LA BOITE A OUTILS Git-GCO

Stéphane MARTINEZ – Version 1 - 2012/09/10

Préambule

Ce document ne prétend pas être un manuel utilisateur de Git ! Le but est ici de présenter les différents concepts et fonctionnalités de base de Git, avec ClearCase comme point de comparaison.

On reviendra également sur ce qui a motivé le développement d'un serveur de méta-données Git-GCO et d'une boîte à outils utilisateur. Un petit tutoriel simple permettra de se familiariser avec cette boîte à outils.

Enfin, on trouvera en annexe la description complète de l'ensemble des commandes utilisateur de la boîte à outils, ainsi qu'un récapitulatif de certaines commandes Git natives très utiles.

Table des matières

Les concepts et fonctionnalités de Git →

Les dépôts et les branches →
Les modifications locales sont... locales! →
Eléments tracés par Git →
La notion de *commit* →
La notion de *tag* →
Le merge de branches →
Autres fonctionnalités →

Le serveur de méta-données Git-GCO →

Pourquoi ce serveur ? →

La boîte à outils Git-Gco →

Pourquoi une telle boîte à outils ? →
Sur quelles machines est -elle installée ? →

Tutoriel →

Avant toute chose, exécuter *git_start* →
Accès au dépôt Git – Création d'une branche →
Modifications de fichiers sur la branche créée →
Suppressions – Renommages – Ajouts →
Enregistrement des modifications →
Mise à jour du serveur de méta-données et du dépôt central →
Merge de branches →
Au fait, qu'ai-je modifié dans ma branche ? →

ANNEXE: Les commandes utilisateurs de la boîte à outils →

ANNEXE: Quelques commandes Git natives →

Les concepts et fonctionnalités de Git

Les dépôts et les branches:

La notion de dépôt n'est pas une inconnue pour les utilisateurs ClearCase. Sur *merou* , l'ensemble des utilisateurs travaillent sur un même dépôt central situé dans le répertoire *~marp001/dev* . L'accès au code source se fait alors en ouvrant une vue, et la sélection de ce qu'on souhaite voir se fait via des règles de vue, dans lesquelles apparaîtront la (les) branche(s) requises. Cette sélection est complètement transparente pour les utilisateurs ClearCase, à condition d'utiliser les scripts *cc_getpack* ou *cc_getview* .

Sous Git, on va procéder de manière différente. Les utilisateurs ne vont pas travailler directement sur de dépôt central, mais sur un « clone » de ce dépôt situé n'importe où sous l'arborescence locale de l'utilisateur, sur une machine pouvant être différente que celle sur laquelle est installée le dépôt central (à condition que Git soit installé, bien évidemment).

Il n'y a pas à proprement parler de notion de vue sous Git, l'accès au code source est immédiat, il suffit de se rendre dans le répertoire créé après avoir cloné le dépôt central. Par conséquent, la notion de vue publique n'a plus aucun sens...

La notion de branche est quant à elle bien présente avec Git, on peut créer et modifier n'importe quelle branche dans son propre dépôt, même si on n'est pas le créateur de la branche!

NB: même s'il est très simple de créer une branche sous Git, un outil ***git_branch*** a été développé, très proche d'un point de vue fonctionnel de *cc_getpack* .

Les modifications locales sont... locales!

Il faut bien avoir à l'esprit que les modifications de sources que va faire un utilisateur dans son propre dépôt sont locales, et qu'elles ne toucheront pas au dépôt central! Les mises à jour du dépôt central se font soit en faisant un « push » de sa branche vers le dépôt central (à condition que l'administrateur du dépôt central le permette...), soit en faisant un « pull » de cette branche au niveau du dépôt central (via l'administrateur de ce dépôt).

En corollaire, il n'est pas forcément aisé de savoir de ce que font les autres utilisateurs, du moins tant que le dépôt central n'est pas mis à jour.

Eléments tracés par Git:

Git ne trace que les fichiers, pas les répertoires. En particulier, un répertoire vide ne sera pas tracé. Le « traçage » des fichiers sous Git se fait de façon la plus simple qui soit, par le le nom complet du fichier (ex: *arp/adiab/cpg.F90*).

NB: pour les suppressions et les renommages/déplacements de fichiers ou de

répertoires, aucun outil n'a été développé à GCO, pour la simple et bonne raison qu'on peut le faire de façon simple avec des commandes Git natives.

La notion de *commit* :

La notion de *commit* est déjà connue des utilisateurs subversion , on l'utilise plus ou moins sous ClearCase. Lorsqu'on est dans une vue ClearCase, on utilise usuellement le script *cc_edit* pour modifier un élément. Ce script réalise en fait 3 actions:

- le *checkout* de l'élément: extraction de l'élément de la base ClearCase dans le but de le rendre modifiable (NB: les éléments sont par défaut en lecture seule);
- édition de l'élément ;
- le *checkin* de l'élement: mise à jour de l'élément de la base et création d'une nouvelle version (NB: l'élément est de nouveau en lecture seule).

Sous Git, c'est beaucoup plus simple: lorsqu'on est dans une branche tous les éléments sont modifiables instantanément, sans avoir à faire de *checkout*, en utilisant son éditeur de texte préféré! On va donc pouvoir ajouter/supprimer/renommer/modifier tous les fichiers qu'on souhaite, puis faire un *commit* pour enregistrer l'ensemble des modifications. Un *commit* est donc global, alors que sous ClearCase le *checkin* se fait élément par élément.

Un *commit* est associé à un code hexadécimal unique de 41 caractères appelé *commit-hash* .

En corollaire, le versionnement des éléments se fait élément par élément sous ClearCase, de façon globale sous Git .

NB:

1) Par commodité, 2 outils ont été développés pour l'édition de fichiers ou pour faire un commit, à savoir ***git_edit*** (fonctionnant de manière identique à *cc_edit*) et ***git_commit*** .
2) Si un (ou plusieurs) fichiers doit (doivent) être renommé(s) puis modifié(s), il est conseillé de procéder aux renommages/déplacements dans un premier temps, de faire un *commit*, puis de procéder aux modifications.

La notion de *tag*:

La notion de *tag* est déjà connue sous ClearCase: ce sont les labels (ex: CY37T1_op1.08) et les releases (ex: CY38). Sous Git, un *tag* se comporte comme une sorte d'alias vers un *commit* .

Le merge de branches:

La notion de merge de branches est déjà connue sous ClearCase: le merge de branches consiste à fusionner le contenu d'une branche donnée sur la branche courante. Git est capable de résoudre des conflits de façon plus évoluée que ClearCase, les modifications « othogonales » seront ainsi traitées automatiquement, alors qu'elles font l'objets de merges manuels sous ClearCase.

Contrairement à ClearCase, aucune interface graphique n'est disponible pour faciliter

les merges manuels. Cela dit, Git est capable d'utiliser sous la forme de plugins des outils issus du monde libre ou non pour disposer de cette fonctionnalité.

NB: Par commodité, l'outil ***git_merge*** a été créé, fonctionnant de façon analogue à *cc_merge* .

