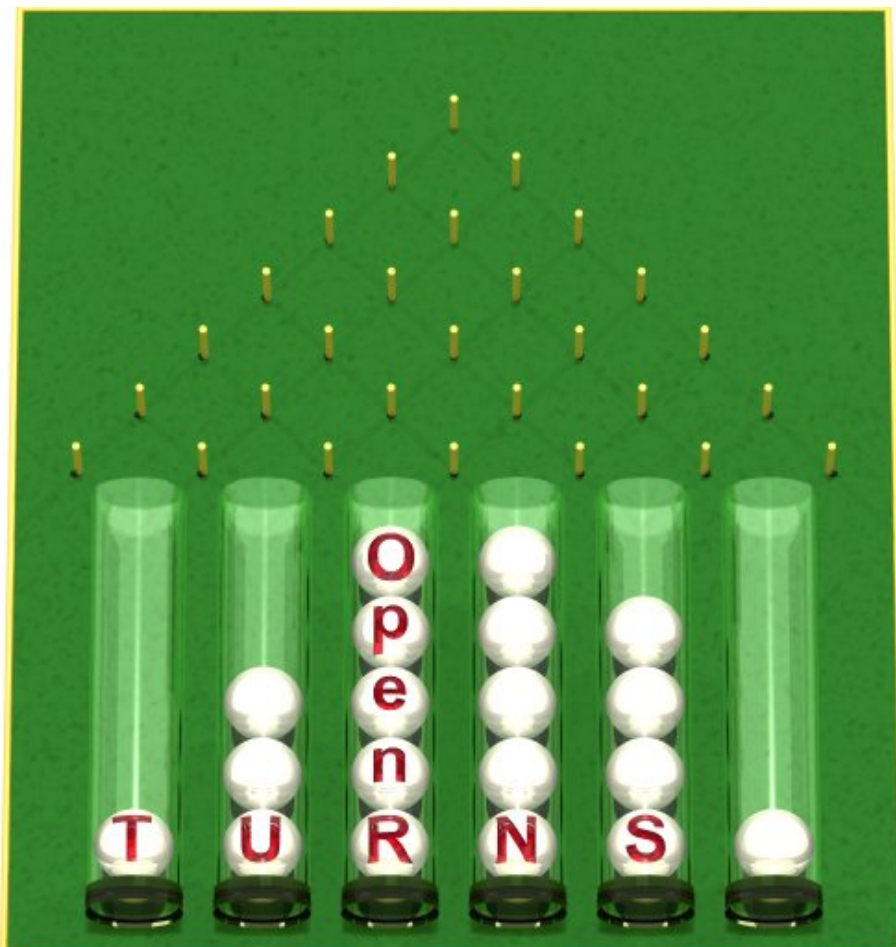


# Contribution Guide

Open TURNS version 1.0

Documentation built from package openturns-doc-April2012

April 20, 2012



## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>How to add a class MyClass in an existing directory of OpenTURNS sources?</b>	<b>2</b>
2.1	First, add the class to the OpenTURNS C++ library . . . . .	2
2.2	Second, document your contribution (in english, using LaTeX) . . . . .	3
2.3	Third: make your contribution usable from the Textual User Interface . . . . .	4
2.4	Almost finished. Document your contribution to the TUI . . . . .	4
<b>3</b>	<b>How to add a whole set of classes in a new subdirectory of OpenTURNS sources?</b>	<b>5</b>
3.1	Autotool infrastructure in the root directory . . . . .	5
3.2	Autotool infrastructure in the parent subdirectory . . . . .	5
3.3	Autotool infrastructure in the new subdirectory . . . . .	6
<b>4</b>	<b>How to develop a new extra module MyModule</b>	<b>7</b>
4.1	Copy and adapt an existing template . . . . .	7
4.2	Develop the module . . . . .	8
4.3	Install and test . . . . .	8
<b>5</b>	<b>How to use version control system</b>	<b>9</b>
5.1	File hierarchy . . . . .	9
5.1.1	Description . . . . .	9
5.1.2	An example . . . . .	10
5.2	Merging process . . . . .	11
5.3	Tagging and releasing process . . . . .	12
5.3.1	Tagging . . . . .	13
5.3.2	Releasing . . . . .	13

