Météo-France's GIT toolbox: GIT-tools

Version 1 English text by C. Fischer (2013/02/11)
based on Version 1 French text by S. Martinez (2012/09/10)

This document introduces the GIT toolbox, developed at Météo-France (MF) and used since CY39T1/CY40 as Source Code Repository (SCR) in Toulouse. Therefore, these "GIT-tools" also become a reference device for Aladin and Hirlam partners committing code to MF's based common libraries.

The note is a shortened translation of the comprehensive GIT-tools documentation written by S. Martinez (COMPAS/GCO of MF). The latter in French, describes in more details the reasons for the toolbox, with the former Clearcase (cc) handling in mind, and gives a more thorough explanation about the various command lines proposed in the toolbox. Like its French base version, this English note is not supposed to be a guide to GIT. For a reference documentation about GIT, we refer to standard on-line available tutorials.

Table of contents:

Concepts and functions in GIT:

Repositories (SCR) and branches:

For Clearcase (cc) familiar developers, we recall that all branches in cc are stored in a common central repository located on merou at ~marp001/dev. Access to the code is done via "views" on main releases, and extra existing branches can be added in the local view using the rules of view specifications. Much of this handling is transparent for those using ***cc_getpack*** or ***cc_getview***.

With GIT, this procedure is modified. Developers do not work directly on the central repository but on a local *clone*. The clone is a local copy stored anywhere below $HOME. The clone may be on a different server than the central repository, provided GIT is installed.

The concept of views disappears, and a direct access to the code is done in the classical Unix way (moving in the directories). The concept of a "public" or "private" view also disappears with GIT.

Branches still exist ! Any developer may modify any existing branch in his local repository,even is he is not the original author of that branch.

Although it is rather simple to create branches with native GIT, a specific command ***git_branch*** has been created in order to mimic ***cc_getpack*** (& even though, a rather simple access to any branch, and copy elsewhere, is possible within the GIT repository).

Modifications are local !:

Changes included in the local GIT SCR remain only local. They won't affect the central SCR. The changes are committed to the central repository either by the developer in “push” mode (provided the GIT administrator has given permission to do that) or in “pull” mode by the administrator. As a consequence, it is not easy to see what other users have changed or are changing, as long as the changes have not been committed to the central SCR.

GIT traceback:

Only individual files are traced back, not directories. An empty directory won't be traced at all. Files are traced (recognized) simply by their full name (for instance, arp/adiab/cpg.F90).

To create, suppress, rename or move a file, no specific GIT-tool command has been developed by GCO. Simply use the corresponding native GIT commands. Those are indeed safe and simple to use.

The “*commit*”:

This action corresponds to the ***cc_edit*** command in cc. ***cc_edit*** actually is performing 3 actions in one: check-out of an element (its local version is read-only); edit; check-in (which creates a new version in the cc branch).

In GIT, this is simpler: any element can be immediately modified by a simple edit, and more generally any creation/suppression/renaming/move is done using Unix commands. The “commit” will eventually upgrade the whole GIT branch in one go, not element-wise like in cc. Each “commit” is associated with a unique hexadecimal identification of 41 characters, denoted the “*commit-hash*”. Thus, the management of versions in GIT is global to the branch, not element-wise like in cc.

For the sake of commodity, 2 specific GIT-tools have been developed for editing a file or for committing a file: resp. ***git_edit*** (~ ***cc_edit***) and ***git_commit***. If files need to be renamed/moved and changed, we advise to first rename/move, then commit, then change its content (edit).

The concept of a “tag”:

In cc, tags are the specific names of versions like for instance CY38T1_bf.02 or CY38. In GIT, a “tag” behaves like an alias to a commit (= an alias associated to a given commit-hash).

Merging branches:

In cc, a merge action consists in inserting the content of one branch into the current branch (of a given view), in an incremental way. In GIT, mergers are simpler and the resolution of conflicts is more elaborate than with cc. GIT in principle is able to identify orthogonal changes in a file, and will handle them automatically.

In standard version, GIT does not offer a graphics interface for merge operations (unlike ***cc_merge***). However, freeware graphics interfaces exist that can be used with GIT. The command ***git_merge*** has been developed in order to work in a similar way than ***cc_merge***.

Other functions:

- GIT provides a traceback history of all changes, that can be viewed and handled for "*diffs*" between several versions. Like for the merge, GIT however requires external "diff" tools. Those do not allow to compare more than 3 different versions. 2 specific GIT-tools have been developed to ease those actions: ***git_history*** and ***git_diff***.
- Unlike cc, GIT has no standard tool for checking the content of a given branch (***cc_list***). This action is however extensively used in cc in order to check and identify versions and their tags. A specific GIT-tool has been developed to remedy this: ***git_list***.

The GIT-GCO meta-data server:

Why this meta-data server ?

- As discussed above, GIT in its standard version does not provide some usual information like the identification of the reference version on which a given local branch is based. In this particular case, a git log might be used which would show all the piling up of successive modifications for a file. That information could allow one to "guess" what the base version was/is. Nevertheless, we wish to get the exact information, and in a simple manner.
- The native git log command on a specific SCR element will provide lots of good information about the history of changes, like: author, date, commit-hash, documentation associated to each commit. However, for instance, the name of the branch in which a given commit has been inserted is missing ...
- we wish to replace the little readable commit-hash identification with an explicit "branch/version" tag, such as "marp001_CY38T1_bf/2"
- we wish to implement an administrator control for branches. Standard GIT allows any user to modify in his local clone any base branch, including those he has not created. This is too permissive in our view for the IFS/Arpège code management. A user identification mechanism therefore has been designed.
- Conversely, the user-restricted permissions can be tuned such as to allow specific users to change branches of designated other users.

Eventually, the need to extend some of cc's limitations, and the need to somewhat re-design GIT's functions, has lead to the elaboration of the meta-data server.

The GIT-GCO toolbox:

GIT-tools are a comprehensive set of Perl scripts that encapsulate the native GIT commands, in order to ease their usage and implement some of the above concepts. GIT-tools therefore also manage the handle to the meta-data server. For many standard developers' needs, GIT-tools will be the sufficient toolbox for work; some specific actions however remain to be done with native GIT commands (see later).
The list of tools/commands can be found in the Appendix.

At present, GIT-tools are installed on the following platforms: merou (central SCR server), yuki & kumo (MF's NEC SX9), tori (MF NEC SX8), serran & mostelle (MF Production servers), mirage (COMPAS server). More installations will come soon.

Tutorial:

At first ... get started !: run ***git_start***:

Before you use GIT-tools at any time, on any platform, you need to initialize your environment by running git_start (located in the directory ~marp001/git-install/client/default/bin).

At run time, ***git_start*** will first ask you whether you do have a GIT user account. If so, answer 'y', otherwise answer 'n' (or RETURN). The script will proceed normally in any case. Note that if you do not yet have a GIT user account, please check with MF's GCO team[1] and the French LTM[2] to get a valid account[3].

