ISTITUTO NAZIONALE DI FISICA NUCLEARE

CNAF

INFN/TC-04/02
18 Marzo 2004

A SUPPORTING THE DEVELOPMENT PROCESS OF THE DATAGRID WORKLOAD MANAGEMENT SYSTEM SOFTWARE WITH GNU AUTOTOOLS, CVS AND RPM

List of authors in the following page

Abstract

DataGrid is project funded by the European Commission to develop and deploy distributed computing components for reference applications in the domains of Particle Physics, Earth Observation and Bioinformatics. The software *development* is shared among nine contractual partners, in seven different countries, and is organized in work-packages covering different areas. In this paper, we discuss how combination of Concurrent Version System, *GNU autotools* and other tools and practices was organised to allow the development, build, test and distribution of the DataGrid Workload Management System. This is the product of one speci_c work-package that is not only characterised by rather high *internal* geographic and administrative dispersion (four institutions with developers at nine different locations in three countries), but by the fact we had to integrate and interface to dozen of third-party code packages coming from different sources, and to the software products coming from other three development work-packages internal to the project.

high level of central co-ordination needed to be maintained for project-wide steering, and this had also to be re_ected in the software development infrastructure, while maintaining ease-of-use for distributed developers and automated procedures wherever possible.

PACS.: 11.30.Er,13.20.Eb;13.20Jf;29.40.Gx;29.40.Vj

Published by ***SIS–Pubblicazioni***
Laboratori Nazionali di Frascati

G. Avellino^{1}, S. Beco^{1}, B. Cantalupo^{1}, A. Maraschini^{1}, F. Pacini^{1},
S. Monforte^{2}, M. Pappalardo^{2},
D. Kouril^{3}, A. Krenek^{3}, Z. Kabela^{3}, L. Matyska^{3}, M. Mulac^{3},
J. Pospisil^{3}, M. Ruda^{3}, Z. Salvet^{3}, J. Sitera^{3}, M. Vocu^{3},
F. Giacomini^{4}, E. Ronchieri^{4},
A. Gianelle^{5}, R. Peluso^{5}, M. Sgaravatto^{5}
M. Mezzadri^{6}, F. Prelz^{6},
A. Guarise^{7}, R. Piro^{7}, A. Werbrouck^{7}
D. Colling^{8}
$^{1)}$ *DATAMAT S.p.A, GRID R&D Group, Space & Environment Division, Via Laurentina, 760 -I- 00143 Rome, Italy*
$^{2)}$ *INFN, Sezione di Catania, Dip. di Fisica e Astronomia dell'Universita' di Catania, Via S. Sofia 64, I-95123 Catania, Italy*
$^{3)}$ *CESNET, z.s.p.o., Zikova 4160 00 Praha, Czech Republic*
$^{4)}$ *INFN CNAF, Viale B. Pichat 6/2, I-40127 Bologna, Italy*
$^{5)}$ *INFN, Sezione di Padova, Dip. di Fisica Galileo Galilei, Via Marzolo 8, I-35131, Padova, Italy*
$^{6)}$ *INFN, Sezione di Milano, Dip. di Fisica dell'Universita' di Milano, Via G. Celoria 16, I-20133, Milano, Italy*
$^{7)}$ *INFN, Sezione di Torino, Via P. Giuria 1, I-10125, Torino, Italy*
$^{8)}$ *Imperial College London, UK*

1 Introduction

There are many problems related to the distributed development model of the DataGrid [1] project. A large number of people spread all over Europe has to write software packages that are inter-dependent and therefore call for frequent integration. The work-package 1, WP1, [2] dealing with the provision of a Workload Management solution was rather complicated in the project, both in terms of *internal* geographic and administrative dispersion (four institutions with developers at nine different locations in three countries), and in terms of software dependencies. It was therefore divided into components, under the responsibility of local development teams. The fundamental requirement for concurrent development directed us to the use of Concurrent Version System (CVS) [14], which is the most common solution in the open software community, as detailed in section 2.