Autres fonctionnalités:

- Tout comme sous ClearCase, il est possible de connaître l'historique des modifications faites sur un élément, et de faire des « diffs » sur des différentes versions. Comme dans le cas du merge, le « diff » s'appuie sur des outils externes, qui ne permettent malheureusement pas de comparer plus de 3 versions. Deux outils ont été développés à GCO pour simplifier la tâche: ***git_history*** et ***git_diff*** .

- Sous ClearCase, il est relativement facile de savoir ce qui a été modifié dans une branche. Par contre, c'est nettement mois évident sous Git. Il est en effet nécessaire de connaître la version du code source sur laquelle a été construite la branche: aucune commande Git ne permet d'avoir cette information... Des développements ont réalisés à GCO pour palier à ce manque (cf. partie suivante), et une commande ***git_list*** (analogue à *cc_list*) a été développée en ce sens.

Le serveur de méta-données Git-GCO

Pourquoi ce serveur ?

- Comme on l'a vu dans la partie précédente, les commandes natives Git ne permettent – malheureusement – pas d'accéder à certaines informations utiles, comme par exemple la version du code source sur laquelle est basée une branche. Certes, on peut toujours faire un *git log* dans la branche pour voir l'empilage des modifications successives, et ainsi « deviner » la version de base par le biais de son *commit-hash* ... Sauf qu'on souhaiterait avoir cette information automatiquement et simplement!

- La commande native *git log* avec un nom d'élément en argument permet certes d'obtenir l'historique des modifications de cet élément, ainsi que des informations sur chaque *commit* : auteur, date, *commit-hash*, documentation associée. Par contre, pas d'information concernant la branche dans laquelle a été fait ce *commit* ...

- Un *commit-hash* n'étant franchement pas très lisible, on souhaiterait voir à la place une version sur une branche, comme c'est le cas avec ClearCase (ex: *marp001_CY38_t1/4)* .

- Comme on l'a vu plus haut, un utilisateur peut modifier dans son dépôt local n'importe quelle branche, même s'il ne l'a pas créée. On souhaiterait donc introduire un mécanisme d'identification de l'utilisateur permettant de contrôler l'identité de l'utilisateur, et de savoir s'il a effectivement le droit de modifier cette branche.

- On souhaiterait néanmoins pouvoir autoriser certains utilisateurs (pourquoi pas tous?) à modifier une branche dont on est le propriétaire, cela peut être très pratique dans certains cas.

Les points ci-dessus ont motivé le développement d'un serveur de méta-données: d'une part pour palier à certains manques réels de Git par rapport à ClearCase, d'autre part pour des raisons plus « esthétiques » et pratiques.

La boîte à outils Git-Gco

Pourquoi une telle boîte à outils ?

Tout comme ce qui a été avec ClearCase (commandes *cc_**), un certains nombres d'outils écrits en Perl ont été développés pour faciliter l'usage de Git . Ces outils encapsulent à la fois l'usage des commandes natives Git et l'accès au serveur de méta-données Git-GCO .

Pour un usage « standard », il n'est pas forcément utile de connaître les commandes natives Git, sauf quelques exceptions que nous verrons par la suite.

La liste et la description complète des ces commandes est disponible en annexe.

Sur quelles machines est-elle installée ?

Pour le moment, la boîte à outils Git-GCO est installées sur les machines suivantes:

- *merou* ,
- *yuki* & *kumo* ,
- *tori* ,
- *serran* & *mostelle* ,
- *mirage* (NB: machine de COMPAS).

Il est prévu de l'installer également sur *yukisc2* & *kumosc2*, une fois que Git y sera installé.

Tutoriel

Avant toute chose, exécuter *git_start*

Avant d'utiliser la boîte à outils Git-GCO sur une machine pour la première fois, il est nécessaire de lancer la commande ***git_start*** (dans le répertoire *~marp001/git-install/client/default/bin*) pour vérifier/initialiser son environnement.

Lors de l'exécution de ce script, il vous sera demandé si vous disposez déjà d'un compte utilisateur Git. Si vous en avez un, répondez 'y' pour vous authentifier, sinon répondez 'n' (ou ENTER directement), l'exécution du script se poursuivra normalement.

NB: dans le cas où vous ne disposez pas d'un compte utilisateur Git, n'hésitez pas à demander l'ouverture d'un compte à GCO !

A la fin de l'exécution de ***git_start*** , vous devriez obtenir un « output » ressemblant fortement à ceci:

```
% ~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
```

Dans cet exemple, l'utilisateur *stef* part d'un environnement Git totalement vierge. Il lui reste donc à définir dans son *.profile/.bash_profile/etc...* les variables suivantes:

- GIT_INSTALL : répertoire dans lequel est installé la boîte à outils Git-GCO ;
- GIT_ROOTPACK : emplacement du dépôt central ;
- GIT_HOMEPACK : emplacement du dépôt local de l'utilisateur ;
- GIT_WORKDIR : répertoire (temporaire) de travail de l'utilisateur .

Les variables à ajouter dans le PATH correspondent à l'emplacement des commandes natives Git, et des commandes utilisateurs Git-GCO .

Dans le cas où tout s'est déroulé correctement (NB: demandez de l'aide à GCO dans le cas contraire), et une fois que votre environnement est initialisé, les choses sérieuses vont pouvoir commencer... Au passage, **tout ce qui suit suppose que vous êtes un utilisateur authentifié** !!

Accès au dépôt Git – Création d'une branche:

En plus d'intialiser l'environnement, la commande ***git_start*** réalise en plus la copie locale du dépôt central, en clair le *clonage* .

Le dépôt local Git contenant le code source ARPEGE/ALADIN/AROME se nomme *arpifs* , et se trouve dans le répertoire *$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
```

Comme vous vous en apercevez, la majorité des projets a changé de nom par rapport à ClearCase, pour plus de lisibilité: *arp* devient *arpifs* , *ald* devient *aladin* , *xla* devient *algor* , *bip* devient *biper* , *bla* devient *blacklist* , *tfl* devient *trans* , *tal* devient *etrans* , *xrd* devient *ifsaux* , *obt* devient *obstat* , *sat* devient *satrad* , *sct* devient *scat* , *scr* devient *scripts* , *sur* devient *surf* , *uti* devient *utilities* .
La version de sources que vous avez sous les yeux correspond tout simplement à la branche courante du dépôt central au moment de l'exécution de ***git_start*** . Pour information, il est très simple de la connaître en utilisant la commande ***git_view*** :

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

Cette commande nous donne l'empilage des versions successives, de façon analogue à ce que donne la commande ClearCase *cc_view* . A ce jour (2012/09/06), l'intégralité des cycles (du cycle CY18 au cycle CY38T1) est disponible, ainsi qu'un certain nombres de branches GCO et de branches utilisateurs (sur le cycle CY38T1).

Pour se créer une branche sur une version de source donnée, l'outil ***git_branch*** a été développé à GCO, et fonctionne de manière analogue à la commande ClearCase *cc_getpack* .
Par exemple, pour se créer une branche *test* sur la version 05 de la chaîne en double CY37T1_op1 :