At the end of the execution of ***git_start***, you should get the following type of return output:

```
% ~marp001/git-install/client/default/bin/git_start
check env variable GIT_INSTALL ... undef
check env variable GIT_ROOTPACK ... undef
check env variable GIT_HOMEPACK ... undef
check env variable GIT_WORKDIR ... undef
check presence of GIT_BIN_PATH in user's path ... undef
check presence of GIT_CORE_PATH in user's path ... undef
check presence of GIT_CLIENT_PATH in user's path ... undef

You are not authentified. Do you already have a Git account ? [y/n] y
Please authentify yourself.
login : stef
passwd: XXXXXXXX
You are authentified as: stef

set global configuration...

> try to clone master repository ...
Initialized empty Git repository in /home/marp/marp003/git-dev/arpifs/.git/
OK

> please add those environement variables to your profile:
export GIT_INSTALL="/home/marp/marp001/git-install"
export GIT_ROOTPACK="git://mirage"
export GIT_HOMEPACK="/home/marp/marp003/git-dev"
export GIT_WORKDIR="/home/marp/marp003/.git-workdir"

> please add those variables to your PATH:
/home/marp/marp001/git-install/default/bin
/home/marp/marp001/git-install/default/libexec/git-core
/home/marp/marp001/git-install/client/default/bin

this output is available here: /home/marp/marp003/git_start.log
```

1 gco@meteo.fr
2 At the time of writing this note (January 2013) : Claude.Fischer@meteo.fr
3 GIT user accounts on MF servers will be granted to NWP consortia partners on a case by case basis, but basically for all Aladin source code contributors (such as ALARO contact persons, other identified contributors to a given cycle, phasers) and the Hirlam System Experts.

In this example, user “stef” starts from a fully uninitialized GIT environment. He therefore has to set the following variables in his .profile/.bash_profile/etc... file:

- GIT_INSTALL: GIT-tools repository directory
- GIT_ROOTPACK: location of central SCR
- GIT_HOMEPACK: location of the user's local SCR (clone)
- GIT_WORKDIR: user temporary working directory
- don't forget to also add the location of the native GIT commands, as well as the GIT-tools commands, in your PATH environmental variable

In case of problems or suspicions: help desk is gco@meteo.fr
In the sequel of this note, we assume the user is a well authenticated GIT user as described above.

Access to the GIT SCR – how to create a branch:

In addition to the start initialization, ***git_start*** will create a local copy of the central SCR, that is, a clone. In our case, the Arpège/Aladin/Arome source code will be stored in a local repository named “arpifs”, located in the directory $GIT_HOMEPACK:

```
% cd $GIT_HOMEPACK
% ls -l
total 4
drwxr-xr-x 22 marp003 marp 4096 jun 12 13:53 arpifs

% cd arpifs
% ls -l
total 76
drwxr-xr-x 44 marp003 marp 4096 jun 12 13:53 aeolus
drwxr-xr-x 18 marp003 marp 4096 jun 12 13:53 aladin
drwxr-xr-x  6 marp003 marp 4096 jun 12 09:20 algor
drwxr-xr-x 39 marp003 marp 4096 jun 12 13:53 arpifs
drwxr-xr-x  8 marp003 marp 4096 jun 12 09:20 biper
drwxr-xr-x  7 marp003 marp 4096 jun 12 13:53 blacklist
drwxr-xr-x  6 marp003 marp 4096 jun 12 09:20 etrans
drwxr-xr-x 22 marp003 marp 4096 jun 12 13:53 ifsaux
drwxr-xr-x  8 marp003 marp 4096 jun 12 09:20 mpa
drwxr-xr-x  9 marp003 marp 4096 jun 12 13:53 mse
drwxr-xr-x  9 marp003 marp 4096 jun 12 09:20 obstat
drwxr-xr-x 27 marp003 marp 4096 jun 12 13:53 odb
drwxr-xr-x 15 marp003 marp 4096 jun 12 13:53 satrad
drwxr-xr-x 11 marp003 marp 4096 jun 12 13:53 scat
drwxr-xr-x 16 marp003 marp 4096 jun 12 09:20 scripts
drwxr-xr-x  7 marp003 marp 4096 jun 12 09:20 surf
drwxr-xr-x  5 marp003 marp 4096 jun 12 13:53 surfex
drwxr-xr-x  7 marp003 marp 4096 jun 12 09:20 trans
drwxr-xr-x 17 marp003 marp 4096 jun 12 13:53 utilities
```

For the cc familiar reader: please notice from the above example that the former “cc projects” have

been renamed: arp => arpifs; ald => aladin; xla => algor; bip => biper; bla => blacklist; tfl => trans; tal => etrans; xrd => ifsaux; obt => obstat; sat => satrad; sct => scat; scr => scripts; sur => surf; uti => utilities.
The local version in this clone simply is the current last version available in the central SCR, at the very time git_start was launched. Use ***git_view*** to easily learn about the history of this version in the central repository:

```
% git_view
CY37T1_op1.08
CY37T1_bf.03
CY37T1
```

Thus, this command shows how successive versions pile up, like ***cc_view*** did in cc. At the date of 2012/09/06, all cycles and versions from CY18 through CY38T1 are already made available. A number of GCO and specific user branches also are present.

To create a user-local branch, use ***git_branch*** from the GIT-tools. This command works in a similar manner to ***cc_getpack***. For example, for creating a branch on top of version 05 of MF's E-suite version CY37T1_op1, type:

```
% git_branch -r 37t1 -b op1 -v 05 -u test
 > Do you want to create branch stef_CY37T1_test ? [y/n] y

 >
 > Branch : stef_CY37T1_test
 > Path   : /home/marp/marp003/git-dev/arpifs
 >
 >  1. CY37T1_op1.05
 >  2. CY37T1_bf.03
 >  3. CY37T1
 >
 > Available projects:
 >   aeolus aladin algor arpifs biper blacklist etrans ifsaux mpa mse obstat
odb satrad scat scripts surf surfex trans utilities
 >
```

You can notice that the naming of branches starts with the GIT user name (as opposed to the Unix system user name, which can differ from a platform to another. “stef” is a unique definition for the GIT user both in the central SCR and for clones. “marp001” may become “gco” as a Unix user name on a different machine, or “stef”).

<u>How to change files in your branch</u>:

As explained earlier in this note, you do not have to “checkout” nor to “check-in” any code change in a file. So basically GIT is comfortable with only edits (***vi***, ***vim***, etc. Ended by a save/quit). However, cc's ***cc_edit*** command had two specific functions of interest included:

1. any file name is searched throughout the whole cc view. Therefore, one could ***cc_edit*** for instance cpg.F90 from any directory (not only from “adiab”).
2. At the beginning of ***cc_edit***, the command return any potential list of branches in which the edited file already has been modified or is being edited, if those branches are based on the same release.

