TECHNICAL REPORT
IGE–374

DEVELOPMENT PROCEDURES FOR VERSION5 OF REACTOR PHYSICS CODES

A. Hébert

Institut de génie nucléaire
Département de génie mécanique
École Polytechnique de Montréal
August 11, 2020

SUMMARY

Version 5 is a new distribution of reactor physics codes developed at the Groupe d'Analyse Nucléaire (GAN) of École Polytechnique de Montréal. This distribution is developed using modern software engineering techniques that are likely to improve its quality and help the developers in their daily work. Three aspects of development procedures are described in this report:

- version control of the project components
- issue tracking and spiral development management
- configuration management of the codes DRAGON, TRIVAC and DONJON.

We will discuss the implementation of the development procedures and their use by Version5's developers. These procedures are often assimilated to a *quality assurance* (QA) plan (*plan d'assurance qualité particulier* in french), although they are conceived to facilitate the evolution of Version5 rather than to slow down its development.

1 Version5 basics

One of the main goal of Version5 is to encapsulate all reactor physics codes developed at GAN in a unique and consistent software development project, and to maintain its consistency and quality during its development. We want to avoid any duplication of code among software components and maintain consistency in LCM object specifications and module specifications. In this system, DRAGON is consider as a foundation code providing functionalities to the other codes in the system. Any LCM object or module available in DRAGON and required by another code is simply accessed from DRAGON library and is not duplicated as previously done in Version3.

Version5 is divided into software components, some of them producing only libraries and some producing both libraries and executables. Version5 development is performed under the GNU Lesser General Public License (LGPL).[1] The components are

UTILIB Set of numerical analysis Fortran subroutines compiled as a library.

GANLIB Set of C and Fortran subroutines implementing the computer science layers in Version5, including memory management, LCM access routines, CLE-2000 macro-language[2] and utility modules. Both library and executable are produced.

TRIVAC Full-core finite element code (static and space-time kinetics).[3] Both library and executable are produced.

DRAGON Lattice code.[4, 5] DRAGON produces a *multi-parameter reactor database* that can be used as input in full-core calculations. Both library and executable are produced.

DONJON Full-core simulation of reactor operation.[6] Both library and executable are produced.

2 Version control of the project components

Version control is the art of managing changes to information. Although mainly used for software development, its use can be extended to manage the production of any textual document involving many contributors. A version control tool make possible the collaborative work of many developers on the same project by implementing a systematic way of making modifications. Each modification bring to the project is recorded and can be removed if required. Tools are available to manage the tedious situation where many developers are working on the same project component.

Many tools are available to perform version control. We have chosen Subversion, an open-source version control system that is free, powerful, well-accepted by computer scientists and widely available.[7] Subversion is a widely-used system used to keep track of the historical evolution of software, procedures, scripts non-regression tests and related documentation. We propose to use Subversion for the totality of Version5 components. Subversion can be used from command line in a UNIX shell or from a graphical user interface such as WorkBench.[8]

The basic principle behind version control is to keep the project information in a *version control database* or *repository* and to record every operation performed on this repository. The repository hierarchical structure is depicted in Fig. 1. The repository represents the *eternity* of the project. The information in the repository is organized with a directory structure, as shown in Fig. 1. Each directory contains a hidden sub-directory named `.svn` with version control information.

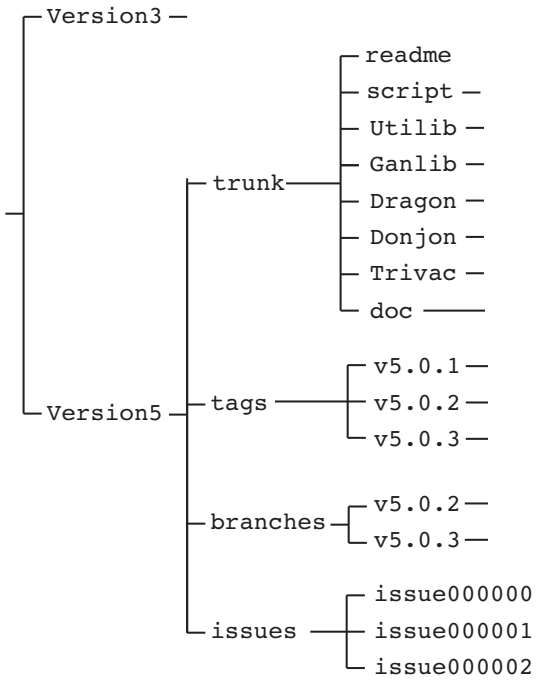

Figure 1: Directory layout in the repository

2.1 Initial commit of the project

Some operations are required to start a project and to build the initial Subversion repository on a server. These operations are done once so that most project developers never have to perform them.