## 1 Introduction

This documentation aims at guiding the developers in their contributions to the OpenTURNS software. This contribution can be done in (at least) the two following contexts:

- a contribution to the C++ library,
- a contribution to the TUI written in python,

## 2 How to add a class `MyClass` in an existing directory of OpenTURNS sources?

This how-to explains the process that must be followed to fully integrate a new class that provides an end-user facility (e.g. a new distribution). We suppose that this class will take place in an existing directory of the sources directories, to avoid the burden of the autotools infrastructure creation.

### 2.1 First, add the class to the OpenTURNS C++ library

1. Create `MyClass.hxx` and `MyClass.cxx` in the same directory. The files must have the standard header comment, with a brief description of the class in Doxygen form and the standard reference to the LGPL license.

For the header file `MyClass.hxx`, the interface must be embraced between the preprocessing clauses:

```
#ifndef OPENTURNS_MYCLASS_HXX
#define OPENTURNS_MYCLASS_HXX
...
your interface
...
#endif OPENTURNS_MYCLASS_HXX
```

to prevent from multiple inclusions.

See any pair of `.hxx/.cxx` files in the current directory and the PGQL document for the OpenTURNS coding rules: case convention for the static methods, the methods and the attributes, trailing underscore for the attribute names for naming a few.

2. Modify the `Makefile.am` file in the directory containing `MyClass.hxx` and `MyClass.cxx`:
  - add `MyClass.hxx` to the `otinclude_HEADERS` variable
  - add `MyClass.cxx` to the `libOTXXXXXX_la_SOURCES` variable, where `XXXXXX` is the name of the current directory.
3. Modify the `CMakeLists.txt` file in the directory containing `MyClass.hxx` and `MyClass.cxx`:
  - add `MyClass.hxx` to the headers using `ot_install_header_file ( MyClass.hxx )`.
  - add `MyClass.cxx` to the sources using `ot_add_source_file ( MyClass.cxx )`.
4. Add `MyClass.hxx` to the file `OTXXXXXX.hxx`, where `XXXXXX` is the name of the current directory.
5. Create a test file `t_MyClass_std.cxx` in the directory `lib/test`. This test file must use the standard functionalities of the class `MyClass`.

6. Create an autotest file `t_MyClass_std.at` in the directory `lib/test`. This file describes the test and how to run it.
7. Create an expected output file `t_MyClass_std.expout` that contains a verbatim copy of the expected output (copy-paste the *validated* output of the test in this file).
8. Modify the `CMakeLists.txt` file in `lib/test`: add `ot_check_test ( MyClass_std )` in this file.
9. Modify the `Makefile.am` file in `lib/test`:
  - add `t_MyClass_std` (which is the name of the test executable) to the variable `CHECK_PROGS` or `INSTALLCHECK_PROGS` depending on the fact the test checks the correct behaviour of OpenTURNS independently of its installation or not. The several executables are organized following the library organization, you must follow this rule.
  - add `t_MyClass_std.at` to the variable `CHECK_TESTS` or `INSTALLCHECK_TESTS` and in the correct set of autotest files, following the same rules than for the executable.
  - add `t_MyClass_std.expout` to the variable `OUTFILES` in the relevant paragraph.
  - Create a variable called `t_MyClass_std_SOURCES` and set its value to `t_MyClass.cxx` in the relevant set of sources.
10. Modify the `Makefile.am` file in `lib/m4/examples`:
  - add `t_MyClass_std` to the variable `bin_PROGRAMS`, which is the list of program examples.
  - create a variable `t_MyClass_std_SOURCES` and set its value to `t_MyClass.cxx` in the relevant set of sources.
11. Add `t_MyClass_std.at` to the file `check_testsuite.at` or `installcheck_testsuite.at` using the same rule than for the `Makefile.am` modification.
12. If the validation of your class involved advanced mathematics, or was a significant work using other tools, you can add this validation in the `validation/src` directory.
  - copy all of your files in the `validation/src` directory.
  - modify the `Makefile.am` file by appending the list of your files to the `dist_validation_DATA` variable.