In order to preserve these two functions under GIT, a GIT-tool named ***git_edit*** has been developed. For instance, when the routine apl_arome.F90 is edited from the directory $GIT_HOMEPACK/arpifs, the following output is obtained:

```
% git_edit apl_arome.F90
 > Warning: apl_arome.F90 not found in current directory
 > Getting file arpifs/phys_dmn/apl_arome.F90 ? [y/n] y
 > Versions of arpifs/phys_dmn/apl_arome.F90 - cycle CY37T1:
CY37T1_op1.01     GCO         modified     2012/05/14-08:30:26
CY37T1_bf.02      GCO         modified     2012/05/14-08:24:02
 > Still getting file arpifs/phys_dmn/apl_arome.F90 ? [y/n] y
```

The desired changes are inserted in apl_arome.F90, and the edit is saved/quit. ***git_edit*** then issues the following request for confirmation:

```
> Save modifications ? [y/n]
```

At this stage, the user still can decide whether to accept the changes, or not.

<u>How to suppress/rename/add files</u>:

For suppressing files, the corresponding native GIT command has been found well suited for our SCR management needs. It is a rather simple command:

```
% git rm [-r] fichier-à-effacer [fichier-à-effacer ...]
```

Option ***-r*** allows to suppress one or more whole directories.

Another native GIT command allows to rename files or directories, or to move files/directories:

```
% git mv source [source ...] destination
```

CAUTION:
GIT will not complain if one uses the standard Unix system ***rm*** command for removing files/directories. However, using the ***mv*** command may break GIT's ability to maintain the historical traceback of the element of concern. Therefore, we recommend not to use ***mv*** !

To add new files/directories, many Unix commands are perfectly accepted and will work with GIT: ***touch***, ***mkdir***, ***cp***, ***vim***, etc.

A file is tracebacked by GIT using its fully extended name (for instance, arpifs/phys_dmn/apl_arome.F90). If a file has been removed in an old branch, and one wishes to re-create this file, there is no need for a wrapped command like ***cc_add*** was in cc. It is simply sufficient to re-create the file as indicated above, and one retrieves the full old history of the previously removed file.

How to register your modifications:

For any of the changes performed locally, as described before, 2 operations are necessary for registration with native GIT commands:

1. run ***git add*** in order to update the file index of the local repository
2. run ***git commit*** (without the underscore !) to register the changes

The GIT-tools command ***git_commit*** performs both operations in one go. A specific documentary message needs to be provided in order to submit a description of those changes. For a short message, option ***-m*** of ***git_commit*** is enough:

```
% git_commit -m « ceci est mon premier commit »
```

For bigger changes, or when a substantial description is desired (one or several paragraphs), option ***-f*** of ***git_commit*** will be used. The argument of the option then is a separate file, located outside the SCR, and containing the message:

```
% git_commit -f /tmp/mon_message
```

Update of the meta-data server, and of the central repository:

Thus far, all the performed actions and changes only affected the local GIT SCR. The central SCR was not modified. Obviously, this allows any developer to take a while and possibly revisit his/her changes and documentation. Nevertheless, once the changes are found consolidated and ripe for a definitive commitment, they can be sent to the central repository. This operation is done with the command ***git_post***:

```
% git_post
stef_CY37T1_test/1
```

In a normal case of execution, ***git_post*** will, after a few minutes delay, send back the updated version to the developer's local current branch (synchronization step). The command ***git_view*** then should return a picture like this:

```
% git_view
stef_CY37T1_test/1
CY37T1_op1.05
CY37T1_bf.03
CY37T1
```

Note that it is not necessary to ***git_post*** changes one by one. You certainly should/can leave your changes committed locally, and only update the central SCR when a complete, firm set of changes has been obtained. Indeed, ***git_post*** will preserve all your intermediate changes/commits you did locally, and they will remain in the history traceback udpated in the central SCR, and be sent back to your local branch.

<u>How to merge branches</u>:

The native GIT command ***git merge name_of_branch*** does allow very simple merges. However, this command is unable to tell in advance of starting the actual operation on files, which type of actions will be conducted. We wish to keep this facility, which is provided within cc. This is the reason for having a specific GIT-tools command ***git_merge***, which operates in a similar manner to cc's ***cc_merge***. Before merging, information about what has to be merged will be issued (in terms of suppression, renaming, adds, modifications) and of what type each action will be (automatic or manual).

For manual merges, GIT unfortunately does not provide a standard graphical interface. External interfaces can however be used, as desired, among for instance ***vimdiff, gvimdiff, tkdiff, meld, kdiff3, etc.*** GCO is proposing ***kdiff3*** since this interface bear many similarities with the one used with ***cc_merge***.

CAUTION:
kdiff3 only is available on merou for the time being. Therefore, the sequel of this tutorial applies

strictly speaking only to this platform.

As an example of how to use ***git_merge***, we first create a new branch named “merge” on top of CY36T1:

```
% git_branch -r 36t1 -u merge
 > Do you want to create branch stef_CY36T1_merge ? [y/n] y


 >
 > Branch : stef_CY36T1_merge
 > Path   : /home/marp/marp003/git-dev/arpifs
 >
 >  1. CY36T1
 >
 > Available projects:
 >   aeolus aladin algor arpifs biper blacklist etrans ifsaux mpa mse obstat
odb satrad scat scripts surf surfex trans utilities
 >
```