The components contained in the Workload Management System (WMS) [15] have a complex dependency structure: such dependencies can be divided in four categories:

1. **Non-EDG packages:** packages that are developed outside the DataGrid project, such as MySQL, Boost libraries, Condor libraries and executables.

2. **Non-WMS EDG packages:** packages that are developed by EDG work-packages

different than WP1. For instance, in this category we can find for example Data Management and Information Service libraries.

3. **Modified non-EDG packages:** packages that are developed outside the DataGrid project, but that needed to be modified by WP1, such as trio, Bypass and the Globus FTP server. They needed to be customized to our requirements and they are managed in our package and distributed with it.

4. **WMS components:** software components developed entirely by WP1. The cross-dependencies among WMS components make it rather difficult to cleanly partition it as separate packages. However, we put some extra effort in identifying and separating our internal dependencies: at the end we were able to provide several WMS RPMs [11] and not just a single monolith.

These constraints specifically affect the semantics of the package configuration options, that must have the ability to scan and resolve the dependencies needed by either a single WMS component or the entire system.

This paper is structured as follow: section 2 briefly describes our usage of CVS, section 3 describes how we have addressed the aforementioned issues with the help of *GNU autotools* [3], section 4 is dedicated to the way we release and distribute our software using RPM [11] and LCFG [12], section 5 describes our modus operandi, section 6 describes the procedure before the release and the distribution of our software.

2 CVS and the EDG-WMS code

Given the project structure, as outlined in the section 1, CVS was adopted for early WMS development, and was later agreed upon as standard software repository tool for the project at large. CVS lets developers to separately modify a file and then keep track of the changes made by other people. Moreover, it makes possible to allow a programmer to have more than one version of the same code file.

CVS solves the problem of concurrent accesses to files by insulating the different developers from each other: each developer works in his own directory, then CVS helps him to merge the results when the work is done. In addition, this merge is done authomatically as long as it is safe, otherwise conflicts are clearly identified so that they can be resolved manually, preventing loss of others' works. Another interesting feature is the possibility to store each modified version of files. Doing this, it can be easy to intercept the right point in the development where a bug is introduced.

Within CVS branches are the way to keep track of different source trees containing very different versions of the same file. This is useful, for example, to when developing new features together with correcting bugs on the already written (and working) code.

The branching feature of CVS let us be able to create parallel trunks of the software, so that it is possible to modify and debug our code without affecting the main branch (called CVS HEAD). Due to the small time slots between two releases of the DataGrid software, we also add new software functionalities in *ad-hoc* created “branch”. Then, when needed, and after some stabilization period, such modifications can be merged to the main trunk. The way we have used to deploy our software using CVS is described in section 6.

CVS, however, is not a complete tool to develop code. It is not made to substitute any management model. It cannot provide good communication between developers. The use of other tools is needed in order to achieve those results. Besides the use of e-mails and other “classical” communication media, we have found greatly beneficial to maintain constant communication among developers via IRC (Internet Relay Chat). In addition, we found very useful to keep trace of open actions and bugs using Bugzilla (Bug Tracking System) [4].

3 Configuration of the EDG-WMS software

Currently, the DataGrid software builds and runs on just one architecture (RedHat Linux 7.3), but it will probably run on others in the future. This is one of the reasons for the generalised adoption in all software packages and components in DataGrid of the *GNU autotools* . Another useful feature of *GNU autotools* is to simplify building and distributing of source code programs: any of them may be built using a simple, standardized two step process (`configure, make`). There is no need to install any special tool in order to compile the code.

The *GNU autotools* also allow to cleanly address a specific requirement of the WMS, namely the configuration of individual components inside the same package, and the handling of numerous and diverse external or third-party packages and libraries.

Another reason of our *GNU autotools* choice lies in the fact that they can be easely used to obtain the configuration even of sub-packages. In addition, the handling of external or third part software and libraries is quite easy.