That's it! Your class is integrated to the library and will be checked for non-regression in all the subsequent versions of OpenTURNS, assuming that your contribution has been incorporated in the "official" OpenTURNS release. But nobody can use it!

## 2.2 Second, document your contribution (in english, using LaTeX)

13. Add an entry in the document `doc/src/ArchitectureGuide/OpenTURNS_ArchitectureGuide.tex` if your class has a significant impact on the library architecture.
14. Add an entry in the document `doc/src/WrappersGuide/OpenTURNS_WrappersGuide.tex` if your class has a significant impact on the way OpenTURNS interfaces external codes.
15. Add an entry in the document `doc/src/ReferenceGuide/OpenTURNS_ReferenceGuide.tex` if your class add a new concept not already described in the reference guide. Your entry must take the form of a specific description using the same template than the other descriptions.

Ok, your contribution can be used by a programmer who uses the library. But for the other users, some work remains.

### 2.3 Third: make your contribution usable from the Textual User Interface

16. Create MyClass.i in the python/src directory. In most situations, it should be:

```
// SWIG file MyClass.i
// Author: $LastChangedBy: schueller $
// Date: $LastChangedDate: 2012-02-27 14:48:06 +0100 (lun. 27 fÃ©vr. 2012) $
// Id: $Id: OpenTURNS_ContributionGuide.tex 1539 2012-02-27 13:48:06Z schueller $

% {
#include "MyClass.hxx"
%}

#include MyClass.hxx
namespace OT { %extend MyClass { MyClass(const MyClass & other)
  { return new OT::MyClass(other); } } }
```

17. Modify the CMakeLists.txt file in python/src: add MyClass.i to the relevant ot\_add\_python\_module clause.
18. Modify the Makefile.am file in python/src: add MyClass.i to the variable OPENTURNS\_SWIG\_SRC
19. Locate and modify the file yyyy.i, where yyyy is the name of the python module related to MyClass, to include MyClass.i in the correct set of .i files (see the comments in yyyy.i file). In order to identify the correct python module, remember that the modules map quite closely the source tree organization.
20. Create a test file t\_MyClass\_std.py in the directory python/test. This test implements the same tests than t\_MyClass\_std.cxx, but using python.
21. Create an autotest file t\_MyClass\_std.atpy and the associated t\_MyClass\_std.expout file that have the same role than t\_MyClass\_std.at and t\_MyClass\_std.expout, but for the python test.
22. Modify the Makefile.am file in python/test:
- add t\_MyClass\_std.py to the variable PYTHONINSTALLCHECK\_PROGS. The several executables are organized following the library organization, you must follow this rule.
  - add t\_MyClass\_std.atpy to the variable PYTHONINSTALLCHECK\_TESTS.
  - add t\_MyClass\_std.expout to the variable OUTFILES.

### 2.4 Almost finished. Document your contribution to the TUI

1. Comment your python test as a new use-case in the document doc/src/OpenTURNS\_UseCasesGuide/UseCasesGuide.tex following the generic format of this document:
  - describe the inputs of your use-case.
  - extract code snippets that show the user interaction with your class.
  - add the relevant keywords to the index.
2. Gives a description of your class in the document doc/src/UserManual/OpenTURNS\_UserManual.tex
  - following the general form of this document, fill-in the sections but only describe the methods the user is intended to use (forget the most computer programming inclined methods).

