**Distributed revision control system**

**Switch to Git**

Adrien Bruneton ([adrien.bruneton@cea.fr](mailto:adrien.bruneton@cea.fr))

06/2013

(last update 02/2014)

[1. Motivation - SALOME context 1](#_Toc379976251)

[2. Collaboration model 2](#_Toc379976252)

[Roles 2](#_Toc379976253)

[Repository organization – Single reference repository model 2](#_Toc379976254)

[3. Workflow 3](#_Toc379976255)

[Release branch vs development lines 4](#_Toc379976256)

[Integrator per module 5](#_Toc379976257)

[4. Comparison with the CVS work-flow 6](#_Toc379976258)

[5. Git public repositories 7](#_Toc379976259)

[6. Effective switch to the new system 7](#_Toc379976260)

[Preliminary tests 7](#_Toc379976261)

[Keeping source history 7](#_Toc379976262)

[Switch to the new setup 8](#_Toc379976263)

[7. Guidelines 8](#_Toc379976264)

[Commits 8](#_Toc379976265)

[Branch management 9](#_Toc379976266)

[8. Conclusion 10](#_Toc379976267)

[9. References 10](#_Toc379976268)

# Motivation - SALOME context

The SALOME platform is made of several modules relatively independent one from the other. The code is currently revised under CVS, the central repositories being managed by OpenCascade in Nijni, Russia.

CVS is quite an old Version Control System (VCS) and presents some drawbacks in comparison to newer tools:

* centralized repository: this represents a weak point in the global infrastructure and restricts the choice of available collaboration models (see section below).
* there is no distinction between what can be made locally and what is published (unable to commit/save work when off-line);
* commits are performed on a file basis (no notion of global change-set / unit of work);
* cumbersome and heavy branch management (e.g. need to tag after a merge to keep track of the merge history);
* *<many more ...>*

In this document we propose a strategy to switch to a newer system (namely Git) and detail how the change would be performed. Most importantly we also suggest a collaboration model and work-flow to be used with this new tool, so that all SALOME developers work in a consistent way.

# Collaboration model

We present here the different profiles involved when dealing with Git and the organization of the public and private repositories. This topic is closely related to how branches are managed: why and when are they created, and who manages them (see next section).

## Roles

Based on the actual setup the following roles are identified:

The “simple” roles first:

* **Readers:** read-only access to the repositories (all branches/tags): can't push; can't put tags; no destructive permissions.
* **Writers:** 1st-level write access: write to any branch except master and release branches; can put any tag except version tag; no destructive permissions.

Then the “advanced” roles (also called “integrators” later in the document):

* **Committers:** 2nd-level write access: write to any branch including master except release branches; can put any tag except version tag; no destructive permissions.
* **Releasers:** 3rd-level write access: write to any branch including master and release branches; can put any tag including version tag; destructive permissions (delete branches).
* **Admin:** administrators of the repository. Can add / remove / configure repositories; can add / remove users and groups; can change permissions.

## Repository organization – Single reference repository model

Several organization models are possible with a newer VCS. The reader is referred to [1] for an example of possible models. For SALOME it is suggested to adopt the following approach.

**One Git repository per SALOME module**

Given their complexity and relative independence each SALOME module (KERNEL, GUI, MED, etc ...) will still be stored in a dedicated repository. This is already currently the case with CVS (with the exception of LIBBATCH and KERNEL) and we thus maintain the possibility to easily “compose” a custom application involving different modules.

**Single reference repository**

For each SALOME module, one central reference repository (also called public repository here) is managed by the integrator. A contributor makes his own private clone of the reference repository and pulls/pushes directly from/to it.

This works well if the work-flow is mainly sequential, and non-conflictual pieces of work are submitted in general, which is the case for SALOME. When a conflict arises, the contributor who is currently pushing has to resolve it. This is more or less the current model under CVS, but with Git we gain the feature that one has a local copy of the whole repository, thus allowing off-line work.