The example consists in merging all the changes present in the bugfix branch of this cycle. This is GCO's branch gco_CY36T1_bf:

```
% git_merge -u gco -r 36t1 -b bf
 > Merging from branch origin/gco_CY36T1_bf ...
 > Base is 9ee6fa1dd782a2302ee71070d83d1a64859a8f43 ...
Add File "arpifs/c9xx/csstbld.F90" (automatic)
Add File "arpifs/function/fcgeneralized_gamma.h" (automatic)
Add File "ifsaux/utilities/ismax_1.F" (automatic)
Add File "ifsaux/utilities/ismin_1.F" (automatic)
Add File "odb/bufr2odb/satobfreq_bynam.F90" (automatic)
Add File "satrad/interface/rttov_ec_alloc.h" (automatic)
Merge File "aladin/adiab/espchor.F90" (automatic)
Merge File "aladin/control/espcm.F90" (automatic)
Merge File "aladin/programs/blendsur.F90" (automatic)
Merge File "aladin/programs/check_limits.F90" (automatic)
[...]
Merge File "surfex/teb/init/writesurf_pgd_teb_parn.f90" (automatic)
Merge File "surfex/teb/phys/urban_drag.f90" (automatic)
Merge File "utilities/ctpini/module/constantes.F90" (automatic)
Merge File "utilities/ctpini/module/fonctions_inversion.F90" (automatic)
Merge File "utilities/ctpini/programs/inversion_master.F90" (automatic)
Merge File "utilities/pinuts/module/egg_tools_mod.F90" (automatic)

deleted files     : 0
renamed files     : 0
added files       : 6
automatic merges : 146
manual merges     : 0
```

```
 > Do you want to perform automatic merges ? [y/n] y
Updating 9ee6fa1..092ef19
Fast-forward
 aladin/adiab/espchor.F90                             |  184 ++----
 aladin/control/espcm.F90                             |  133 ++++-
 aladin/programs/blendsur.F90                         |   23 +-
 aladin/programs/check_limits.F90                     |   18 +-
 aladin/utility/elalo2xy.F90                          |    3 +
 aladin/var/suescal.F90                               |    2 +-
 algor/external/fourier/fft992.F                      |    1 +
 arpifs/adiab/call_sl_ad.F90                          |    4 +-
 arpifs/adiab/laitri.F90                              |    5 +-
 arpifs/adiab/larcin2ad.F90                           |    2 +-
 arpifs/c9xx/csstbld.F90                              |  259 +++++++++
[...]
 utilities/ctpini/module/fonctions_inversion.F90 |  234 +++++---
 utilities/ctpini/programs/inversion_master.F90  |    7 +-
 utilities/pinuts/module/egg_tools_mod.F90       |   68 ++-
 152 files changed, 3228 insertions(+), 2745 deletions(-)
 create mode 100644 arpifs/c9xx/csstbld.F90
 create mode 100644 arpifs/function/fcgeneralized_gamma.h
 create mode 100644 ifsaux/utilities/ismax_1.F
 create mode 100644 ifsaux/utilities/ismin_1.F
 create mode 100644 odb/bufr2odb/satobfreq_bynam.F90
 create mode 100644 satrad/interface/rttov_ec_alloc.h
```

On execution, the reader will first notice the speed of this operation (as compared, for those familiar, with ***cc_merge*** operations). Second, the second line in the output of ***git_merge*** reads: “Merging from branch origin/gco_CY36T1_bf”. What the hell does “origin” mean here ?
This is linked with the fact that gco_CY36T1_bf is not a branch on the local repository, but on the central one (that is, the one that has been cloned previously). It is in the jargon named “a distant branch”. Had we used the GIT native command ***git merge gco_CY36T1_bf***, we would have got an error message: “fatal: gco_CY36T1_bf – not something we can merge”. This type of specific GIT behavior, forcing one to specifically understand the nature of a given branch, is avoided by the use the ***git_merge*** instead. Note that a similar extended facility is associated with ***git_branch***.

If now ***git_view*** is run, we obtain:

```
% git_view
CY36T1_bf.09
CY36T1
```

which is the normal message, since we simply have merged all bugfixes on top of the originally cloned base version CY36T1.

At present, branch “gco_CY36T1_op1” will be merged as an additional content:

```
% git_merge -u gco -r 36t1 -b op1
 > Merging from branch origin/gco_CY36T1_op1 ...
 > Base is 092ef191a05cf29d586d82fa1c8c1dd83d97bf9b ...
Remove File "arpifs/parallel/disgrid.F90" (automatic)
Remove File "arpifs/parallel/diwrgrid.F90" (automatic)
Remove File "mpa/turb/internals/updraft_sope.f90" (automatic)
Rename File "arpifs/module/yomarar.F90" -> "arpifs/module/yomparar.F90"
(automatic)
Add File "arpifs/dia/wrgrida.F90" (automatic)
Add File "arpifs/module/disgrid_mod.F90" (automatic)
Add File "arpifs/module/diwrgrid_mod.F90" (automatic)
Add File "arpifs/namelist/namfpdyi.h" (automatic)
Add File "arpifs/obs_preproc/sortscatidx.F90" (automatic)
[...]
Merge File "utilities/progrid/procor2.F" (automatic)
Merge File "arpifs/phys_dmn/achmt.F90" (manual)
Merge File "arpifs/phys_dmn/mf_phys.F90" (manual)
Merge File "arpifs/setup/sugrida.F90" (manual)
Merge File "mpa/turb/internals/compute_updraft.f90" (manual)
Merge File "surfex/sea/phys/coupling_seaflux_sbl.f90" (manual)

deleted files    : 3
renamed files    : 1
added files      : 118
automatic merges : 249
manual merges    : 5

 > Do you want to perform automatic merges ? [y/n] y
```

The presence of manual merges may be surprising. This will however be a rather frequent situation. In our case, the reason for manual merges is that “op1” is a branch based on bugfix version 06 of CY36T1, while the latest bugfix version already is 09.
At first, the manual merges will be performed:

```
Trying really trivial in-index merge...
Nope.
Trying simple merge.
Simple merge failed, trying Automatic merge.
Auto-merging arpifs/dia/cpxfu.F90
Auto-merging arpifs/namelist/namphy0.h
Auto-merging arpifs/op_obs/hretr.F90
Auto-merging arpifs/phys_dmn/achmt.F90
ERROR: content conflict in arpifs/phys_dmn/achmt.F90
Auto-merging arpifs/phys_dmn/mf_phys.F90
ERROR: content conflict in arpifs/phys_dmn/mf_phys.F90
Auto-merging arpifs/pp_obs/pos.F90
Auto-merging arpifs/setup/sugrida.F90
ERROR: content conflict in arpifs/setup/sugrida.F90
Auto-merging arpifs/var/rdfpinc.F90
Auto-merging mpa/turb/internals/compute_updraft.f90
ERROR: content conflict in mpa/turb/internals/compute_updraft.f90
Auto-merging odb/pandor/module/bator_decodbufr_mod.F90
Auto-merging odb/pandor/module/bator_ecritures_mod.F90
Auto-merging odb/pandor/module/bator_init_mod.F90
```

```
Auto-merging odb/pandor/module/bator_lectures_mod.F90
Auto-merging surfex/sea/phys/coupling_seaflux_sbl.f90
ERROR: content conflict in surfex/sea/phys/coupling_seaflux_sbl.f90
Automatic merge failed; fix conflicts and then commit the result.

 > Do you want to perform manual merges ? [y/n] y
```