In version control terminology, a *commit* (aka *check-in*) is an operation where a developer input some information in the repository and recover a corresponding *revision identifier*. Note that a commit operation can never be undo, the information written in the repository is written forever. However, it is always possible to perform a subsequent commit operation to annihilate the effect of a previous commit, without erasing any information. In a version control system, it is always possible to recover the state of a system at *any* revision identifier.

The first step consists to create the repository on the server. This operator is done by the system administrator in charge of your network. The operations are following:

```
svnadmin create --fs-type fsfs /usr/local/Version5_beta
chgrp -R admin /usr/local/Version5_beta/
chmod -R g+rwx /usr/local/Version5_beta/
chmod -R o+rx /usr/local/Version5_beta/
```

where `admin` is the name of the UNIX group allowed to modify the repository. The system administrator can also start the remote server for the repository:

```
svnserve -d -r /usr/local/Version5_beta/ &
ps aux | grep svn
```

All these operations must be performed as `root`.

Next, the initial commit of the project can be performed by a developer of the project. An image of the project layout, named `trunk` and similar to Fig. 2, is created and is used to fill the repository. After creation of the repository, the image `trunk` is destroyed.

```
svn mkdir file:///usr/local/Version5_beta -m 'create Version5 directory'
svn mkdir file:///usr/local/Version5_beta/issues -m 'create issues directory'
cp trunk/script/*-commit /usr/local/Version5_beta/hooks/
svn import trunk file:///usr/local/Version5_beta/trunk \
       -m 'issue000000: Initial import of Version5'
rm -r -f trunk
```

Note the third line used to install the pre-commit and post-commit scripts in the repository. The use of these scripts will become clear in Section 3.

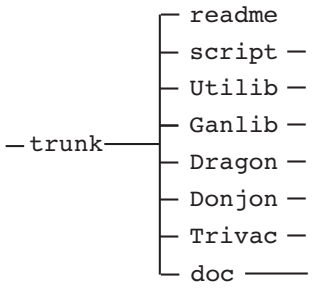

Figure 2: trunk layout

A *working copy* of the project is next created (check-out) using

```
svn checkout file:///usr/local/Version5_beta/trunk/ Version5_wc
```

Here, the working copy is named `Version5_wc`, although another name could be used. It is created without the `trunk` intermediate directory. The working copy is also different from the initial image as it contains the hidden directories `.svn`.

There is a subroutine in `Ganlib` named `KDRVER.f` that is used to print the *tag identifier* of each frozen version of the project (such as `v5.0.1`). This subroutine automatically recover the corresponding Subversion *revision identifier*, provided that *keyword expansion* is enabled for this subroutine. This is done with the following two lines:

```
svn propset svn:keywords "Date Revision" Version5_wc/Ganlib/src/KDRVER.f
svn commit -m 'issue000000: Put keyword expansion' Version5_wc
```

Finally, the issue tracking information is recovered (check-out) using

```
svn checkout file:///usr/local/Version5_beta/issues/ issues_wc
```

This information is recovered as a directory named `issues_wc` containing a set of card-indexs, each of them representing a development issue. Their use will become clear in Section 3. At this point, only the card-index relative to the activation of keyword expansion (named `issue000000`) exists.

2.2 Daily operations on the repository

During the life of the project, selected developers are allowed to perform read and/or write operations on the repository. We will cover the more frequent cases. More information can be found in Ref. 7.