```
% 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
 >
```

Vous remarquerez que le nom des branches commence par le nom d'utilisateur Git, et non par le nom d'utilisateur système, qui peut être différent selon la machine (ex: *mrpm602* sur *merou* et *khatib* sur *yuki*).

Modifications de fichiers sur la branche créée:

En théorie, pour modifier un fichier existant, il peut sembler inutile de créer un outil tel que *cc_edit* , vu que tous les fichiers présents dans le dépôt sont modifiables sans avoir a faire de *checkout/checkin* : il suffit juste de les éditer et de les modifier dans son éditeur de texte préféré!

Toutefois, l'outil ClearCase *cc_edit* est pourvu en plus de deux fonctionnalités très pratiques:

1) on peut éditer un fichier même s'il n'est pas dans le répertoire local, une recherche est faite dans toute la vue pour le retrouver (NB: on peut donc faire *cc_edit -f cpg.F90* n'importe où) ;
2) on peut savoir au lancement de *cc_edit* si ce fichier a été modifié sur d'autres branches basées sur la même release.

C'est dans ce but que l'outil ***git_edit*** a été développé, il intègre les 2 mêmes fonctionnalités décrites ci-dessus.

Par exemple, si on édite apl_arome.*F90* à partir du répertoire *$GIT_HOMEPACK/arpifs* , on obtient ceci :

```
% 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
```

On fait alors les modifications qu'on souhaite dans *apl_arome.F90* , on les sauve, on sort de son éditeur, et on arrive au final à une demande de confirmation:

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

A ce stade, on est libre d'accepter ses modifications, ou de changer d'avis et de répondre non! Vous pouvez donc modifier de la sorte ce que vous voulez.

Suppressions – Renommages – Ajouts:

- Pour la suppression de fichiers/répertoires, aucun *wrapper* n'est nécessaire vu la simplicité de la chose avec Git :

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

On utilisera l'option ***-r*** pour effacer un (ou plusieurs) répertoire(s) entier(s).

- Pour renommer des fichiers, des répertoires, ou encore pour déplacer des fichiers/répertoires, là encore nul besoin de *wrapper* avec Git :

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

ATTENTION!!

On peut utiliser la commande *rm* du système pour effacer des fichiers/répertoires, ça ne posera pas de problèmes à Git . Par contre, l'utilisation de la commande *mv* du système ne garantit pas la possibilité de suivre l'historique d'un élément après son renommage...

- Pour l'ajout de nouveaux fichiers/répertoires, là on fait ce que l'on veut: *touch*, *mkdir*, *cp*, *vim*, etc...

Un fichier est « tracé » par Git en fonction de son nom complet (i.e. *arpifs/phys_dmn/apl_arome.F90*). Si on a effacé un fichier dans une branche ancienne et qu'on souhaite le recréer dans la branche actuelle, nul besoin d'utiliser un équivalent de *cc_add* : il suffit de le créer à nouveau comme indiqué ci-dessus, et on récupère l'historique du fichier précédemment effacé.

Enregistrement des modifications:

Pour prendre en compte les modifications faites jusqu'ici, 2 opérations sont nécessaires lorsqu'on utilise les commandes natives Git :

1) utilisation de la commande *git add* pour mettre à jour l'index des fichiers du dépôt ;
2) utilisation de la commande *git commit* pour enregistrer les modifications .

La commande utilisateur ***git_commit*** réalise ces 2 opérations en une fois. Chaque commit doit être accompagné d'un message pour documenter ses modifications.

- Si le message est court (une ligne), on pourra utiliser l'option ***-m*** de ***git_commit*** avec comme argument le message de commit.

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

- Si le message est long (un ou plusieurs paragraphes), on utilisera l'option ***-f*** de ***git_commit*** avec comme argument un fichier contenant ce message (et situé de préférence hors du dépôt!).

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

Mise à jour du serveur de méta-données et du dépôt central:

Jusqu'ici, comme il a été dit ici , les modifications faites sont locales: on ne touche pas au dépôt central. Lorsqu'on est sûr de ses modifications, on peut alors demander la mise à jour du serveur de méta-données et du dépôt central via la commande ***git_post*** .

```
% git_post
stef_CY37T1_test/1
```

Si tout se passe bien, au bout de quelques secondes, la commande ***git_post*** renvoie la nouvelle version sur la branche courante. Si on exécute ***git_view*** , on obtient quelque chose qui doit fortement ressembler à ceci:

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

Il n'est nullement besoin de faire un ***git_post*** après chaque commit, vu que tous les commits réalisés depuis la dernière exécution de ***git_post*** seront enregistrés sur le serveur.

Merge de branches:

Un merge de branche peut se faire de manière on ne peut plus simple via la commande native *git merge nom-de-la-branche* . Néanmoins, l'utilisation de *git merge* ne permet pas de savoir à l'avance ce qu'on va avoir à merger, comme c'est le cas avec ClearCase . C'est en ce sens que la commande ***git_merge*** a été développée, fonctionnant de manière analogue à la commande ClearCase *cc_merge* . Avant de lancer le merge, on aura des informations sur ce qu'il faut merger (suppressions/renommages/ajouts/modifications), et sur le type de merge (automatique/manuel).

Pour la réalisation des merges manuels, Git ne dispose malheureusement pas d'une interface graphique comme c'est le cas avec ClearCase. Cela dit, il est possible d'utiliser des outils externes comme *vimdiff/gvimdiff*, *tkdiff*, *meld,* ou *kdiff3* , avec une nette préférence pour ce dernier vu que l'interface proposée pour le merge manuel ressemble beaucoup à celle utilisée avec ClearCase.

ATTENTION!!
kdiff3 n'étant disponible que sur *merou* , nous vous conseillons vivement de poursuivre ce tutoriel sur *merou* !

Pour tester ***git_merge*** en grandeur nature, on va d'abord commencer par se créer une nouvelle branche nommée *merge* basée sur le cycle 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
 >
```

Nous allons merger dans notre branche actuelle l'ensemble des bugfixes du cycle CY36T1, donc la branche *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 peut déjà remarquer deux choses:

1) La rapidité d'exécution! Le merge (automatique) des 146 routines de la bugfix du cycle CY36T1 est quasi-instantané.

2) La deuxième ligne de l' « output » de la commande ***git_merge*** mentionne ceci:
*Merging from branch **origin/gco_CY36T1_bf***
Que vient donc faire ce ***origin*** dans l'histoire ? En fait, la branche *gco_CY36T1_bf* n'a pas été créée localement, elle a été créée dans de dépôt qu'on a cloné (i.e. dans notre cas le dépôt central): c'est ce qu'on appelle une *branche distante* . Si on avait utilisé la commande native:
git merge gco_CY36T1_bf , nous aurions obtenu le message d'erreur suivant:
fatal: gco_CY36T1_bf - not something we can merge

L'utilisation de ***git_merge*** permet donc de ne pas se poser de question au sujet de la nature d'une branche. A noter que c'est aussi valable pour ***git_branch*** !

A ce stade, si on lance la commande git_view , on obtient ceci:

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

C'est parfaitement normal, vu qu'on part du cycle CY36T1 sur lequel on merge toutes les bugfixes!

On va maintenant merger la branche gco_CY36T1_op1:

```
% 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
```

On peut être surpris de voir qu'il y aura des merges manuels, mais c'est normal: la branche *gco_CY36T1_op1* est basée sur la bugfix n°06 du cycle CY36T1, alors qu'il y a 9 bugfixes pour ce cycle...