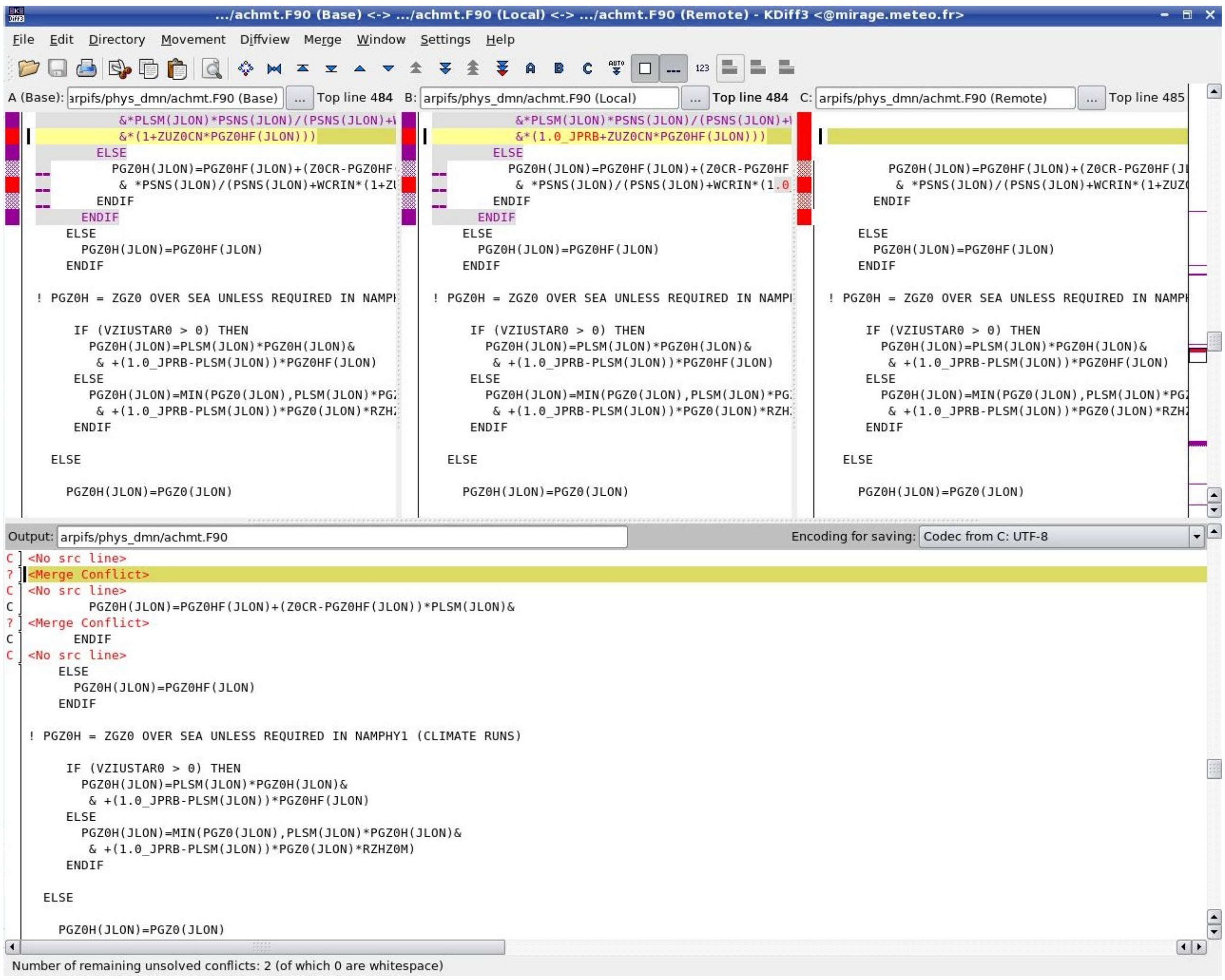
A few comments here:

1. so-called “fast-forward” manual merges are not listed above (they mean that the branch gco_CY36T1_op1 is the sole contributor)
2. merges solved as “Auto-merging” without errors are those declared as manual by GIT, but nevertheless solved by GIT (for instance, orthogonal contributions)
3. “Auto-merging” merges that issue an error are those where a true manual merge is required (thus, to be taken care by the developer)

As with cc, the remaining manual merges are done file by file, with a request for confirmation:

```
Merging:
arpifs/phys_dmn/achmt.F90
arpifs/phys_dmn/mf_phys.F90
arpifs/setup/sugrida.F90
mpa/turb/internals/compute_updraft.f90
surfex/sea/phys/coupling_seaflux_sbl.f90

Normal merge conflict for 'arpifs/phys_dmn/achmt.F90':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (kdiff3):
```

Typing RETURN, the ***kdiff3*** interface will be open on screen:


.../achmt.F90 (Base) <-> .../achmt.F90 (Local) <-> .../achmt.F90 (Remote) - KDiff3 <@mirage.meteo.fr>

A (Base): arpifs/phys_dmn/achmt.F90 (Base) Top line 484

```
            &*PLSM(JLON)*PSNS(JLON)/(PSNS(JLON)+\
            &*(1+ZUZ0CN*PGZ0HF(JLON)))
         ELSE
           PGZ0H(JLON)=PGZ0HF(JLON)+(Z0CR-PGZ0HF
            & *PSNS(JLON)/(PSNS(JLON)+WCRIN*(1+ZU
         ENDIF
       ENDIF
     ELSE
       PGZ0H(JLON)=PGZ0HF(JLON)
     ENDIF

 ! PGZ0H = ZGZ0 OVER SEA UNLESS REQUIRED IN NAMPH

      IF (VZIUSTAR0 > 0) THEN
        PGZ0H(JLON)=PLSM(JLON)*PGZ0H(JLON)&
         & +(1.0_JPRB-PLSM(JLON))*PGZ0HF(JLON)
      ELSE
        PGZ0H(JLON)=MIN(PGZ0(JLON),PLSM(JLON)*PG
         & +(1.0_JPRB-PLSM(JLON))*PGZ0(JLON)*RZH
      ENDIF

   ELSE

     PGZ0H(JLON)=PGZ0(JLON)
```

B: arpifs/phys_dmn/achmt.F90 (Local) Top line 484

```
            &*PLSM(JLON)*PSNS(JLON)/(PSNS(JLON)+\
            &*(1.0_JPRB+ZUZ0CN*PGZ0HF(JLON)))
         ELSE
           PGZ0H(JLON)=PGZ0HF(JLON)+(Z0CR-PGZ0HF
            & *PSNS(JLON)/(PSNS(JLON)+WCRIN*(1.0
         ENDIF
       ENDIF
     ELSE
       PGZ0H(JLON)=PGZ0HF(JLON)
     ENDIF

 ! PGZ0H = ZGZ0 OVER SEA UNLESS REQUIRED IN NAMPH

      IF (VZIUSTAR0 > 0) THEN
        PGZ0H(JLON)=PLSM(JLON)*PGZ0H(JLON)&
         & +(1.0_JPRB-PLSM(JLON))*PGZ0HF(JLON)
      ELSE
        PGZ0H(JLON)=MIN(PGZ0(JLON),PLSM(JLON)*PG
         & +(1.0_JPRB-PLSM(JLON))*PGZ0(JLON)*RZH
      ENDIF

   ELSE

     PGZ0H(JLON)=PGZ0(JLON)
```