The complete list of repositories available for the SALOME platform can be found at:

[http:/git.salome-platform.org/gitweb/](http://git.salome-platform.org/gitweb/)

Pros / cons:

* simple setup
* less integration issues / “low” maintenance cost for the integrator
* easily managed at release time (see below)
* on the negative side: any contributor can mess up the reference repository accidentally (pushing his own private branches, delete remote branches, etc ...).

The last problem can be mitigated by the setup of appropriate Git hooks, preventing for example an unwanted push in the “master” branch. It is also assumed that the group of people actually contributing (i.e. doing more than just cloning) is knowledgeable enough to avoid big mistakes …

# Workflow

The proposed workflow is as follows: the main branch of the repository is the "master" one. Apart from the commits generated by the integrators when merging, no direct modification is made in this branch.

---o----o---o---o--->

topic branches /

/ --o--o--o--

/ / \

master ---------------------------o------->

\

V7\_2\_BR -------o----o---o----o------o----o---]

| |

Tag: V7\_2\_0 Tag: V7\_2\_1

*Proposed branching model – One line of development*

Developments or bug fixes, even minor, are performed by creating a new topic branch derivating from "master":

git checkout -b <my\_imp>

Such a branch can be short-lived for a simple bug-fix for example, or may remain alive for a long time when dealing with a change in architecture, or a significant improvement.

The branch is published on the public repository and automatically tracked with

git push --set-upstream origin <my\_imp>

(the "--set-upstream" option, or shortly "-u", sets the tracking of the local branch to the remote one).

When the development is ready to be integrated, the integrator is informed (by mail) and the branch is merged (by an integrator) with:

git checkout master

git merge --no-ff --log <my\_imp>

(the two options --no-ff and --log are good practice, but can be omitted - they allow to produce more comprehensive merge messages).  
Significant topic branches may need to go through COTECH approval before being merged – see the discussion in the section Guidelines below.

If the branch will not be used anymore it can be deleted locally issuing the command:

git branch -d <my\_imp>

and deleted also on the public repository with:

git push origin :<my\_imp>

## Release branch vs development lines

One development line

This is the preferred situation shown on the first figure above.

Everytime a release is being prepared a new branch is created from “master”, marking the point where developments which were not merged at that time are no more integrated in the release candidate. Only bug fixes and other minor tweaks are committed into this branch.

Ideally there is one such branch for every release, except for patch releases. For example, one would create the branch V6\_7\_BR to prepare the release of SALOME V6.7.0. The release of V7.2.0 triggers the creation of another branch V7\_2\_BR, also taken from "master". It is thus assumed that "master" is the **principal line of development**, and always contains the most recent work.

If a patch release is to be done (see tag V7\_2\_1 above), the corresponding bug fix is published in the already existing release branch V7\_2\_BR, and can be, **only if needed**, cherry-picked and integrated into "master". Some bug fixes are indeed specific to a given release (e.g. the module affected by the bug is anyway being completely rewritten for the next release), meaning the replication to "master" is not always meaningful.

Two parallel development lines

Recent history (SALOME version 6.7.0 and 7.2.0) shows that there might be the need for more than one line of development. Such a situation should be, as much as possible, be solved using **topic branches**. Those can be started early, kept alive for a long time, and merged only when the new feature is deemed ready for integration. They can be kept up-to-date with recent changes in "master", with a regular merge of master into the topic branch:

git checkout <my\_topic\_branch>

git merge master

(this is better than a rebase, since pulling/pushing rebased branches might be problematic). Ideally, no release branch should be rooted directly from such a branch.

*Practically* however, if there is **a real need of another active development line** (with a separate release process), the integrator may choose to derivate a new major branch from master. For example, a branch V7\_BR is created, assuming implicitly that the work merged into "master" is reserved for a next major release (V8 in this case). A release is then made by rooting a third branch from this V7\_BR as shown on the figure below:

topic --o--o--o--

/ \

master ---------------------------------------------> (1)

\

V7\_BR -------------------o------------------> (2)

\ / \

topic --o--o--o-- \

\

V7\_2\_BR -------o----o---o--->

|

Tag: V7\_2

*Alternative model – Two lines of development (1) and (2)*

This is naturally more work for the integrators who have two maintain more than one development line, and more than one release branch.