- don't hesitate to give some reminders of theoretical aspects if needed, in the form of an equation or a short (1 or 2 sentences) mathematical explanation. Give a pointer to the relevant reference guide section.

That's all, folks!

Some timings from an OpenTURNS Guru: 2 days of work for the most trivial contribution (a copy-paste of a class with 5 methods, no mathematical or algorithmic tricks). For a well-trained OpenTURNS contributor, a user-visible class with a dozen of methods and well-understood algorithms, a new class should not be less than a week of work...

### 3 How to add a whole set of classes in a new subdirectory of OpenTURNS sources?

This how-to explains the process that must be followed to fully integrate a set of classes that provides an end-user facility (e.g. a new simulation algorithm) developed in a new subdirectory of the existing sources. The task is very similar to the steps described in the how-to (2), only the new steps will be described. We suppose that the subdirectory has already been created, as well as the several source files. There are three new steps in addition to those of the how-to (2): the creation of the autotool infrastructure in the new subdirectory, the modification of the autotool infrastructure in the parent directory and the modification of the autotool infrastructure in the root directory.

#### 3.1 Autotool infrastructure in the root directory

You have to tell the configure script to create the Makefile in the new directory and to add this new directory into the list of directories where the preprocessor will look for header files. Let's assume that you want to add a subdirectory named NewDir to a directory which path is lib/Base/Directory/:

1. add the name of the Makefile to be created to the AC\_OUTPUT target:

```
AC_OUTPUT([...
          src/Base/Directory/NewDir
          ...
```

You have to put your new directory at the right place: all the directories are sorted in lexical order and in an increasing depth order.

2. add the path of the new directory to the openturns\_cppflags variable. The path must be relative to the top\_srcdir directory:

```
openturns_cppflags=
for flag in "
...
-I\$(top_srcdir)/../lib/Base/Directory/NewDir
...
```

#### 3.2 Autotool infrastructure in the parent subdirectory

You have to set up the recursive call of Makefiles from a parent directory to its subdirectories, and to aggregate the libraries related to the subdirectories into the library associated to the parent directory:

- add NewDir to the DIRS variable:

```
DIRS += NewDir
```

if NewDir is the first directory in the variable DIRS, replace += by =.

- aggragate the library associated to NewDir with the library associated to Directory:

```
libOTDirectory_la_LIBADD += $(builddir)/NewDir/libOTNewDir.la
```

the library names are prefixed by libOT. If libOTNewDir is the first library added to libOTDirectory, replace += by =.

### 3.3 Autotool infrastructure in the new subdirectory

You have to create the Makefile.am file. Its general structure is given by the following template:

```
#                                     -*- Makefile -*-
#
# Makefile.am
#
# (C) Copyright 2005-2009 EDF-EADS-Phimeca
#
# This library is free software; you can redistribute it and/or
# modify it under the terms of the GNU Lesser General Public
# License as published by the Free Software Foundation; either
# version 2.1 of the License.
#
# This library is distributed in the hope that it will be useful
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public
# License along with this library; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
#
# @author: $LastChangedBy: schueller $
# @date:   $LastChangedDate: 2012-02-27 14:48:06 +0100 (lun. 27 fÃ©vr. 2012) $
# Id:     $Id: OpenTURNS_ContributionGuide.tex 1539 2012-02-27 13:48:06Z schueller $
#
include $(top_srcdir)/config/common.am

SUBDIRS = FirstDir
SUBDIRS += SecondDir
...
SUBDIRS += LastDir

AM_CPPFLAGS += -I$(srcdir)/Relative/Path/To/First/Include/Dir
```

```

...
AM_CPPFLAGS += -I$(srcdir)/Relative/Path/To/Last/Include/Dir

otincludedir = $(includedir)/openturns
otinclude_HEADERS = \
FirstFile.hxx \
    ...
LastFile.hxx

noinst_LTLIBRARIES = libOTNewDir.la
libOTNewDir_la_LDFLAGS = -no-undefined
libOTNewDir_la_SOURCES = \
FirstFile.cxx \
    ...
LastFile.cxx

libOTNewDir_la_LIBADD = $(builddir)/FirstDir/libOTFirstDir.la
libOTNewDir_la_LIBADD += $(builddir)/SecondDir/libOTSecondDir.la
...
libOTNewDir_la_LIBADD += $(builddir)/LastDir/libOTLastDir.la