On va donc dans un premier temps effectuer les merges automatiques:

```
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
```

Quelques commentaires:

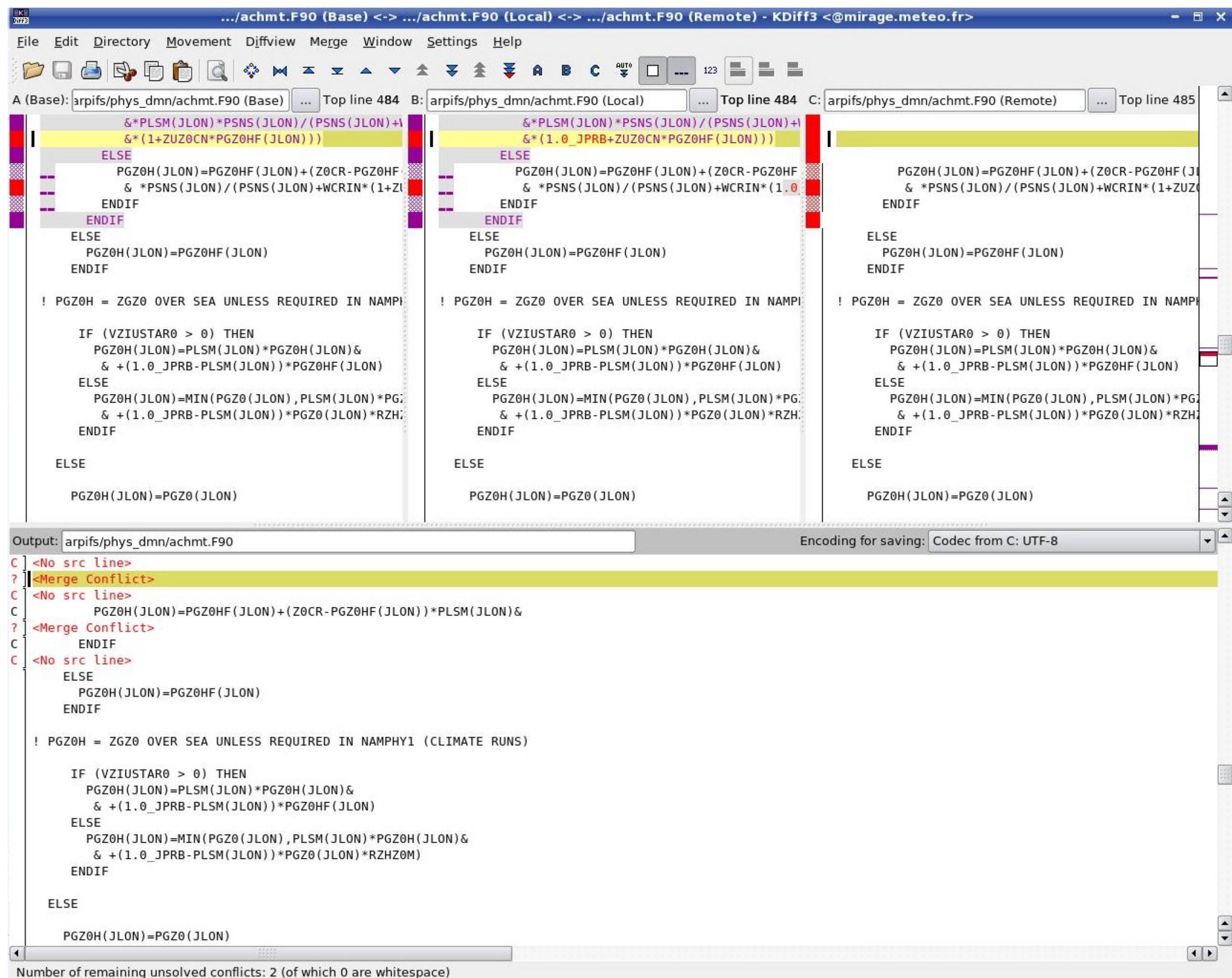
1) Les merges dits *fast-forward* (où la branche gco_CY36T1_op1 est seule contributrice ne sont pas mentionnés) ;
2) Les merges apparaissant en *Auto-merging* sans erreur sont des merges en principe manuels, mais néanmoins résolue par Git (modifications orthogonales par exemple) ;
3) Les merges apparaissant en A*uto-merging* avec erreur sont des vrais merges manuels, qu'il va donc falloir résoudre!

Comme c'est le cas avec *cc_merge* , les merges manuels se font routine par routine avec demande de 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):
```

En appuyant sur *RETURN* , l'interface de kdiff3 apparaît à l'écran:



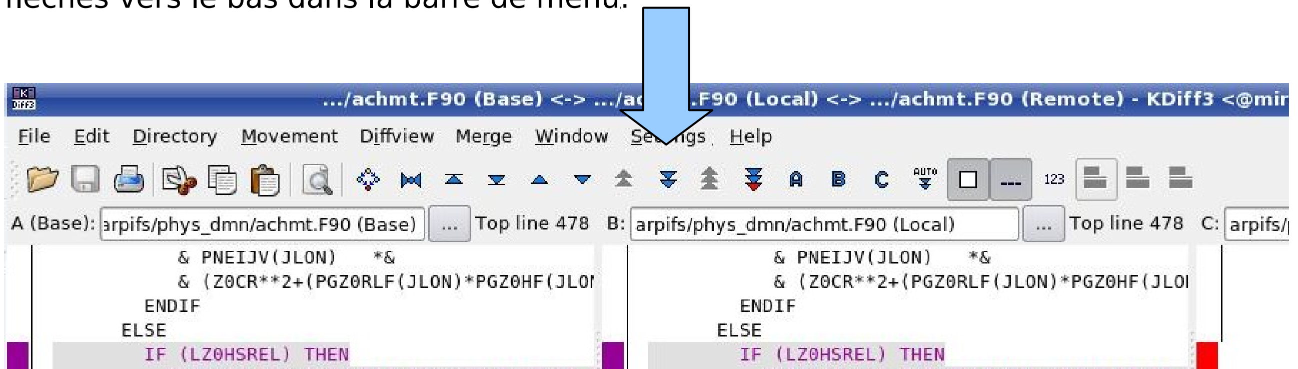
Dans le fenêtre du haut on a successivement la version de base commune à la version actuelle et la version à merger (Base), la version actuelle (Local) , la version à merger (Remote). Dans la fenêtre du bas, on a le résultat du merge.

Les conflits automatiques sont résolus par *kdiff3* , seuls les vrais conflits subsistent et sont mentionnés dans la fenêtre du bas en *<Merge Conflict>* . On est placé automatiquement au niveau du premier conflit non résolu, les lignes à comparer sont surlignées en jaune.

Pour le conflit en cours (le premier), il suffit de se se placer dans la fenêtre du bas sur la ligne surlignée en jaune, de faire un clic droit, et de choisir l'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

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

C  <No src line>
?  <Merge Conflict>
C  <No src line>        A Select Line(s) From A   Ctrl+1
C       PGZ0H(JLON      B Select Line(s) From B   Ctrl+2   *PLSM(JLON)&
?  <Merge Conflict>     C Select Line(s) From C   Ctrl+3
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
```

Ensuite, on passe au conflit suivant en appuyant sur le bouton comportant deux flèches vers le bas dans la barre de menu:



Et ainsi de suite pour tous les conflits non résolus!! Si le bouton est grisé, cela veut bien évidemment dire que tous les conflits sont résolus. Ensuite, on sauvegarde le résultat, et on sort de *kdiff3* .

