Major Atmospheric Gamma Imaging Cerenkov Telescope

MAGIC-TDAS 00-06
000309/JCGonzalez

On the policies for submission of new revisions of source code files

J C González
Max Planck Institute for Physics, Munich
<gonzalez@mppmu.mpg.de>

March 9, 2000

Abstract

For the development of the different programs needed for the analysis, data acquisition and control systems for The MAGIC Telescope a repository (archive) of source codes and documentation is needed. This repository must be handled by a version control framework, probably under the management of a *repository coordinator*. But, in addition, we need to define a clear policy regarding commits to this repository, as well as a set of rules about how to set test suites to be applied to any sub module of any given part of the system.

Contents

1 Introduction

For the development of the different programs needed for the analysis, data acquisition and control systems for The MAGIC Telescope a repository (archive) of source codes and documentation is needed. This is beyond any doubt. The structure of this repository will be very important in the further development of the system. A well organized repository will not only help the occasional developer to find the modules or programs that they are looking for. It will also help to modularize a whole application, by mapping the program hierarchy of subsystems into different modules in the repository.

Such a repository must be handled by a version control system. Currently the most wide-spreadly used program for that purpose is CVS (*Concurrent Versions System*)[1]. Some other good tools are available. Nevertheless, I will assume from now on that CVS is our version control system.

If you are doing development on your own, you can probably skip all this. However, when more than a person is working in a repository, all the questions exposed here will become very important.

2 The repository

2.1 An example

This is not the right place to talk about how to set up, use and manage a repository[2]. But, without any doubt, once the structure of the repository has been fixed, the most important thing to do is to decide which policy to use regarding commits. In order to fix concepts I will use a small example using CVS. The usual scenario could be as follows:

- A given user gets one or more modules from the server:

```
HAL10M:~> cvs checkout module1
HAL10M:~> cd module1
HAL10M:~/module1> ls
CVS         Makefile    algor.c     main.c      param.c     test.c
HAL10M:~/module1> _
```

- The user hacks away in this piece of code:

```
HAL10M:~/module1> emacs algor.c
  ...
HAL10M:~/module1> ls
CVS         Makefile    algor.c     algor.c~    main.c      param.c
test.c
HAL10M:~/module1> _
```

- Check that everything works (at least compiles):

[1]This is the tool we already decided to use for this task inside the MAGIC collaboration — see minutes of the 1st MAGIC Software Meeting, Feb.1999. There is already a (preliminary) repository running for our project, located in Munich.

[2]This is however very easy to do. Just have a look into [4]. There is also a nice front end to manage and access CVS repositories, called jCVS. See [2].

```
HAL10M:~/module1> make
 compiling algor.c ...
 compiling main.c ...
 compiling param.c ...
 linking -> libmodule1.so ...
 done.
HAL10M:~/module1> ls
CVS           Makefile      algor.c       algor.c~      algor.o
libmodule1.so  main.c        main.o        param.c       param.o
test.c
HAL10M:~/module1> _
```

- Check with a test program:

```
HAL10M:~/module1> make test
 compiling test.c ...
 linking -> module1-test ...
 done.
HAL10M:~/module1> ls
CVS           Makefile      algor.c       algor.c~      algor.o
libmodule1.so  main.c        main.o        module1-test*  param.c
param.o       test.c        test.o
HAL10M:~/module1> module-test
 ...
 done.
HAL10M:~/module1> _
```

- Commit your changes and release module:

```
HAL10M:~/module1> cvs commit
 ...
Checking in algor.c;
/usr/local/cvsroot/ta/algor.c,v  <--  algor.c
new revision: 1.5; previous revision: 1.4
done
HAL10M:~/module1> make clean
HAL10M:~/module1> cd ..
HAL10M:~> cvs release -d module1
You have [0] altered files in this repository.
Are you sure you want to release (and delete) module 'module1': y
HAL10M:~/> _
```

- Eventually, go home ...

2.2 Phases in the use of a version control system

In this sample session we have seen the essentials of a version control system:

1. *Check-out* of the module(s) you want to modify[3]

[3]To *check-out* a module is to get a working copy of the module from the repository.

2. *Modification/upgrade/correction* of the files
3. *Ask for compilation* (this is subject to discussion)
4. *Ask for test compliance* (this is also subject to discussion)
5. *Check-in* of the modified module(s)[4]

Strictly speaking, only the points 1, 2 and 5 are fundamental for the process of modification of a module from the repository, while the points 3 and 4 are two minimal suggestions to take into account in our submission policy[5].

2.2.1 Asking for compilation

In the point 3 we *ask for compilation.* This means, in practical terms, that *no modification done in the code must be committed* if the resultant code does not compile with the standard Makefile associated with that module. Of course, it might well be that the Makefile itself must be modified, but this is of course a valid operation. This is just a way to avoid problems for the rest of the developers, which might need the module you are using, but do not need or want to understand the details inside. In this sense, only *stable versions* will be found in the main trunk of the repository.