C: arpifs/phys_dmn/achmt.F90 (Remote) Top line 485

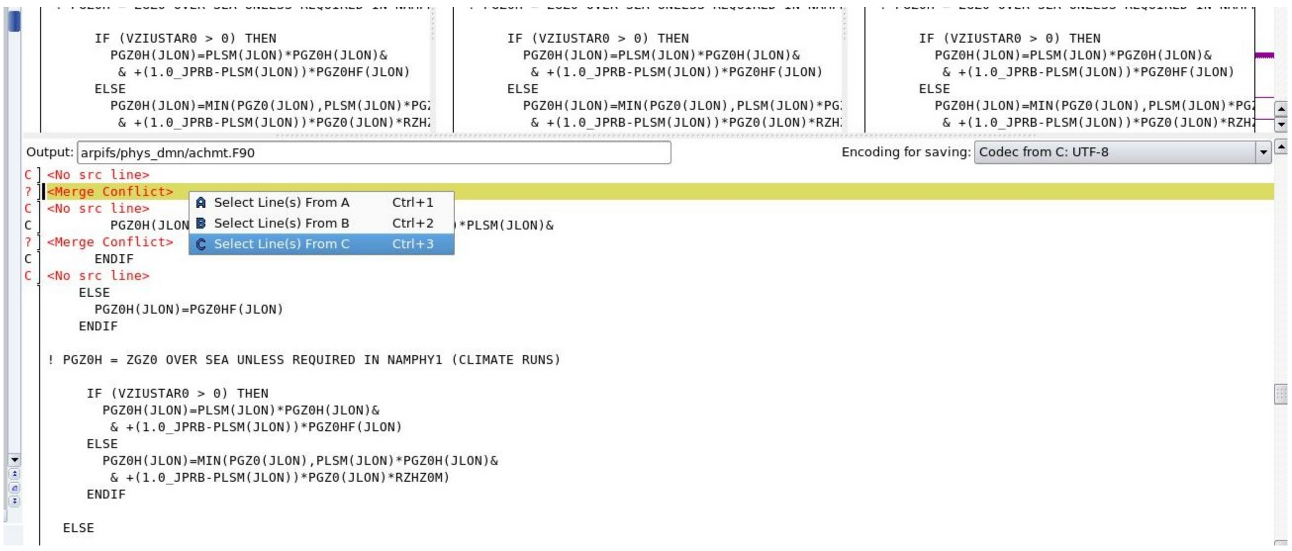
```

           PGZ0H(JLON)=PGZ0HF(JLON)+(Z0CR-PGZ0HF(J
            & *PSNS(JLON)/(PSNS(JLON)+WCRIN*(1+ZUZ
         ENDIF

     ELSE
       PGZ0H(JLON)=PGZ0HF(JLON)
     ENDIF

 ! PGZ0H = ZGZ0 OVER SEA UNLESS REQUIRED IN NAMPH

      IF (VZIUSTAR0 > 0) THEN
        PGZ0H(JLON)=PLSM(JLON)*PGZ0H(JLON)&
         & +(1.0_JPRB-PLSM(JLON))*PGZ0HF(JLON)
      ELSE
        PGZ0H(JLON)=MIN(PGZ0(JLON),PLSM(JLON)*PG
         & +(1.0_JPRB-PLSM(JLON))*PGZ0(JLON)*RZH
      ENDIF

   ELSE

     PGZ0H(JLON)=PGZ0(JLON)
```

Output: arpifs/phys_dmn/achmt.F90   Encoding for saving: Codec from C: UTF-8

```
C ] <No src line>
? ] <Merge Conflict>
C ] <No src line>
C           PGZ0H(JLON)=PGZ0HF(JLON)+(Z0CR-PGZ0HF(JLON))*PLSM(JLON)&
? ] <Merge Conflict>
C         ENDIF
C ] <No src line>
        ELSE
          PGZ0H(JLON)=PGZ0HF(JLON)
        ENDIF

  ! PGZ0H = ZGZ0 OVER SEA UNLESS REQUIRED IN NAMPHY1 (CLIMATE RUNS)

         IF (VZIUSTAR0 > 0) THEN
           PGZ0H(JLON)=PLSM(JLON)*PGZ0H(JLON)&
            & +(1.0_JPRB-PLSM(JLON))*PGZ0HF(JLON)
         ELSE
           PGZ0H(JLON)=MIN(PGZ0(JLON),PLSM(JLON)*PGZ0H(JLON)&
            & +(1.0_JPRB-PLSM(JLON))*PGZ0(JLON)*RZHZ0M)
         ENDIF

      ELSE

        PGZ0H(JLON)=PGZ0(JLON)
```

Number of remaining unsolved conflicts: 2 (of which 0 are whitespace)

In the upper window, we successively find: the base version common to both the current branch and the branch to merge (Base), the current version (Local), the version that is being merged to it (Remote). The lower window displays the result of the merge.
Automatic conflicts are solved by ***kdiff3***, and the remaining conflicts need to be visually handled. Those are the ones that remain displayed in the lower window as <Merge Conflict>. ***Kdiff3*** situates the edited file immediately at the level of the first unsolved conflict, and underlines in yellow the lines to be solved.
For the conflict displayed above, one simply puts the cursor on the yellow-marked line, right-clicks, and chooses the option “Select Line(s) from C”:



```
      IF (VZIUSTAR0 > 0) THEN
        PGZ0H(JLON)=PLSM(JLON)*PGZ0H(JLON)&
         & +(1.0_JPRB-PLSM(JLON))*PGZ0HF(JLON)
      ELSE
        PGZ0H(JLON)=MIN(PGZ0(JLON),PLSM(JLON)*PG
         & +(1.0_JPRB-PLSM(JLON))*PGZ0(JLON)*RZH
```

```
      IF (VZIUSTAR0 > 0) THEN
        PGZ0H(JLON)=PLSM(JLON)*PGZ0H(JLON)&
         & +(1.0_JPRB-PLSM(JLON))*PGZ0HF(JLON)
      ELSE
        PGZ0H(JLON)=MIN(PGZ0(JLON),PLSM(JLON)*PG
         & +(1.0_JPRB-PLSM(JLON))*PGZ0(JLON)*RZH
```