3.1 Source tree organization

The EDG WMS package is divided in smaller components. Each component represents some special functionality of the package, containing for instance daemons, libraries, test

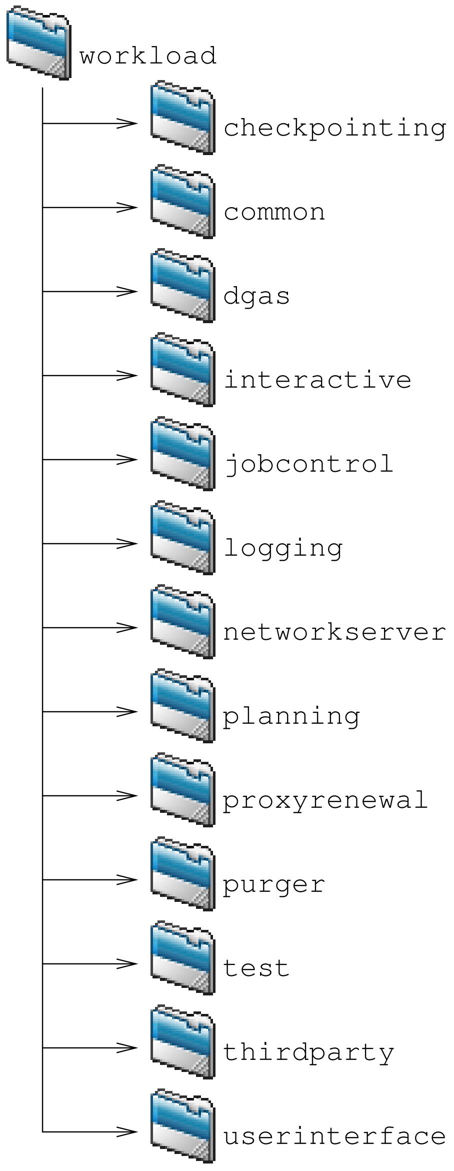

Figure 1: WMS Source Tree

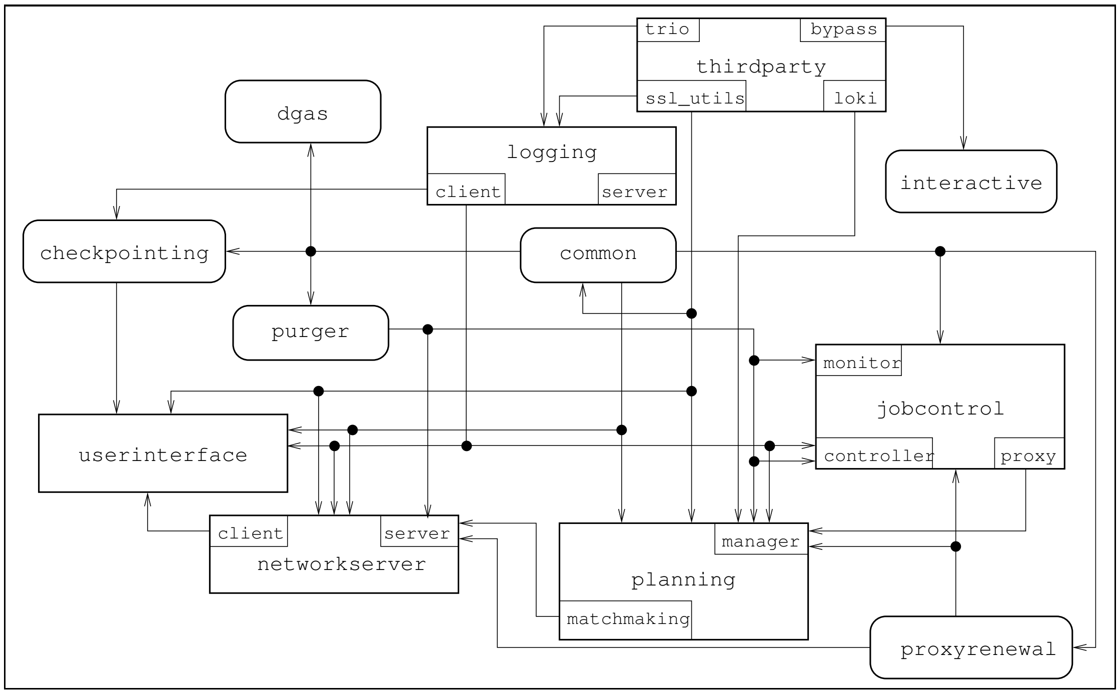

Figure 2: WMS dependency graph

programs, and documentation. The components show various levels of inter-dependencies (e.g. "common" components that are used by other components, components providing various services accessible via APIs).