It is again highlighted that this should not be the standard:

* the maintenance is more complicated
* there is a significant risk that all developments are made into V7\_BR and that nothing more goes into master
* the development lines may diverge a lot after some time

A typical usage of such a model in the past was triggered by the changes of prerequisites in SALOME (switching to higher versions). This can perfectly be done via a topic branch, in which one regularly merges the latest developments of master.

## Integrator per module

Nijni remains the main integrator for all SALOME modules, and keeps the exclusivity on the fundamental (and historical) modules. In addition the following persons are responsible and competent for the modifications/merges in the corresponding module:

**KERNEL**  Nijni / one person from EDF (P. Rascle?)

**GUI**  Nijni

**GEOM**  Nijni / P. Rascle (EDF)

**SMESH** Nijni / P. Rascle (EDF)

**MED**  A. Geay (CEA) / A. Bruneton (CEA)

**PARAVIS** A. Ribes (EDF) / Nijni

**YACS**  O. Mircescu (EDF) / P. Rascle (EDF)

**HEXABLOCK** F. Kloss (CEA)

# Comparison with the CVS work-flow

This section is targeted at SALOME's developers coming from the CVS world and not knowing Git yet.

Cloning and branch selection

cvs co -r V6\_main -d MY\_KERNEL\_1 KERNEL\_SRC

-> this only pulls the branch “V6\_main”. Working in another branch typically requires another copy (in the directory MY\_KERNEL\_2 here):

cvs co -r V7\_main -d MY\_KERNEL\_2 KERNEL\_SRC

In Git branches are very light objects. The user starts cloning the repository, which automatically retrieve all the live branches:

git clone path/to/the/repository/KERNEL\_SRC MY\_KERNEL

and can then select in which branch he/she wishes to work:

git checkout –b a\_nice\_feature origin/a\_nice\_feature

The initial (read-only) clone can be made very light with the --depth option:

git clone --depth=1 path/to/the/repository/KERNEL\_SRC MY\_KERNEL

Note however that this option prevents the repository of being used for contributing.

Commit

A typical commit workflow in CVS is performed by:

cvs update -dP

where the user has to resolve the conflicts at this stage. Then the commit can be performed with:

cvs commit -m “A relevant message”

but both commands will only work if the host machine is on-line.

With Git (when no new file has been created) the commit is performed locally by:

git commit -a -m “A relevant message”

(which will always work whatever the network status) and is then synchronized with the reference repository with:

git pull

(at which point potential conflicts have to be resolved and re-commited) and then:

git push

which updates the reference repository with the user's changes.

Merge

The merge operation is the most significant in terms of ease of use when using Git instead of CVS.

With CVS, this involves some work:

cd BRANCH\_DIR

cvs update -kk -j <A\_BRANCH> -j <ANOTHER\_BRANCH>

cvs commit -m “merging another branch”

cvs tag -c “after\_merge”

which means that an explicit change of directory is required, then an explicit commit and finally a tag to identify the point of merge in the history (otherwise this information is lost).

In Git, supposing you're in “master” and want do merge “my\_dev”:

git merge [-–no-ff] my\_dev

This will create a merge commit (if there is no conflict, as it should be the case here since no commits come into “master”), and Git mechanics is so that the merge information is also explicitly saved (the commit has two parents). The “--no-ff” option can be added to ensure the merge is always visible in the history, even if “master” had no new developments.

This last point makes the intensive use of branches a very handy tool when working with Git.

# Git public repositories

The author has little experience managing a public repository which is to be exposed above a VLAN. Nijni should advise if there is any restriction from a server point of view. They will indeed keep on managing all public repositories and perform the actual migration.

The communication with an external Git repository is also to be tested from the EdF R&D site at Clamart, where strong restrictions apply when performing remote communications.

# Effective switch to the new system

## Preliminary tests

The reference repository is setup at a given time point. A few voluntary contributors agree to perform a few commits into the new GIT repository to ensure everything works smoothly in terms of synchronization and contribution.

Following discussions, it is suggested to perform such a test for a week on the KERNEL module, and on the biggest SALOME modules (to test the performance of the initial git clone).