ATTENTION!!
C'est ici un exemple simple de résolution de conflits! Certains merges se révèlent extrêmement délicats, et nécessitent pas mal de réflexion, voire même une aide extérieure...

Ensuite, on procédera de la même façon pour les routines suivantes... sauf que pour *arpifs/setup/sugrida.F90* et *mpa/turb/internals/compute_updraft.f90* , l'éditeur *kdiff3* ne prend même pas la peine de s'ouvrir et on passe à la routine suivante! L'explication est simple: *kdiff3* est un éditeur très avancé pour les tâches de merges, il est donc capable de régler certains conflits par lui-même.

Une fois passés les merges manuels, vous devriez avoir à l'écran quelque chose qui ressemble à:

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

A ce stade, la commande git_view renvoie désormais quelque chose dans ce genre:

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

La suite bizarroïde de 41 caractère n'est autre que le *commit-hash* du commit effectué sans que vous ne vous en aperceviez à la fin des merges manuels. Si vous postez maintenant vos modifications avec git_post , et que vous exécutez ensuite git_view , vous devriez voir à l'écran quelque chose comme:

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

Le serveur de méta-données a associé à votre commit une version sur la branche courante, pour plus de lisibilité.

Au fait, qu'ai-je modifié dans ma branche ?

A ce stade, vous avez fait énormément de modifications dans votre branche *merge*, sans vous en rendre forcément compte! Pour savoir ce qui a été modifié sur votre branche, l'outil ***git_list*** va s'avérer incontournable, tout comme l'est *cc_list* avec ClearCase .

```
% 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
```

La seule différence par rapport à *cc_list* , c'est qu'on a aussi des informations sur la nature de la modification d'un fichier: suppression, renommage, ajout, modification.
On peut également voir ce qui a été modifié sur n'importe quelle branche avec les options **-u**, **-r**, et **-b**, comme on le fait sous ClearCase avec *cc_list* .

Conclusion:

A ce stade, vous avez en principe toutes les bases pour bien débuter avec la boîte à outils Git-GCO . La partie suivante dresse un inventaire détaillé de l'ensemble des commandes utilisateur de la boîte à outils Git-GCO .

ANNEXE: Les commandes utilisateurs de la boîte à outils

NB: dans ce tableau, *ct* = *cleartool* pour les équivalents ClearCase .

Commande	Description	Equivalent ClearCase
git_add_branch	Enregistrement d'une branche qui n'a pas été créée avec git_branch	
git_branch	Création d'une branche, ou ouverture d'une branche existante	***cc_getpack cc_getview***
git_commit	Réalisation d'un commit	
git_diff	Visualisation des différences entre 2 ou 3 versions d'un élément du dépôt	***cc_diff***
git_edit	Edition d'un élément du dépôt	***cc_edit***
git_finger	Recherche d'un utilisateur	
git_help	Obtenir de l'aide en ligne sur une commande de la boîte à outils Git-GCO	***cc_help***
git_history	Historique des modifications d'un élément du dépôt	***ct lshistory***
git_info	Description d'un élément du dépôt, d'une branche, d'un tag, ou d'un commit	***ct describe***
git_list	Liste des modifications effectuées dans une branche	***cc_list***
git_login	Authentification sur la machine locale	
git_logout	Dés-authentification sur la machine locale	
git_merge	Merge d'une branche sur la branche courante	***cc_merge cc_fmerge***
git_mod	Liste des modifications faites sur un élément du dépôt, basées sur la même release que la branche courante	***cc_md***
git_password	Changement du mot de passe utilisateur	
git_post	Demande de mise à jour du dépôt central et du serveur de méta-données avec les modifications faites sur la branche courante	
git_public	Possibilité de rendre une branche publique, pour certains utilisateurs, ou pour tous	***cc_public***
git_start	Initialisation de son environnement Git-GCO	
git_user	Obtenir des informations sur un utilisateurs	
git_view	Règles de vue de la branche courante	***cc_view***

git_add_branch

Objet:	*Register current branch (use carefully!).*
Niveau:	*Avancé*
Usage:	*git_add_branch [commit-hash]*
Options:	*no*

Dans le cas où une branche n'a pas été créée avec la commande ***git_branch***, la commande ***git_add_branch*** permet d'intégrer les méta-données relatives à cette branche dans la base de données GCO . Tant qu'une branche n'est pas enregistrée de la sorte, il n'est pas possible de poster ses modifications vers le dépôt central via la commande ***git_post*** .

L'argument *commit-hash* est facultatif: s'il est absent, le commit le plus récent est pris en compte. Ce commit correspond à la version de base sur laquelle a été créée la branche courante.

Pour information, la connaissance de ce commit « de base » est cruciale pour le fonctionnement de la commande ***git_list*** . Les commandes Git standard ne permettent pas d'obtenir cette information.

ATTENTION!!

Il est préférable de demander l'aide de GCO avant d'exécuter cette commande!

git_branch

Objet:	*Create and checkout a new branch, or checkout an existing branch.*
Niveau:	*Standard*
Usage:	*git_branch [-U user-name] [-r release] [-b branch] [-v version]* *[-u user-branch|-l] [-q]*
Options:	*no = current branch* *-l = list of available branches matching with given user/release (flag - optional)* *-U = base user name (optional)* *-r = reference release label (optional)* *-b = reference branch name (optional)* *-v = version number on reference branch (optional, default is the latest)* *-u = user's branch name (optional, to be defined if you wish to create a branch)* *-q = skip confirmation step (optional)*

La commande ***git_branch*** est en quelque sorte l'équivalent de la commande ClearCase *cc_getpack* . Elle permet :

1) de créer et de faire le « checkout » d'une nouvelle branche ;
2) de sélectionner (et de faire le « checkout ») une branche existante dans une liste ;
3) d'obtenir une liste de branches disponibles (options ***-l***) .

L'option ***-q*** permet de « shunter » la demande de confirmation de création d'une nouvelle branche.

NB: Lorsqu'une branche est créée avec ***git_branch***, elle est automatiquement ajoutée dans la base de méta-données GCO .

ATTENTION!!

1) L'option ***-U*** a pour argument le nom d'utilisateur au sens utilisateur Git enregistré à GCO, et non au sens utilisateur système (ex: *gco* et pas *marp001*).

2) La notion de vue publique n'a plus de sens sous Git: on peut donc ouvrir et modifier les branches de n'importe quel utilisateur autre que soit-même. Par contre, il ne sera pas possible pour un utilisateur de poster des modifications faites sur une branche dont il n'est pas propriétaire, via la commande ***git_post*** .

git_commit

Objet:	*Add file contents to the index, and record changes to the repository.*
Niveau:	*Standard*
Usage:	*git_commit [-f file-name|-m commit-message]*
Options:	*-f = take the commit message from given file (optional)* *-m = use the given message as commit message (optional)*

La commande ***git_commit*** est une sorte de « wrapper » de la commande Git standard *commit* . Elle réalise deux opérations successives:

1) l'ajout des modifications à l'index Git à partir de la racine du dépôt Git (commande classique *git add)*;
2) exécution du commit (commande classique *git commit*) .