A developer recover the most recent development version of the project and create its own *working copy* using

```
svn checkout file:///usr/local/Version5_beta/trunk/ Version5_wc
```

A developer recover the frozen version v5.0.1 of the project using

```
svn checkout file:///usr/local/Version5_beta/tags/v5.0.1 Version5_wc
```

A developer A has already checkout a development version of the project some time ago, and want to update it to the most recent development version:

```
cd Version5_wc
svn update .
```

The `update` operation does not destroy the modifications already made by developer A in its working copy; the modifications committed by other users are automatically merged to developer's A modifications. A conflict may occurs if developer A has modified a line also modified and committed by another developer. In this case, the commit operation write a `C` before the conflicting file. In the following message,

```
alainhebert$ cd Version5_wc
alainhebert$ svn update .
C    readme
Updated to revision 9.
```

the message indicates that file `readme` has a conflicting modification and needs further attention from developer A. The file `readme` was modified in such a way to highlight the conflict:

```
#
# Instructions for configuring Version5 of Dragon/Donjon.
<<<<<<< .mine
# test pas allo
=======
# test allo
>>>>>>> .r9
#
```

The `revert` operation is more radical than `update` as it destroy developer A modifications and revert its working copy to the official revision of the repository:

```
cd Version5_wc
svn revert -R .
```

A developer want to delete file `Version5_wc/Dragon/abcd` from its working copy:

```
cd Version5_wc/Dragon/
svn delete abcd
```

Note that `abcd` will not be deleted from the official revision of the repository until a `commit` operation is performed. Even when the deletion is committed, it is always possible to recover this file by asking for a previous repository revision. Nothing is never lost.

A developer want to add file `efgh` in directory `Version5_wc/Dragon/` of its working copy:

```
cd Version5_wc/Dragon/
svn add efgh
```

Here, file `efgh` is now a candidate for inclusion in the repository at the next `commit` operation. Any file present in the working copy but not added with `svn add` *will not be committed in the repository*. This feature will be useful for performing configuration management. Although possible, *we highly recommend to avoid adding binary files with the* `svn add` *operation.* Binary files may be created in the working copy due to configuration management, but should not be committed.

Finally, a developer may want to commit (aka *check-in*) its modifications to the repository. Before committing anything, it is a good idea to check the status of the working copy using

```
cd Version5_wc/
svn status .
```

The commit operation is next performed using

```
cd Version5_wc/
svn commit -m 'issue000023: Introduction of hexagonal SPN capabilities' .
```

The commit operation *must* be performed with a commit message starting with the *issue identifier*, a string of the form `issuennnnnn:`. The number `nnnnnn` is a six-digit integer, with leading zeros, representing the issue associated to the commit. Its meaning will become clear in Section 3. Its value is equal to an existing value or equal to the maximum existing value plus one.

Committing modifications from a working copy is an important operation as it cannot be reverted. A developer committing information for the first time should always be accompanied by an experimented developer and should have obtained permission from the code-keeper of the sub-project affected by the `commit` operation.

Each `commit` operation will increase the revision index of the repository by two, as a second commit is automatically performed by a post-commit script to update the issue-tracking information.

If a developer want to modify the issue-tracking information, the commit message *must* be a twelve-character string of the form

```
'issuennnnnn:'
```

in order to avoid the second automatic commit. A developer may want to add information to the `issue000023` card-index present in the `issues_wc` working copy and commit this information using

```
cd issues_wc/
svn commit -m 'issue000023:' issue000023
```

Finally, we consider the operations related to the production of a frozen version:

1. Modify the `REV=` line of `Version5_wc/Ganlib/src/KDRVER.f` subroutine to reflect the *tag identifier* (`5.0.1` in this example) of the new frozen version:

```
*DECK KDRVER
      SUBROUTINE KDRVER(REV,DATE)
*
*----------------------------------------------------------------------
*
*Purpose:
* To extract CVS or SVN version and production date.
*
*Copyright:
* Copyright (C) 2006 Ecole Polytechnique de Montreal
*
*Author(s): A. Hebert
*
*Parameters: output
* REV     revision character identification
* DATE    revision character date
*
```

```
      *----------------------------------------------------------------------
      *
            CHARACTER REV*48,DATE*64
      *
            REV='Version 5.0.2 ($Revision: 509 $)'
            DATE='$Date: 2016-02-01 13:19:26 -0500 (Tue, 02 Feb 2016) $'
            IF(REV(22:).EQ.'ion$)') THEN
      *        CVS or SVN keyword expansion not performed
               REV='Version 5.0.6'
               DATE='February 2, 2016
            ENDIF
            RETURN
            END
```