```

Be aware of the fact that you *must* use tabulations to indent the list of names in both `otinclude_HEADERS` and `libOTNewDir_la_SOURCES` variables.

The meaning of the different variables is recalled here:

6. `SUBDIRS` is a variable local to the Makefile.am file: it gathers the list of subdirectories of the current directory.
7. `AM_CPPFLAGS` is a variable that gathers the list of directories in which the preprocessor will look for header files needed by the source files in `NewDir`.
8. `libOTNewDir.la` is the library associated to the current directory, namely `NewDir`.
9. `otinclude_HEADERS` is the list of header files present in the current directory, namely `NewDir`.
10. `libOTNewDir_la_LDFLAGS` is the list of flags passed to the linker when creating `libOTNewDir`.
11. `libOTNewDir_la_SOURCES` is the list of source files present in the current directory, namely `NewDir`.
12. `libOTNewDir_la_LIBADD` is the list of libraries to aggregate in `libOTNewDir.la`. There should be exactly one library by subdirectory.

## 4 How to develop a new extra module MyModule

### 4.1 Copy and adapt an existing template

1. Copy and rename the source tree of an example module (for example the Strange module) from the OpenTURNS source tree. The examples modules are located under the module subdirectory of OpenTURNS source tree:

```
svn export https://svn.openturns.org/openturns-modules/template MyModule
```

2. Adapt the template to your module:

```
./customize MyModule MyClass
```

This command changes the module name into MyModule in all the scripts, and adapt the example class to the new name MyClass.

## 4.2 Develop the module

3. Implement your module using the same rules as described in the sections 2 and 3.
4. Build your module as usual:

```
./bootstrap
mkdir build
cd build
../configure --with-openturns=OPENTURNS_INSTALLDIR --prefix=$PWD/install
make
make check
make install
make installcheck
```

## 4.3 Install and test

1. Check that you have a working OpenTURNS installation, for example by trying to load the OpenTURNS module within an interactive python session:

```
python
>>> from openturns import *
```

and python should not complain about a non existing openturns module.

2. Create a source package of your module:

```
make dist
```

It will create a tarball named mymodule-X.Y.Z.tar.gz (and mymodule-X.Y.Z.tar.bz2), where X.Y.Z is the version number of the module.

3. In the python directory of the OpenTURNS install directory, you can find a script named *openturns-module*. You use this script to install the tarball mymodule.tar.gz (or mymodule.tar.bz2) in your home directory (*\$HOME/openturns*):

```
openturns-module --install=mymodule-X.Y.Z.tar.gz --prefix=user
```

The installation script has many more capabilities, you can access to its embedded documentation by invoking it without argument:

```
openturns-module
Usage: openturns-module [--silent] [--install=<module> | --remove=<module>} [--prefix=PFX] [extra_configure_args]
      openturns-module [--silent] [--install <module> | --remove <module>} [--prefix PFX] [extra_configure_args]
      openturns-module [--silent] [--module=<module> | --module <module>} [options]
```

Note:

For installation, 'module' can be either a path to a directory or a path to a archive file (compressed or not).  
For removal, 'module' is the module name.  
The extra configure args are passed as is to the module configure script.

Example:

```
(install)
openturns-module --install=mymodule/
openturns-module --install=/path/to/mymodule/
openturns-module --install=mymodule.tar
openturns-module --install=mymodule.tar.gz
openturns-module --install=mymodule.tgz
openturns-module --install=mymodule.tar.bz2
openturns-module --install=mymodule.tbz
```

You can provide a prefix to choose where the module will be installed.