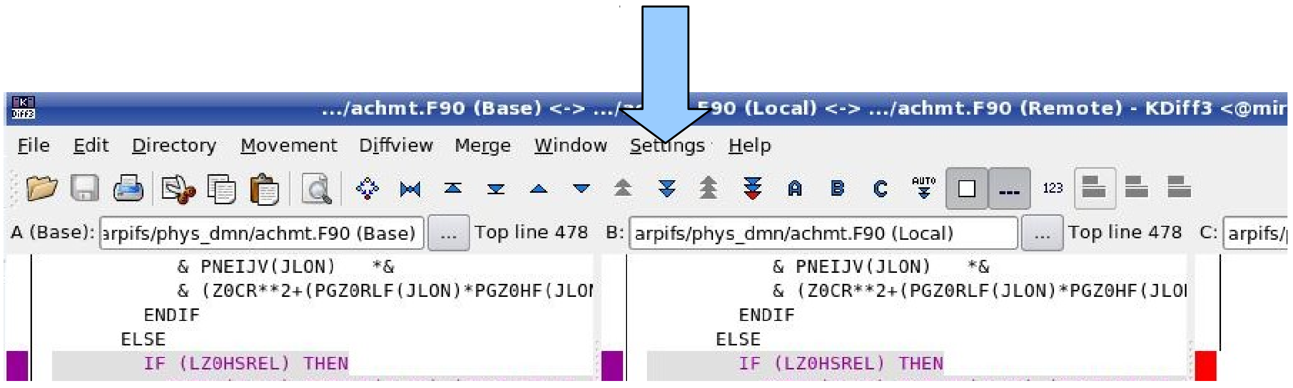
```
      IF (VZIUSTAR0 > 0) THEN
        PGZ0H(JLON)=PLSM(JLON)*PGZ0H(JLON)&
         & +(1.0_JPRB-PLSM(JLON))*PGZ0HF(JLON)
      ELSE
        PGZ0H(JLON)=MIN(PGZ0(JLON),PLSM(JLON)*PG
         & +(1.0_JPRB-PLSM(JLON))*PGZ0(JLON)*RZH
```

Output: arpifs/phys_dmn/achmt.F90 Encoding for saving: Codec from C: UTF-8

```
C <No src line>
? <Merge Conflict>
C <No src line>
C       PGZ0H(JLON                         )*PLSM(JLON)&
? <Merge Conflict>
C     ENDIF
C <No src line>
    ELSE
      PGZ0H(JLON)=PGZ0HF(JLON)
    ENDIF

 ! PGZ0H = ZGZ0 OVER SEA UNLESS REQUIRED IN NAMPHY1 (CLIMATE RUNS)

     IF (VZIUSTAR0 > 0) THEN
       PGZ0H(JLON)=PLSM(JLON)*PGZ0H(JLON)&
        & +(1.0_JPRB-PLSM(JLON))*PGZ0HF(JLON)
     ELSE
       PGZ0H(JLON)=MIN(PGZ0(JLON),PLSM(JLON)*PGZ0H(JLON)&
        & +(1.0_JPRB-PLSM(JLON))*PGZ0(JLON)*RZHZ0M)
     ENDIF

  ELSE
```

Select Line(s) From A Ctrl+1
Select Line(s) From B Ctrl+2
Select Line(s) From C Ctrl+3

We then move to the next conflict by clicking on the menu-button having the two arrows downward:



.../achmt.F90 (Base) <-> .../achmt.F90 (Local) <-> .../achmt.F90 (Remote) - KDiff3 <@mir

A (Base): arpifs/phys_dmn/achmt.F90 (Base) Top line 478 B: arpifs/phys_dmn/achmt.F90 (Local) Top line 478 C: arpifs/

```
            & PNEIJV(JLON)    *&
            & (Z0CR**2+(PGZ0RLF(JLON)*PGZ0HF(JLO
         ENDIF
       ELSE
         IF (LZ0HSREL) THEN
```

```
            & PNEIJV(JLON)    *&
            & (Z0CR**2+(PGZ0RLF(JLON)*PGZ0HF(JLO
         ENDIF
       ELSE
         IF (LZ0HSREL) THEN
```

and so on for all unsolved conflicts. Once the down-arrows button appears in grey, all conflicts are considered as solved. The result is saved and we can leave ***kdiff3***:

CAUTION:
The example above is mostly to illustrate the basic work with ***git_merge*** and ***kdiff3***. Therefore, the actions suggested for solving the manual conflicts are extremely simple (to simply choose “C” for instance). In practice, a number of conflicts will require a careful code analysis, and possibly some thorough scientific understanding and help by colleagues. We strongly suggest that large or difficult merges are considered as a collaborative effort, not a local one-person activity !

We then can proceed to the next file. Note that ***kdiff3*** is an advanced editor that is able to solve various conflicts by itself. If ***kdiff3*** finds files where it can solve all conflicts, then it will do so, and it will not open the graphics interface at all. Don't be therefore perturbed if not all presumably manually-to-be-merged files actually are proposed for a visual merge by ***kdiff3***.

Once all manual merges are completed, you should get the following message on screen:

```
[stef_CY36T1_merge 8d2ebfe] Merge with origin/gco_CY36T1_op1
```

At this stage, ***git_view*** should return the following:

```
% git_view
8d2ebfe35ca6e5651eefd3006251ba5a28d930fa
CY36T1_op1.18
CY36T1_bf.09
CY36T1
```

The ugly-looking 41 character string above is the commit-hash. This commit-hash is associated with the commit that has been performed behind the scene, at the end of the set of manual merges. If your changes are then posted/registered by ***git_post***, ***git_view*** should return:

```
% git_post
stef_CY36T1_merge/1
% git_view
stef_CY36T1_merge/1
CY36T1_op1.18
CY36T1_bf.09
CY36T1
```

which tells you that the meta-data server has associated a specific version number to your current branch, corresponding to your merged version. This numbering provides a much clearer visibility than the hash.

<u>How to actually see what has been changed during the merge ?</u>:

In general, a merge operation results in a rather big volume of modifications here and there. Especially, many of them have occurred with your explicit notice ! To get a comprehensive view of which files actually have just been changed in your branch "merge", ***git_list*** will prove most useful (as was ***cc_list*** for cc):

```
% git_list
deleted mpa/turb/internals/updraft_sope.f90
deleted arpifs/parallel/diwrgrid.F90
deleted arpifs/parallel/disgrid.F90
deleted arpifs/module/yomarar.F90
added arpifs/c9xx/csstbld.F90
added arpifs/dia/wrgrida.F90
added arpifs/function/fcgeneralized_gamma.h
added arpifs/module/disgrid_mod.F90
added arpifs/module/diwrgrid_mod.F90
[...]
modified utilities/ctpini/programs/inversion_master.F90
modified utilities/gobptout/prochien.F
modified utilities/gobptout/procor1.F
modified utilities/gobptout/progeom.F
modified utilities/pinuts/module/const_standart_mod.F90
modified utilities/pinuts/module/editfield_prg_mod.F90
modified utilities/pinuts/module/egg_tools_mod.F90
```

The only difference with ***cc_list*** is that ***git_list*** also provides an information about the nature of the change: suppression, renaming, add, modification. It is also possible to check what was modified on any specific branch using the options ***-u, -r and -b***, in a similar manner as with ***cc_list***.

Conclusion:

You now do have all the basic knowledge to get started and work with the GIT-tools. The remaining part of this note gives a more detailed inventory of all available commands.

Appendix A: list of GIT-tools commands

In this Table, ct = cleartool to indicate Clearcase counterparts.