2. Copy the `trunk` version to the `tags` directory:

   ```
   cd Version5_wc
   svn commit -m 'issue000049: Commiting KDRVER.f' .
   /usr/bin/svn copy . file:///usr/local/Version5_beta/tags/v5.0.2 \
   -m 'issue000049: Tagging the 5.0.2 release of Version5
   ```

   Note that a `commit` operation is performed inside the `copy` operation. The Subversion copy does not duplicate information inside the repository; only file increments are stored.

2.3 How it works

The pre- and post-commit scripts have been added to the repository to validate `commit` operations and to automatically perform the second automatic issue-tracking `commit`. Both scripts are written in Python and are based on the `pysvn` api.[8]

The pre-commit script reads

```
#!/usr/bin/python
"""
Subversion pre-commit hook which currently checks that the card-index
information is consistent and correctly given.
"""
#Author: Alain Hebert, Ecole Polytechnique, 2006.

import os, sys, pysvn

def main(repos, txn):
  # Recover the transaction data:
  t = pysvn.Transaction( repos, txn )
  all_props = t.revproplist()
  message = t.revproplist()['svn:log']
  #
  # Validate the commit message:
  if message[:5] != 'issue':
    sys.stderr.write ("Please begin your commit message with 'issue' characters. message=%s...\n"% \
    message[:15])
    sys.exit(1)
  try:
    cardIndexNumber =  int(message[5:11])
  except:
    sys.stderr.write ("Please begin your commit message with 'issue' characters followed" \
    +" by a six-digit index. message=%s...\n"%message[:15])
    sys.exit(1)
  fileName = message[:11]
  #
```

```
  # List of card-index
  client = pysvn.Client()
  myls = client.ls('file://'+repos+'/'+'/issues/')
  maxIssue = -1
  for k in range(len(myls)):
    maxIssue=max(maxIssue, int(myls[k]['name'].split('/')[-1][5:]))
  if int(fileName[5:]) > maxIssue+1:
    sys.stderr.write ("The six-digit index (%d) must be <= %d. message=%s...\n"% \
    (int(fileName[5:]), maxIssue+1, message[:15]))
    sys.exit(1)
  sys.exit(0)

if __name__ == '__main__':
  if len(sys.argv) < 3:
    sys.stderr.write("Usage: %s repos txn\n" % (sys.argv[0]))
  else:
    main(sys.argv[1], sys.argv[2])
```

The post-commit script reads

```
#!/usr/bin/python
"""
Subversion post-commit hook which copy (append) the issue-tracking information
to a new (or existing) card-index in the /issues/ directory. A commit of this
information is performed.
"""
#Author: Alain Hebert, Ecole Polytechnique, 2006.

import os, sys, pysvn, time

def main(repos, rev):
  # Recover the revision data:
  client = pysvn.Client()
  log_message=client.log('file://' + repos + '', discover_changed_paths=True, \
  revision_end=pysvn.Revision(pysvn.opt_revision_kind.number, rev))
  message = str(log_message[0]['message'])
  if message[11:] != ': Issue-tracking commit' and message[11:] != ':':
    # Recover the existing card-index
    fileName = str(log_message[0]['message'])[:11]
    if os.path.isdir('/tmp/post-issues'):
      os.system("chmod -R 777 /tmp/post-issues/")
      os.system("rm -r /tmp/post-issues/")
    myls = client.ls('file://'+repos+'/'+'/issues/')
    myls2 = []
    for k in range(len(myls)):
      myls2.append(str(myls[k]['name']).split('/')[-1])
    client.checkout('file://'+repos+'/'+'/issues/','/tmp/post-issues/',recurse=False)
    if fileName in myls2:
      # Recover the existing card-index and open it
      f = open('/tmp/post-issues/'+fileName, 'a')
    else:
      # Create a new card-index
      f = open('/tmp/post-issues/'+fileName, 'w')
      f.write('Card-index: '+fileName+'\n')
      f.write('-------------------------------------------------------\n')
      client.add('/tmp/post-issues/'+fileName)
    f.write(str(log_message[0]['author'])+'\n')
    f.write(time.ctime(log_message[0]['date'])+'\n')
    f.write('subversion revision=%d\n'%log_message[0]['revision'].number)
    f.write(message+'\n')
    for cpath in log_message[0]['changed_paths']:
```

```
      f.write(cpath['action']+' '+cpath['path']+'\n')
    f.write('------------------------------------------------------------\n')
    f.close()
    #committing the issue-tracking card-index to the repository
    client.cleanup('/tmp/post-issues/')
    client.checkin(['/tmp/post-issues/'], fileName+': Issue-tracking commit')
    os.system("rm -r -f /tmp/post-issues/")

if __name__ == '__main__':
  if len(sys.argv) < 3:
    sys.stderr.write("Usage: %s repos rev\n" % (sys.argv[0]))
  else:
    main(sys.argv[1], sys.argv[2])
```