Les options ***-f*** et ***-m*** s'excluent mutuellement, mais l'utilisation d'une de ces 2 options est obligatoire. Si le message du commit est court, l'option ***-m*** s'impose. Si le message est composé de un ou plusieurs paragraphes, il vaut mieux écrire le message dans un fichier et utiliser l'option ***-f*** .

NB: La commande ***git_commit*** ne postera pas automatiquement les modifications de la branche courante vers le dépôt central: il faut utiliser la commande ***git_post*** pour cela, après avoir effectué un (ou plusieurs) commit(s) .

git_diff

Objet:	*Displays differences between multiple versions files (predecessor by default).*
Niveau:	*Standard*
Usage:	*git_diff [-h] filename*
Options:	*-h = choose in history of the file*

La commande ***git_diff*** est la parfait équivalent de la commande ClearCase *cc_diff* . Elle permet d'afficher les différences entre 2 (min) ou 3 (max) versions d'un fichier présernt dans le dépôt Git dans un éditeur.

Si l'option ***-h*** est utilisée, l'utilisateur est invité à choisir les versions du fichier à comparer dans la liste des versions disponibles. Dans le cas contraire, le « diff » se fera entre la version courante du fichier et son prédecesseur.

ATTENTION!!

L'éditeur de texte choisi par défaut pour afficher le « diff » dépend de la machine sur laquelle on travaille:

- sur yuki/kumo , l'éditeur par défaut est *vimdiff* ;
- sur serran/mostelle, l'éditeur par défaut est *gvimdiff* ;
- sur mirage/merou, l'éditeur par défaut est *kdiff3* .

On peut choisir son propre éditeur de texte pour le « diff » via la variable d'environnement *GIT_DIFF_EDITOR* , **qui doit comporter le chemin complet vers l'éditeur** (ex: /usr/X11R6/bin/gvimdiff).

Toutefois, certains éditeurs (dont *gvimdiff*) s'exécutent par défaut en « background », à moins d'utiliser une option permettant d'éviter cela (***-f*** pour *gvimdiff*). Dans le cas où on choisit son propre éditeur de texte pour le « diff », il convient donc de préciser cette option, par exemple:
export GIT_DIFF_EDITOR = « /usr/X11R6/bin/gvimdiff -f »

git_edit

Objet:	*Edit a file and save modifications, with confirmation step.* *If option "-k" is used, current version will not be modified, a temporary version will be kept in file's directory.* *A warning is displayed if modified versions of the file exist for the same release.*
Niveau:	*Standard*
Usage:	*git_edit [-k] [-q] file-name*
Options:	*-k = keep origin file (optional)* *-q = quiet mode (optional)*

La commande ***git_edit*** est la parfait équivalent de la commande ClearCase *cc_edit* . Elle permet d'éditer un fichier présent dans le dépôt Git en vue de modifications.

Si l'option ***-k*** est utilisée, la version courante du fichier édité ne sera pas écrasée, et les modifications seront conservées dans un fichier nommé *file-name.KEEP* .

Si l'option ***-q*** est utilisée, il ne sera pas demandé de confirmation de sauvegarde des modifications .

ATTENTION!!

L'éditeur de texte choisi par défaut pour éditer un fichier dépend de la machine sur laquelle on travaille:

- sur yuki/kumo , l'éditeur par défaut est *vim* ;
- sur serran/mostelle/mirage/merou, l'éditeur par défaut est *gvim* .

On peut choisir son propre éditeur de texte via la variable d'environnement *GIT_DEDITOR* , **qui doit comporter le chemin complet vers l'éditeur** (ex: /usr/X11R6/bin/gvim).

Toutefois, certains éditeurs (dont *gvim*) s'exécutent par défaut en « background », à moins d'utiliser une option permettant d'éviter cela (***-f*** pour *gvim*). Dans le cas où on choisit son propre éditeur de texte, il convient donc de préciser cette option, par exemple:

export GIT_EDITOR = « /usr/X11R6/bin/gvim -f »

git_fetch

Objet:	*Fetch a remote (or origin) repository, and update (all or) given branche(s) if required.*
Niveau:	*Standard/Advanced*
Usage:	*git_fetch [-o origin-repository] [-all|branch-name [branch-name ...]]*
Options:	*-o = remote repository (optional - default is origin)* *-all = update all branches (optional)*

La commande *git_fetch* « encapsule » à la fois les commandes Git natives *remote add* (en quelque sorte: création d'un alias vers un dépôt distant) et *fetch* (récupération/mise à jour des références & objets d'un dépôt distant vers le dépôt local).

Si l'option ***-o*** n'est pas utilisée, le dépôt distant correspond tout simplement au dépôt à partir duquel a été cloné le dépôt courant, donc le dépôt central GCO dans la plupart des cas.

Si le dépôt courant est effectivement un clone du dépôt central GCO, l'utilisation de *git_fetch* permettra de mettre à jour les branches distantes, ainsi que l'ensemble des tags contenus dans le dépôt local. Pensez donc à l'utiliser (très) régulièrement, elle est très rapide sans argument.

ATTENTION!!

Néanmoins, si on utilise en argument(s) un ou plusieurs noms de branche (ou ***-all*** pour toutes les branches – ça risque de prendre beaucoup de temps dans ce cas), on aura pour chaque branche donnée:

1) soit cette branche est une branche locale, et dans ce cas elle sera mis à jour par rapport à la branche distante du même nom située dans le dépôt distant, via un simple *merge* (*) ;

2) soit cette branche n'existe pas localement: dans ce cas elle est créée et mise à jour par rapport à la branche distante du même nom située dans le dépôt distant, via un simple *merge* (*) .

Cela suppose bien évidemment que les branches données en argument existent dans le dépôt distant!!

L'utilisation de *git_fetch* se fera sans argument dans la très grande majorité des cas...

(*) En principe, on s'attend à ce que chaque merge de branche soit automatique. Si ce n'est pas le cas, la mise à jour des branches locales échouera.

git_finger

Objet:	*Find and display informations about Git users, matching with given pattern.*
Niveau:	*Standard*
Usage:	*git_finger user-pattern*
Options:	*no*

La commande ***git_finger*** permet de rechercher et d'obtenir des informations sur un (ou plusieurs) utilisateur(s) Git, enregistrés côté GCO . Le *user-pattern* fourni sera recherché dans le nom, le prénom, le nom d'utilisateur, et les alias.

git_help

Objet:	*Display help on a Git's user command.*
Niveau:	*Standard*
Usage:	*git_help [git_]command-name*
Options:	*no*

La commande git_help permet d'afficher un message d'aide concernant la commande Git donnée en argument.

git_history

Objet:	*Display all versions of a file.*
Niveau:	*Standard*
Usage:	*git_history file-name*
Options:	*no*

La commande ***git_history*** permet d'afficher les différentes versions d'un fichier présent dans un depôt Git, de façon analogue à ce que produit la commande ClearCase *cleartool lshistory* .

git_info

Objet:	*Return informations about given file/commit/branch/tag/version.* *If no kind is given, item is supposed to be a Git's repository file.*
Niveau:	*Standard*
Usage:	*git_info [-all] [[kind:]item-name]*
Options:	*-all = also print item's log message (optional)*

La commande ***git_info*** permet d'afficher des informations concernant un élément présent dans un dépôt Git. Cet élément peut être un fichier (kind = element), un commit (kind=commit), une branche (kind=branch), un tag (kind=tag), ou une version (kind=version) . Si le « kind » n'est pas spécifié, l'élément donné en argument est supposé être un fichier.