PFX can take one of the following values:

- \* openturns : the module will be installed in the Open TURNS installation tree
- \* user : the module will be installed in the Open TURNS user directory (/home/regis/openturns)
- \* <dir> : the module will be installed in <dir>. You should append <dir> to the OPENTURNS\_MODULE\_PATH envvar to tell Open TURNS where to find the module

(remove)

```
openturns-module --remove=mymodule
```

(mixed form)

```
openturns-module --remove=myoldmodule1 --install=mynewmodule1 --install=mymodule2
```

(module)

```
openturns-module --module=mymodule
```

Options:

```
--silent          Do not output any message
```

#### 4. Test your module within python:

```
python
```

```
>>> from openturns import *
>>> from mymodule import *
```

and python should not complain about a non existing mymodule module.

That's all folks!

## 5 How to use version control system

This section describes how to use the Open TURNS version control system AKA subversion repository. These instructions may be useful for Open TURNS developers which have an account on subversion.

### 5.1 File hierarchy

#### 5.1.1 Description

As usual with subversion repository, root directory contains three directories described bellow :

- trunk

- tags
- branches

Each user have a personal branch named simply by username. Every branch must be follow some rules to be more easy to merge with others developments. These rules are really simple:

- At the first time, the developer must create one devel directory by copying the trunk using a conventional commit message (see below);
- if a developer wants to make some support on one or more major version, he must copy the corresponding tags, this time using a special commit message;
- IT IS STRICTLY FORBIDDEN for a developer to apply a patch manually across subversion directories: he must ALWAYS use the subverion merging facility to do that;
- IT IS STRICTLY FORBIDDEN for a developer to merge a branch with another one: all the merges must be done from or to the trunk.

For example, with one developer "John Doh", the file hierarchy looks like this:

- trunk
- tags
  - openturns-0.12.1
  - openturns-0.12.2
  - openturns-0.12.3
  - openturns-0.13.0
- branches
  - doh

### 5.1.2 An example

It is simpler to explain how each directory is used through an example. Suppose that a new developer, John Doh, joins the OpenTURNS development team: upcoming developer in OpenTURNS repository :

- On the server side :
  1. Subversion's administrator creates a new account named "doh" for the user John Doh;
  2. Subversion's administrator creates a new subdirectory /doh in /branches, leading to /branches/doh: it is the 'private' user branch;

Now, John Doh can start its work in the repository. The first action for the developer is to "branch" the trunk, which is the reference for the project: the trunk is the place for the latest working development version.

- On the client side :
  1. John Doh can now "checkout" its private branch into a local copy, i.e. on its local computer, not on the server.

```
svn checkout https://svn.openturns.org/openturns/branches/doh doh-branch
```

2. To be able to develop with the latest version, John copies the trunk's content into his local copy. You may have multiple contiguous directories in your branch to split your developments according to their targets. Here we suppose that the developments will be merged in the trunk, for example to add a new functionality.

```
svn copy https://svn.openturns.org/trunk doh-branch
```

3. The previous step has made a local copy of the trunk into the local copy of its branch. Now, John MUST make his first commit :

```
svn commit -m "BRANCH: svn copy https://.../trunk@1170 branch"
```

To ease the management of the version control system, we enforce the systematic use of comments for each commit. When the commit is related to a branch modification such as a copy, a merge, a delete, a tag or anything similar, the comment MUST start by the corresponding keyword in uppercase. These keywords are:

- BRANCH for a branch creation or deletion
- MERGE for a merge between a branche and the trunk (in either direction)
- TAG for a tag

Also, when you create a branch you have to specify the associated trunk revision number (here, r1170). You can find it at <http://trac.openturns.org/browser> It is the number in the "rev" column.