Note the first line of these scripts indicating the position of the Python interpreter. On some system, the interpreter is located at `/usr/local/python`, so that the first line should be modified.

3 Issue tracking and spiral development management

As most software projects, Version5 is evolving with the spiral cycle development model shown in Fig. 3. The spiral development model consists of developing the product as a sequence of cycles; each of them devoted to the development of a single modification (called *project increment*) to any of the project components. Each cycle is tagged with a *issue identifier* of the form `issuennnnnn` used as reference through the development cycle. The traceability of the actions made by the developers is made possible by the introduction of this issue identifier.

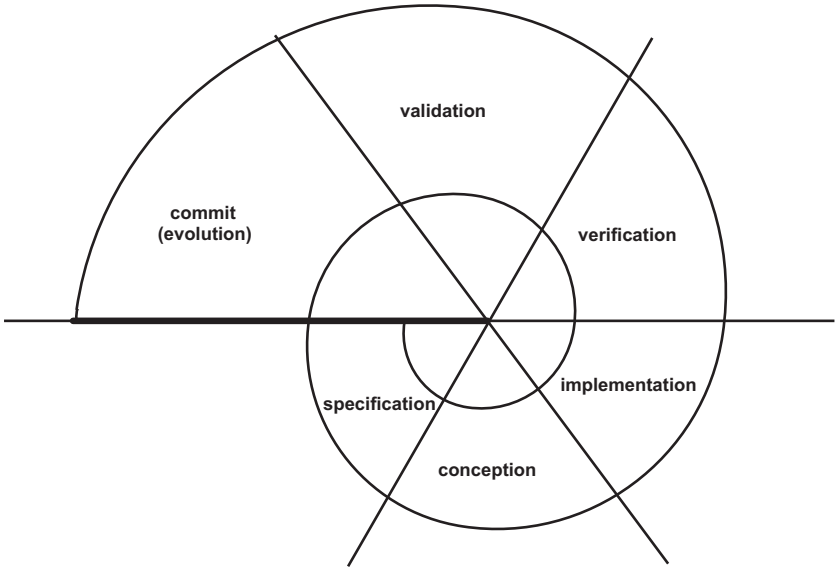

Figure 3: Spiral cycle development model

A cycle consists of the following steps:

1. A development cycle is *always* initiated by a developer at its own initiative or after receiving an issue submission form similar to the example shown in Fig. 4. Initiation of a new cycle *always* involve the assignment of an *issue identifier* of the form `issuennnnnn` where `nnnnnn` is equal to the maximum existing value plus one. This rule is followed whatever the type of project increment, and whatever the deliverable component to modify (including the scripts, non-regression tests, or documentation). Although a developer doesn't need to fill an issue submission form to start a cycle, it is a good practice to fill one if the increment involves more than a few days of work.