Command	Description	Equivalent ClearCase
git_add_branch	To register a branch that was not originally created with ***git_branch***[4]	
git_branch	To create a branch, or to open an existing one[5]	***cc_getpack*** ***cc_getview***
git_commit	To perform a commit (to commit)	
git_diff	To visually display the differences between 2 (or max 3) versions of an element from the repository[6]	***cc_diff***
git_edit	To edit an element of the repository[7]	***cc_edit***
git_fetch	To fetch a remote (or origin) repository, and update (all or) given local branches if required[8]	
git_finger	To search the identification of a user	
git_help	Manual (on-line) help for any GIT-tools command	***cc_help***
git_history	To retrieve the history of changes of an element of the repository	***ct lshistory***
git_info	To get the GIT description of an element of the repository, of a branch, a tag or a commit	***ct describe***
git_list	To list all modifications applied in a branch[9]	***cc_list***
git_login	To enable authentication on the local platform	
git_logout	To disable authentication on the local platform	
git_merge	To merge a remote branch on the current one[10]	***cc_merge***

4 We recommend to ask help/supervision by GCO before using this command !

5 Option ***-U*** takes the GIT user identification, not the System identification (eg. 'gco' not 'marp001') ! Please also keep in mind that 'public branches' in the sense of cc do not hold for GIT: any user may open and change branches of any other user. However, submit changes made on a non-proprietary branch to the central SCR (by ***git_post***) is not possible.

6 The editor tool that will be launched by ***git_diff*** is platform-dependent (depending on the default editor setting). A user may choose his custom editor for ***git_diff*** by setting the environmental variable GIT_DIFF_EDITOR to the complete path+name (eg. ***/usr/X11R6/bin/gvimdiff***). Some editors run in background mode by default, which is unsuitable for ***git_diff***. If so, check for the appropriate command line option to disable background mode (eg. ***export GIT_DIFF_EDITOR = /usr/X11R6/bin/gvimdiff -f*** .

7 Same note as for ***git_diff*** (see above)

8 When a branch of the remote SCR already exists locally, then the local branch will be updated via a merge operation. If no local branch exists, then it is created and updated. Note that merges should be automatic, otherwise the local update of that branch will fail. We expect that most ***git_fetch*** operations will be applied with no command line option.

9 Option ***-u*** requires the GIT user identification, not the user's System identification ! ***git_list*** only will work once the branch of interest has been updated in GCO's meta-data base. Indeed, ***git_list*** requires to know the commit(-hash) associated to a branch, an information which is not available by a standard GIT command.

10 (1) unlike the native ***git merge*** command, only one branch at a time can be merged with ***git_merge*** (2) the default (and rather recommended) working mode for ***git merge*** is 'resolve' – the ***-s*** option in ***git_merge*** allows to switch to another mode of ***git merge*** but we recommend the highest caution to users when changing this mode ! (3) option ***-u*** requires the GIT user identification, not the UNIX System identification (4) instead of using the standard options ***-u/-r/-b***, it is perfectly possible to use option ***-v*** with the name of a branch+version (eg. 'gco_CY37T1_op1/1') or a commit(-hash) or a tag (advanced use) (5) the list of all merge operations that need to

		cc_fmerge
git_mod	To list all modifications performed on an element of the repository, based on the very same release as the current branch	***cc_md***
git_password	To change the user's password	
git_post	To submit (/ ask for) an update of the central repository and the meta-data server, with all modifications performed in the current branch[11]	
git_public	To enable public access to a branch, for all or for specific users	***cc_public***
git_start	To initialize the GIT-tools environment	
git_user	To obtain information on any user	
git_view	To obtain the rules of view for the current branch[12]	***cc_view***

be performed can be saved into an output file using option ***-o*** (6) in the case when the list of merge operations has been saved on file beforehand, the merge itself can be stopped (for instance with CTRL-C) and resumed later using option ***-f*** (7) since GIT does not provide a standard graphical interface to help for manual merge, a user-specified graphical display is needed. GCO recommends ***kdiff3*** which is available on merou (see also body text for details)

11 ***git_post*** won't work if the user is not authenticated on the current platform, if he doesn't have 'write' authorization on the central repository, if he is not the owner/creator of the current branch

12 Unlike in Clearcase, it is not possible to insert any additional user's branch within the rules of view of a current branch

Appendix B: few native GIT commands

Here are listed a few native, standard GIT commands, that can prove to be useful and help making things easier ...

- clone a repository:

*% git **clone** URL-towards-distant-repository*

- clone a repository and change its name with respect to the origin repository:

*% git **clone** URL-towards-distant-repository name-of-local-repository*

- get the list of all local branches:

*% git **branch***

- get the list of all remote (distant) branches:

*% git **branch** -r*

- create a new branch on top of the current one, without however changing branch:

*% git **branch** name-of-new-branch*

- create a new branch on top of any reference (a commit, a tag, a name of branch [local or distant], etc...), without however changing branch:

*% git **branch** name-of-new-branch reference*

- open an existing branch:

*% git **checkout** name-of-branch*

- create and open a new branch on top of the current one:

*% git **checkout** -b name-of-new-branch*

- create and open a new branch on top of any reference (a commit, a tag, a name of branch [local or distant], etc...):

*% git **checkout** -b name-of-new-branch reference*

- get status information about the current situation in a branch, since the last commit was done (modified / added / renamed / deleted files, files not yet added to the index etc.):

*% git **status***

- update the index table of the files present in the repository (Note: this action has to be done from the "top directory" of the repository):

*% git **add** .*

- Perform a commit with a short message:

```
% git commit -m 'message-for-commit'
```

- perform a commit and invoke a text editor for typing an extended message:

```
% git commit
```

- perform a commit, and add as message text contained in a file:

```
% git commit -F name-of-file
```

- change the message of the last commit, or even change the content of the last commit with modifications done since then (Note: changing the content is absolutely proscribed if the commit already has been posted with ***git_post*** !):

```
% git commit --amend
```

- return back to the previous commit, while preserving the modifications done since then. These modifications are then ready to be re-committed (Note: this action is absolutely proscribed if the commit already has been posted with ***git_post*** !):

```
% git reset —soft HEAD^
```

- return back to the previous commit, without preserving the modifications done since then (Note: this action is absolutely proscribed if the commit already has been posted with ***git_post*** !):

```
% git reset —hard HEAD^
```

- merge a branch into the current one:

```
% git merge name-of-branch
```

- “piling up” commits in the current branch:

```
% git log
```

- “piling up” commits in a reference (a branch, a tag, a file, etc...) :

```
% git log reference
```

- cleaning up files that have become useless, from the local repository, and optimization/compression of the data (Note: to be done regularly !):

```
% git gc
```

- add a distant repository, and update the data from that repository:

```
% git remote add alias-of-distant-repository URL-towards-distant-repository
% git fetch  alias-of-distant-repository
```

- suppress one or more files (or directories) (Note: syntax is similar to the Unix command ***rm***):

```
% git rm [-r] name-of-file [name-of-file ...]
```

- rename/move one or more files (or directories) (Note: syntax is similar to the Unix command ***mv***):

*% git **mv** source [source ...] target*