L'option **-all** affiche également le message du commit relatif à l'élément donné.

En un sens, cette commande équivaut à la commande ClearCase *cleartool describe* .

git_list

Objet:	*List all elements of a branch in current directory.*
Niveau:	*Standard*
Usage:	*git_list [-u user] [-r release] [-b branch]*
Options:	*-u = user login name (optional - default is the current one)* *-r = release label (optional - default is the current one)* *-b = branch name extension (optional - default is the current branch extension)*

La commande ***git_list*** est l'équivalent de la commande ClearCase *cc_list* . Elle permet d'afficher les modifications effectuées sur une branche donnée. Si aucun argument n'est fourni, la branche considérée est la branche courante.

ATTENTION!!

1) L'option ***-u*** a pour argument le nom d'utilisateur au sens utilisateur Git enregistré à GCO, et non au sens utilisateur système (ex: *gco* et pas *marp001*).

2) La commande ***git_list*** ne fonctionnera pas si la branche considérée n'a pas été enregistrés dans la base de méta-données côté GCO . En effet, il est nécessaire de connaître le commit sur lequel est basé la branche considérée. Malheureusement, les commandes Git standard ne permettent pas de connaître cette information.

git_login

Objet:	*Login as Git user.*
Niveau:	*Standard*
Usage:	*git_login [user-name] [password]*
Options:	*no*

La commande ***git_login*** permet de s'authentifier sur la machine courante en tant qu'utilisateur enregistré de la base de méta-données Git côté GCO , et ainsi d'être autorisé à mettre à jour le dépôt central Git .

On peut fournir le nom d'utilisateur et le mot de passe directement en ligne de commande, ou bien le nom d'utilisateur seul (dans ce cas le mot de passe sera demandé de façon interactive), ou bien ne rien fournir du tout (dans ce cas le nom d'utilisateur et le mot de passe seront demandés de façon interactive).

Si un utilisateur est déjà authentifié sur une machine, l'exécution de la commande ***git_login*** , avec ou sans arguments, retourne son nom d'utilisateur.

Si le « home directory » de plusieurs machines pointe vers le même répertoire (ex: yuki & yukisc2), il est nécessaire de s'authentifier sur chacune de ces machines.

git_logout

Objet:	*Logout from Git's server usage on local host.*
Niveau:	*Standard*
Usage:	*git_logout*
Options:	*no*

La commande ***git_logout*** permet de se « dés-authentifier » du serveur de méta-données Git, sur la machine courante.

git_merge

Objet:	*Merge specified file or branch to the current view.* *The "from" version is defined by the release, user logname* *and optional branch name extension.* *WARNING: some options are mutually exclusive:* *1) options -f and -u/-r/-b/-v ;* *2) options -u/-r/-b and -v ;* *3) options -f/-u/-r/-b/-v and supply a ref-spec .* *Argument "ref-spec" could be a tag, a commit-hash, a branch name, etc...*
Niveau:	*Standard/Avancé*
Usage:	*git_merge [-s strategy] [-f file-name] [-u user-name] [-r release] [-b branch]* *[-v version] [-o output-file] [-print] [-quiet] [ref-spec]*
Options:	*-s = use the given merge strategy (optional - default is "resolve")* *-f = read merge reference specifications from given log file (optional)* *-u = user Git logname (optional - default is current user)* *-r = release label (optional - default is current release)* *-b = branch name extension (optional - default is curent branch extension)* *-v = full version selector, such as gco_CY37T1_op1/1 (optional)* *-o = merge output file-name (optional)* *-print = print list of merges to perform, then exit (optional)* *-quiet = skip confirmation step (optional)* *ref-spec = real Git's ref-spec to merge (optional)*

La commande ***git_merge*** est plus qu'un « wrapper » pour la commande standard *git merge* : elle permet de connaître à l'avance ce qu'il y aura à faire lors du merge de la branche considérée (merges « fast-forward » ou automatiques, merge manuels, routines supprimées/modifiées/renommées), comme on peut le faire sous ClearCase avec *cc_merge* . Pour cela, on procède de la manière suivante:

1) Le commit courant est mémorisé (i.e. celui qui apparaît en premier via la commande Git réelle *git log*) .
2) On exécute le merge avec la commande Git réelle *git merge* .
3) On récupère la liste des modifications effectuées (merges « fast-forward » ou automatiques, routines supprimées/modifiées/renommées) et la liste des fichiers à merger de façon manuelle via la commande Git réelle *git status* .
4) On revient au commit initial, mémorisé en (1) .

Si l'option ***-q*** est utilisée, il ne sera pas demandé de confirmation avant de lancer effectivement le merge. Par contre, dans ce cas, la commande ***git_merge*** ne lancera pas les opérations de merges manuels, s'il en existe.

ATTENTION!!

1) Contrairement à la commande Git réelle *git merge* , on ne peut merger qu'une seule branche à la fois.

2) L'option ***-s*** permet de modifier la stratégie de merge utilisée par la commande Git réelle *git merge* . La stratégie utilisée par défaut est la plus « safe », à savoir *resolve*. **Cette option est à utiliser avec précaution et en toute connaissance de cause!!** (usage avancé)

3) L'option ***-u*** a pour argument le nom d'utilisateur au sens utilisateur Git enregistré à GCO, et non au sens utilisateur système (ex: *gco* et pas *marp001*).

4) Au lieu d'utiliser les options classiques ***-u***/***-r***/***-b*** , il est possible d'utiliser l'option ***-v*** avec en argument une branche versionnée (ex: gco_CY37T1_op1/1), ou encore de donner en argument un commit ou un tag (usage avancé).

5) La liste des opérations de merge à effectuer peut être sauvegarder dans un fichier en utilisant l'option ***-o*** .

6) L'option ***-f*** permet de reprendre une session de merges manuels interrompue par un Ctrl+c par exemple, dans le cas où la liste des opérations de merge à effectuer a été sauvegardées (cf point (5) ci-dessus).