## Keeping source history

A previous analysis was conducted by Nijni and has shown that a tool like cvs2git can be used to convert the current CVS repositories to Git. The tool is able to automatically identify matching CVS commits across different files and to group them into one sensible Git commit.

Keeping the history has the obvious advantage of being able to easily browsing the past of the project without needing to switch between several infrastructures.

The preliminary tests should confirm that the impact of this on the initial clone is acceptable.

## Switch to the new setup

A list of all current CVS contributors is built from the list of CVS accounts. A date is fixed between two SALOME releases, ideally when as few branches as possible are alive in CVS. All contributors are informed:

* on the day prior to the switch, nobody commits, thus leaving time to the integrator to synchronize the new GIT reference repository with the latest status of CVS. This implies notably:
  + importing the history from CVS
  + doing all the necessary merge/renaming operations so that we achieve the model presented above (branch names for example);
* starting on the day of the switch, a contributor can:
* clone the reference repository to have his own copy;
* checkout/create the appropriate branch;
* make this clone his/her main working directory for further developments;
* stop committing to CVS;

The CVS repositories can be kept alive read-only for a while in case something goes wrong with the new setup.

# Guidelines

## Commits

A commit in the public repository should ideally represent a functional piece of work:

* the whole code (including tests!) can be compiled and run after each commit, at least on one architecture (typically the one where the developer did the work);
* it represents at most one (and only one) functional unit: a bug fix, a new feature, the documentation for a coherent piece of code (a whole class typically), etc ... Note that having several commits for one feature is perfectly OK (as long as the first rule above still holds).

This means for example that two separate bug fixes are better stored as two different commits.

The commit message could be formatted as follows:

*<TYPE>*: *<module/class>*: brief description

*<blank line>*

Long description on several lines

Long description on several lines

...

where <TYPE> is a three-letter acronym in the following list:

* **BUG**: for a bug fix;
* **DOC**: when adding documentation;
* **IMP**: when adding a new feature (improvement);
* **REG**: when re-factoring some piece of code (API names change, etc ...);
* **MIN**: for a minor change not impacting the code a lot (we should see little of these in the reference repository, as the contributor would typically *squash* such a commit with a bigger one);

Example:

IMP: MEDOp: field computation from an analytical function of the space coordinates.

The MEDCoupling method applyFunc() is now transposed into the MEDOp world. The function to be applied is given as a string. Special variables « X », « Y », « Z » that can be used to represent the space coordinates of the underlying mesh.

The (only?) exception to this rule is the merge commit message which is automatically generated in some cases (Git fast-forward for example). As mentioned previously it is however recommended to avoid such situations by explicitly avoiding fast-forwards (--no-ff option).

Note that in the user’s private repository the commits can have any desired form. They are then typically rebased/squash to adhere to the above rule when pushing to the reference repository.

## Branch management

The integrator takes care of the creation and tagging of the release branches.

Feature branches can be created by any contributor and must fork from “master”. However a merge into “master” is subject to the agreement of the integrator responsible of the module (and potentially the COTECH), as this action decides whether it will go into the next release.

When a feature branch is not used anymore it is deleted in the reference repository.

Naming convention

The current naming convention can be retained. A release branch is called:

V<major>\_<minor>\_BR

for example "V6\_7\_BR" and once the release has been done, the corresponding tag is set on the release branch:

V<major>\_<minor>\_<patch>

A topic branch name should be human-readable and clearly have some sort of reference to the feature being worked on. It can also start with the three letters identifying the main person responsible of the topic. For example:

abn/med-calculator

for Adrien Bruneton, working on the MED Calculator.

Merging procedure – COTECH approval

For the most important topic branches, it is suggested to require the COTECH approval before doing any merge.

This ensures the feature being merged is presented to everyone, and potential conflicting developments are spotted early.

# Conclusion

Any constructive suggestion/criticism/comment to this document is highly welcome.

# References

[1] Git Book - <http://git-scm.com/book>

[2] GitHub – SSH key generation - <https://help.github.com/articles/generating-ssh-keys>

[3] Paul Rascle – CVS Workflow – Internal documentation