By the way, a repository is not a one-way line of development. This would be this *main trunk* just mentioned. But in principle one can create separate branches in the development starting at any point in the main trunk. This is specially useful when some corrections must be done in a previous release of an application, different from the current one — provided that this previous release is still in use by some community of users. But it is also very useful in the case a working group tries to improve some part of the application (or include new modules), but this improvement must not collide with the main development branch. Afterwards, when everything is ready, the new parts, modules and improvements in general can be incorporated in the main trunk (and perhaps will lead to a new major release of the application).

Not only stable versions should be available in the repository. Of course, any of the development/unstable versions should be available as well (following the *ask for compilation* rule), in parallel branches. This will lead to a faster improvement and debugging of the new code.

2.2.2 Asking for test compliance

Of course, asking for compilation is not sufficient. It is very easy to make a compilable code which or does not work as expected, or even does not work at all (in fact, it's quite common). Therefore, in order to be sure that our modification is nt a step backwards in the development, an permanent testing of the new code must be implemented. This can be done in different ways:

- By using a **test suite** which will test the final version of the application. This "final version" might be unstable, though. Indeed, we are not proving that each individual module is working properly, and in principle, in case of fatal error, nothing will tell us *where* is the mistake.

[4]To *check-in* a module is to put your modified copy of a module back in the repository. Most of the times, the term *commit* is used instead.

[5]In more elaborated development structures, we would include also the creation of a "differences" file, after checking that the code compiles and passes the test. This is of course not mandatory at all, but it would help to isolate the new code or modifications implemented. Afterwards these "differences" files can be used to apply simple *patches* to another working copies of the same code, checked-out by another developers.

- By using individual **test programs** for each module. This will assure that each of the sections of the program is working fine. This would be complemented by a **main test program**, which will test the final assembly of the whole application. But this is nothing but applying the general rule to the main module (which is a module like the others, anyway).

Several other policies could be applied. However, I suggest to follow the last one, namely to *include a small test program with every separated module, library or sub-system* in the repository, *complemented by another small main test program* also in the case of the "main module" that will use several of the other modules, in order to *check their integration and interrelation* without having to execute the actual (and probably more complex and slow) application.

3 Submission policies

The most dangerous step in the process outlined before is the *commitment*. It is dangerous because, without an strict control (or a full agreement on the rules), anyone can insert, at any point inside any branch of the repository (also the main trunk with the current, most developed version) a piece of code which simply does not work. Subsequently, the main application can fail. If the error is a fatal error (it simply aborts the program) or a severe computational error (it gives simply crazy results), then we would have a minor problem: this wrong code will be (or should be) easy to isolate and detect.

If the code we submitted gives apparently normal results, then we are going to have a really hard time to discover where it is.

There are two points which make of this *commitment* a weak step:

i. It depends on the decision of the user (*When do I commit?*)

ii. In the default way, it is **not** filtered or checked by a *repository coordinator* (*What/how do the people commit?*)

3.1 When do I commit?

This is the policy which the title of this small document refers to. There has to be an agreement and a fixed decision on this topic.

If you commit files too quickly you might commit files that do not even compile, as I said. If your partner updates his working sources to include your buggy file, he will be unable to compile the code.

On the other hand, other persons will not be able to benefit from the improvements you make to the code if you commit very seldom, and conflicts will probably be more common.

It is common to only commit files after making sure that they can be compiled. In addition, as I mentioned, I would recommend to require that the files pass a test suite or a test program in a module basis.

3.2 What/how do the people commit?

A *repository coordinator* is of course needed. It is also mandatory to check everything that is going to be added to the repository. However, one should put things very clear from the beginning and think twice before enforcing such a convention. As stated in [1]:

> "By making the development environment too controlled it might become too regimented and thus counter-productive to the real goal, which is to get software written."

4 Conclusions

The conclusions of this small document are very clear. First, the figure of the **repository coordinator** must be defined as a need. Second, the structure of the repository must be discussed and fixed. Third, not only access guidelines, but also a submission policy must be considered. Strategies like the suggested **ask-for-compilation** and **ask-for-test-compliance** can be very useful when a big group of people is working in the same system. The *repository coordinator* must ensure that this policy is fulfilled.

References

[1] Per Cederqvist et al. *Version Management with CVS.* Signum Support AB, 1992–1993.

[2] Timothy Gerard Endres. jcvs home page. WWW Electronic document, 1997. `http://www.ice.com/cvs`.

[3] J.C. González. *CVS: Concurrent Versions System.* Max-Planck-Institut für Physik, München, Feb. 1999. Presentation given at the I MAGIC Software Meeting, The Eng, Feb. 1999, available at http://hegra1.mppmu.mpg.de/~gonzalez.

[4] Peter Miller. *Aegis, a project change supervisor, User manual.* , 1999.

[5] Richard Stallman. GNU Coding Standards. Technical report, Free Software Foundation, Inc., 1998.