   A development cycle may be initiated by a user (or by a developer) sending an issue submission form. An issue submission form can be rejected by the developer in charge of the sub-project or can be accepted. If the project increment is accepted, the *issue identifier* is assigned and emailed to the user (or developer). Next, the issue identifier and core message of the issue submission form are copied to a file named `issuennnnnn` located in the `issues_wc` working copy. If the user sent attached files, a directory named `issuennnnnn_dir` can be created in the `issues_wc` working copy to hold these files. However, you should avoid to put huge amount of information in the `issues` directory. Finally, the `issues_wc` working copy is committed as

   ```
   cd issues_wc/
   svn commit -m 'issuennnnnn:' .
   ```

   File `issuennnnnn` is the *card-index* (*fiche d'intervention* in french) characterizing the cycle. It is automatically updated at each commit made during the development cycle. At any time during the cycle, the card-index can be updated by the developer in charge of the issue using

Figure 4: Example of an issue submission form.

```
cd issues_wc/
svn update issuennnnnn
```

modified and re-committed. The issue card-index trace the progress of the work made by the developer(s) to solve the issue. If the issue involve large programming efforts, it is also important to document the *closing* or final commit at the end of the *evolution* step (see Fig. 3).

2. The second step include the specification and conception work related to the issue. Here, the amount of work is highly issue-dependent. Some issues (such as assistance request) may not even require any specification and/or conception work; others may take years to complete. This step may involve proposed modifications in documentation (LCM object specifications and user's guide of the modules) and proposed unitary tests. This information is copied in the developer's working copy and can optionally be committed.

3. The third step is the programmation of the increment and its introduction in the developer's working copy. At the end of this step, the developer perform an `update` operation on its working copy to make sure that no conflicts with other developers exist.

4. The fourth step is the validation of the project increment and its commit in the repository. A set of selected non-regression tests are performed with the developer's working copy. If these tests are conclusive, the issue is closed and an *issue closing report* is written, appended to the card-index named `issuennnnnn` and emailed to the project user group. References to the issue-related documentation are also added to the card-index. A failure of the non-regression tests may require to come back to step 3 (or even to step 2 if a specification/conception error is detected). In case of success, both the card-index and the developer's working copy are committed to the repository.

4 Configuration management

Configuration management is the art of assembling the project components, available in the repository, in order to build the end product of the project. In case of Version5, the end-product is a set of executables for codes DRAGON, TRIVAC and DONJON on different UNIX-like operating systems (including PCs under Cygwin[9]) and a set of PDF reports.

Version5 configuration management uses the simplest existing approach as it requires relatively simple compiler technologies. In fact, the Version5 sources can be compiled with ISO Fortran–2003 and ANSI C compilers and its documentation can be compiled with LaTeX2ϵ. Version 5 configuration uses nothing more sophisticated than the technologies available in usual Unix distributions.