7) Contrairement à ClearCase, Git ne dispose pas d'une interface graphique permettant de faciliter les merges manuels. Cela dit, il est possible d'utiliser certains outils issus du monde libre ou non libre (*kdiff3*, *[g]vimdiff*, *tkdiff*, *meld*, *p4merge*, etc...). Parmi ceux-ci, l'outil le mieux adapté est *kdiff3*, installé uniquement sur la machine *merou*: **il est donc fortement recommandé de ne faire des opérations de merge que sur la machine *merou*** .

git_mod

Objet:	*Returns historic list of modifications performed on the specified file for a given release.*
Niveau:	*Standard*
Usage:	*git_mod -r release file-name*
Options:	*-r = release label*

La commande ***git_mod*** est l'équivalent de la commande ClearCase *cc_mod* . Elle retourne la liste des versions d'un fichier d'un dépôt Git, basées sur la release donnée.

git_password

Objet:	*Change password of current authentified Git user.*
Niveau:	*Standard*
Usage:	*git_password*
Options:	*no*

La commande ***git_password*** permet de changer de façon interactive le mot de passe de l'utilisateur Git courant.

git_post

Objet:	*Post modifications on current branch to GCO's main repository.*
Niveau:	*Standard*
Usage:	*git_post*
Options:	*no*

La commande ***git_post*** permet:

1) la mise à jour du serveur de méta-données Git-GCO avec les dernières modifications effectuées sur la branche courante ;
2) la mise à jour du dépôt central Git via la commande Git réelle *git fetch* ;
3) la mise à jour de la branche courante sur le dépôt central Git via la commande Git réelle *git fetch* .

ATTENTION!!

La commande ***git_post*** ne fonctionnera pas si:

1) l'utilisateur n'est pas authentifié sur la machine courante ;
2) l'utilisateur n'a pas les droits en écriture sur le dépôt central ;
3) l'utilisateur n'est pas le propriétaire (i.e. Le créateur) de la branche courante.

git_public

Objet:	*Make current branch public (or private), for all (or given) users.*
Niveau:	*Standard*
Usage:	*git_public [-cancel] -all|user-name [user-name ...]*
Options:	*-cancel = make branch private (optional)* *-all = make branch public or private for all users (optional)*

La commande ***git_public*** permet de rendre publique la branche courante soit pour l'ensemble des utilisateurs, soit pour certains utilisateurs. Dans le cas, cette branche devient modifiable par les utilisateurs concernés.

L'option ***-cancel*** permet de rendre à nouveau la branche privée, soit pour tous utilisateurs, soit pour certains utilisateurs.

Il est bien évident que seul le propriétaire de la branche est autorisé à réaliser cette opération...

git_start

Objet:	*Initialization of a Git user's environment.*
Niveau:	*Standard*
Usage:	*git_start*
Options:	*no*

La commande ***git_start*** permet à une utilisateur Git d'initialiser son environnement afin d'utiliser la boîte à outils Git-GCO et le serveur de méta-données Git-GCO sur la machine courante:

- vérification des variables d'environnement spécifiques à définir: GIT_INSTALL, GIT_ROOTPACK, GIT_HOMEPACK, GIT_WORKDIR) ;
- vérifications du PATH ;
- création des répertoires *.git-workdir* & *git-dev* ;
- authentification si l'utilisateur dispose déjà d'un compte Git ;
- « clonage » du dépôt central Git .

Le compte-rendu d'exécution de la commande ***git_start*** est disponible dans le fichier *$HOME/git_start.log* .

A noter que cette commande peut être exécutée n'importe quand: si tout est vérifié elle ne fera... rien!

git_user

Objet:	*Display informations about given Git user.*
Niveau:	*Standard*
Usage:	*git_user user-name*
Options:	*no*

La commande ***git_user*** retourne des information sur l'utilisateur dont le nom d'utilisateur Git est donné en argument.

git_view

Objet:	*Return full selection rules in current (or given) branch.*
Niveau:	*Standard*
Usage:	*git_view [branch-name]*
Options:	*no*

La commande ***git_view*** retourne en quelque sorte l' « empilage » des règles de vue de la branche courante ou de la branche donnée en argument, de la même façon que ce que renvoie la commande ClearCase *cc_view* .

ATTENTION!!

Contrairement à ClearCase, il n'est pas possible d'intercaler une branche d'un utilisateur dans la vue courante.

ANNEXE: Quelques commandes Git natives

Voici quelques commandes Git natives fort utiles, qui permettront quelquefois d'aller un peu plus vite...

- Cloner un dépôt:

*% git **clone** URL-d'accès-vers-le-dépôt-distant*

- Cloner un dépôt en lui attribuant un nom différent du dépôt d'origine:

*% git **clone** URL-d'accès-vers-le-dépôt-distant nom-du-dépôt-local*

- Obtenir la liste des branches locales:

*% git **branch***

- Obtenir la liste des branches distantes:

*% git **branch** -r*

- Créer une nouvelle branche par-dessus la branche courante, sans toutefois changer de branche:

*% git **branch** nom-de-la-branche*

- Créer une nouvelle branche par dessus une référence quelconque (un commit, un tag, un nom de branche [local ou distant], etc...), sans toutefois changer de branche:

*% git **branch** nom-de-la-branche référence*

- Ouvrir une branche existante:

*% git **checkout** nom-de-la-branche*

- Créer et ouvrir une nouvelle branche par dessus la branche courante:

*% git **checkout** -b nom-de-la-branche*

- Créer et ouvrir une nouvelle branche une référence quelconque (un commit, un tag, un nom de branche [local ou distant], etc...):

*% git **checkout** -b nom-de-la-branche référence*

- Savoir où on en est depuis le dernier commit (fichiers modifiés, ajoutés, renommés, effacés, pas encore ajoutés à l'index, etc...):

*% git **status***

- Mettre à jour l'indexation des fichiers contenus dans le dépôt (NB: à exécuter dans le « top directory » du dépôt):

*% git **add** .*

- Faire un commit avec un message court:

*% git **commit** -m 'message de commit'*

- Faire un commit en invoquant un éditeur de texte:

*% git **commit***

- Faire un commit en utilisant le texte contenu dans un fichier comme message:

*% git **commit** -F nom-du-fichier*

- Modifier le message du dernier commit, ou bien carrément modifier le contenu du dernier commit avec des modifications faites depuis(NB: à proscrire si ce commit a déjà été posté via *git_post*):

*% git **commit** --amend*

- Retour arrière vers le commit précédent, en conservant les modifications faites depuis, prêtes à être « commitée » à nouveau (NB: à proscrire si ce commit a déjà été posté via *git_post*):

*% git **reset** –-soft HEAD^*

- Retour arrière vers le commit précédent, sans conserver les modifications faites depuis (NB: à proscrire si ce commit a déjà été posté via *git_post*):

*% git **reset** –-hard HEAD^*

- Merge d'une branche

*% git **merge** nom-de-la-branche*

- « Empilage » des commits sur la branche courante:

*% git **log***

- « Empilage » des commits sur une référence (une branche, un tag, un fichier, etc...) :

*% git **log** référence*

- Nettoyage des fichiers Git du dépôt devenus inutiles, et optimisation/compression des données (NB: à faire régulièrement):

*% git **gc***

- Ajout d'un dépôt distant, et mise à jour des données de ce même dépôt:

*% git **remote add** alias-du-dépôt-distant URL-d'accès-vers-le-dépôt-distant*

*% git **fetch** alias-du-dépôt-distant*

- Supprimer un (ou plusieurs) fichier(s) ou répertoires(s) (NB: syntaxe analogue à la commande UNIX « rm »):

*% git **rm** [-r] nom-du-fichier [nom-du-fichier ...]*

- Renommer/déplacer un (ou plusieurs) fichier(s) et/ou répertoire(s) (NB: syntaxe analogue à la commande UNIX « mv »):

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