4. Each step (ie functionality, bug fix, etc.) must be committed separately with a comment. e.g. if John Doh write an enhanced cache system he writes:

```
svn commit -m "Introduced an enhanced cache system."
```

The -m option is useful for single line comments, ie for very small comments. If your commit needs to be explained in a few lines, it is better to log the entire comments in a plain text file (e.g., svn.log is a quite common name) and to use the -F option followed with the file name.

```
edit svn.log
svn commit -F svn.log
```

## 5.2 Merging process

This section describes the several steps of the merging process, in particular the preparation work that must be done on the developer side before the integrator can merge its developments.

The whole process must succeed in order to be able to merge. If any failure occurs, it MUST be fixed, otherwise the subversion repository would be corrupted. Suppose that John Doh wants to merge its branch with the latest revision of the trunk, nammely the revision 620 (r620). He must:

1. Checkout the branch to merge (ie: for John Doh)

```
svn co https://svn.openturns.org/branches/doh/devel doh-devel
```

2. Find the latest synchronization revision of this branch with trunk (e.g. r365). he finds this information by looking at the latest MERGE or BRANCH keyword in the log of his branch

```
svn log | egrep -B6 -e 'BRANCH|MERGE'
```

Here we suppose that the latest synchronization was done at the revision 365 (r365).

3. Apply the differences between the latest synchronization (r365) and the latest revision of trunk (r620) to the local copy of the branch

```
cd doh-devel
svn merge -r365:620 https://svn.openturns.org/openturns/trunk
```

4. Verify patched files and conflicts

```
svn status | egrep -e '^C'
svn status | egrep -v -e '^[DGURAM]'
```

These commands should not return any error or missing files, i.e. the output should be empty.

5. Verify that the compilation and the archive creation process works fine with this updated version of the branch

```
./bootstrap
mkdir build && cd build
../configure --prefix=\$PWD/install
make
make distcheck DISTCHECK_CONFIGURE_FLAGS='--without-swig'
cd -
```

In this process, it is required to build and test the developments in a directory SEPARATED from the source directory. So DO make a build subdirectory and DO change do it. It does not take time but it ensures that the building process is correct (you can't check for this when building in the same directory than the sources).

6. If there is no error, commit this revision with a special log message

```
svn commit -m 'MERGE: svn merge -r365:620 https://.../trunk'
```

When committing, don't forget to add the correct keyword (MERGE, BRANCH, TAG): it will be crucial for the next merge. Otherwise it is very painful to find the latest synchronization point.

### 5.3 Tagging and releasing process

This section explains how the integrator must tag and release a new version with this version control system. Add this step, we suppose that the merging process has been done. The trunk is clean and is ready to be released as an official version.

### 5.3.1 Tagging

1. The first step, before tagging, is to update all the ChangeLog files with a very pretty tool : 'svn2cl'

```
for i in `find ./ -name 'ChangeLog'`; do
  DIR=`dirname $i`;
  cd $DIR; svn2cl -i;
  cd -;
done
```

2. Edit the 'configure.ac' files to set the new release version

```
for i in `find ./ -name 'configure.ac'`; do
  sed -e "s:\[0.13.0\]:[0.13.1]:" $i > $i.sed && mv $i.sed $i;
done
```

3. You can now commit these modifications

```
svn ci -m 'Next release (0.13.1) preparation.'
```

4. Finally, you can now tag the new version with the following commands

```
cd ..
svn copy trunk tags/openturns-0.13.1
svn ci -m 'TAG: This is official release 0.13.1.'
```

### 5.3.2 Releasing

1. Export the newly created tag in a temporary directory

```
cd /tmp
svn export https://svn.openturns.org/tags/openturns-0.13.1
```

2. Bootstrap, configure and make your tagged version

```
cd openturns-0.13.1
./bootstrap
./configure
make
```

3. Build the tarball

```
make distcheck
cd ..
ls *.tar.*
```