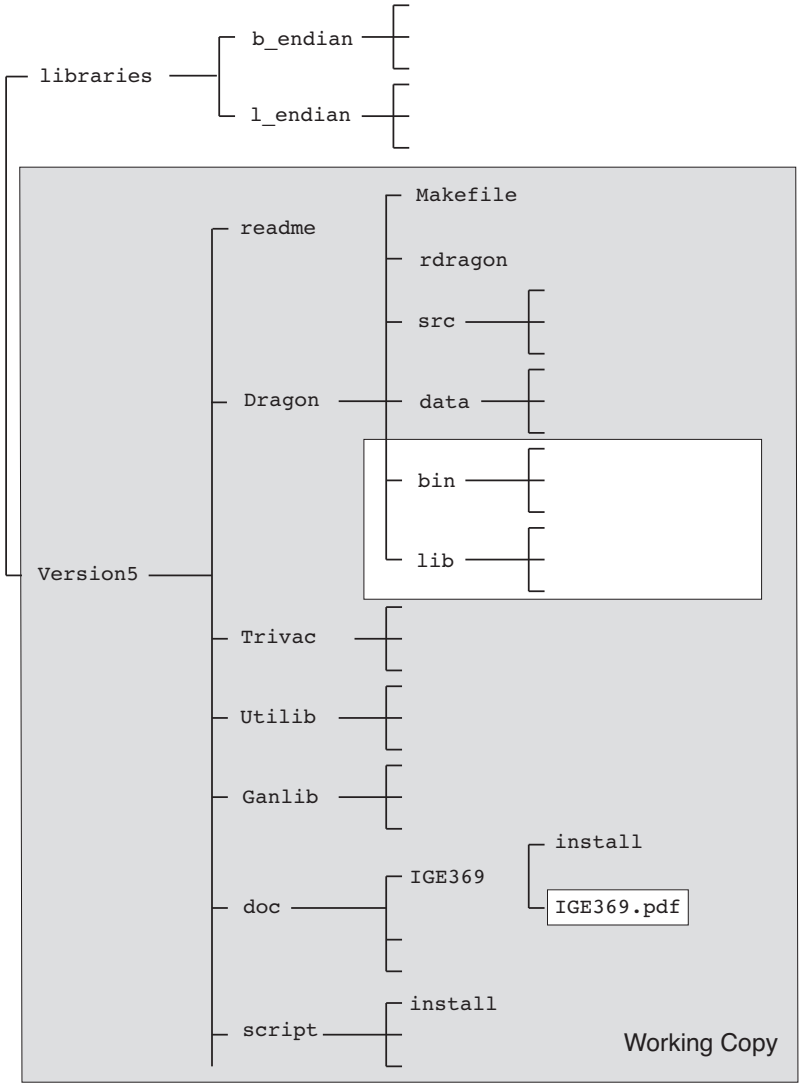

Figure 5: Result of configuration management.

The basic principle of Version5 configuration management consists to executing `install` and/or `make` scripts (`gmake` is used) *within* the user's or developer's working copy, as described in Fig. 5. Binary files (libraries, executables, PDF files) will be created but will not be managed by the version control system (**we must avoid committing any binary information**). A `commit` operation can still be performed on the working copy (without committing any binary information) if no `add` operation is done on this binary information.

For example, an executable of code DRAGON v5.0.1 with its documentation can be constructed using

```
svn checkout file:///usr/local/Version5_beta/tags/v5.0.1 Version5_wc
# build the DRAGON executable
cd Version5_wc/Dragon
make
cd ../..
#
# build the DRAGON documentation
cd Version5_wc/doc/IGE335
./install
cd ../../..
```

The installation of DRAGON5 include the installation of GANLIB5 and TRIVAC5. Similarly, the installation of DONJON5 include the installation of DRAGON5. Note that a generic `install` script is also available as `Version5_wc/script/install` to compile the Fortran sources. Each documentation directory has its own `install` script.

A test-case present in `Version5_wc/Dragon/data/` can then be executed using

```
cd Version5_wc/Dragon
./rdragon iaea2d.x2m
```

The directory `Version5_wc/Dragon/data/` is containing *non-regression test cases*, as shown in Fig. 5. They can be executed using the makefile:

```
cd Version5_wc/Dragon
make tests
```

Many of these testcases require the presence of the `libraries` directory. The user is responsible for setting this directory.

More detailed information is available in file `Version5_wc/readme`.

References

[1] See http://www.gnu.org/copyleft/lgpl.html.

[2] R. Roy, *The CLE-2000 Tool-Box*, Report IGE–163, Institut de Génie Nucléaire, École Polytechnique de Montréal, Montréal, Québec (1999).

[3] A. Hébert, "TRIVAC, A Modular Diffusion Code for Fuel Management and Design Applications", *Nucl. J. of Canada*, Vol. 1, No. 4, 325-331 (1987).

[4] G. Marleau, A. Hébert and R. Roy, "New Computational Methods Used in the Lattice Code Dragon," *Int. Top. Mtg. on Advances in Reactor Physics*, Charleston, USA, March 8-11, 1992.

[5] G. Marleau, A. Hébert and R. Roy, "A User Guide for DRAGON Version5", Report IGE-335, École Polytechnique de Montréal, Institut de Génie Nucléaire (2020).

[6] A. Hébert, J. Sekki and R. Chambon, "A User's Guide for DONJON Version5," Technical Report IGE-344, École Polytechnique de Montréal, Institut de Génie Nucléaire (2020).

[7] B. Collins-Sussman, B. W. Fitzpatrick and C. Michael Pilato, "Version Control with Subversion," O'Reilly Media Inc., USA, June 2004. See http://subversion.tigris.org.

[8] See http://pysvn.tigris.org.

[9] See http://www.cygwin.com.