The whole package is organised in a single directory tree, that can be downloaded from the CVS server and that can be transferred as a single archive file. Each component is identified by a sub-directory. The main directory is called `workload`. In Figure 1 there is a graphical rapresentation of the organization of the main components of WMS. Further levels are present inside each component sub-directory. These inner levels do not have a common structure, as each component may be quite different from the other ones: some of them are daemons, some others are simply libraries. The lack of a common structure makes even more difficult to have a simple and common build strategy for each of them. Clients of the daemons, for example, are claimed to be compiled even by different components than the current one. This is the main source of inter-dependency between components themselves and it has been the most difficult thing to achieve while designing our build structure.

3.1.1 Main components

The main components of `workload` are the following:

`checkpointing`:
: Libraries and header files for user APIs to the EDG checkpointing utility.

`common`:
: General purpose libraries and utilities used by all the `C++` and `Java` code.

`dgas`:
: Daemons, libraries and command line API for the EDG accounting system.

`interactive`:
: Support for Interactive job manipulation.

`jobcontrol`:
: Libraries and daemons for final submission and monitoring of the job status.

`logging`:
: Daemons, libraries, and header files for Logging and Bookkeeping service.

`networkserver`:
: Daemons and clients for secure communication between the User Interface and the WMS.

`planning`:
: The core of the WMS containing the daemons, and libraries for the best resource identification.

`proxyrenewal`:
: Daemons and libraries for user credential handling.

`purger`:
: Daemons and libraries for job space deallocation.

`test`:
: Test programs for all the WMS.

`thirdparty`:
: External packages (gridftp [6], bypass [7], SSL utilities [6], trio [8], loki [9]) modified to fit our need.

`userinterface`:
: Command line and `C++` APIs, Graphical User Interface for the access to the DataGrid .

In Figure 2, it is shown a graphical representation of the internal dependencies in the WMS. Each box (both squared and rounded) represents one of the forgoing components. Sometimes part of the internal structure of a component is shown with a thin box. The direction of the arrow means that the destination object depends on the source one, the dots show where two arrows are connected together.

The meaning of a dependency (arrow) may vary with the two sides of the dependency itself. For example, it may happen that a module depends on just some sort of client of the other one, or it may happen that some module needs to use some functionality of the other one. Component dependency may occur with various mechanisms: client/server, utility library, access API, etc.

For the sake of simplicity, in Figure 2 only internal dependencies among our modules are shown, while external ones are not.

3.2 Configure structure

Each dependency reflects on the way we configure the whole package. The main goal of our configuration system is to allow the developers of a single component to compile as less things as possible instead of compiling the whole package. The build of the entire package can take up to 90 minutes on a commodity PC. Also, not all external dependencies are required by every component. Capturing the complex component interaction pattern of Figure 2 has proven to be difficult, and in some specific cases impossible with the available tools.

The adopted solution is probably not the best one. We have felt the strong need of having some sort of *"autodep"* tool, that is some automatic way to express external and internal dependencies in a fashion similar to that of "common" autotools. This work has been started but not finished due to lack of time, and we hope that we will be able to provide a similar functionality for future releases.

In this situation, the "fast and furious" way has been to hardcode enabling options and conditionals directly into the `configure.in` file [13]. Each conditional was tied to a submodule (or part of it) and its (de)activation has been obtained with a large number of `''if''` statements hardcoded in the `configure.in` .

The effect of enabling some components or some of its parts was to allow the building of some programs/libraries or to include whole directory trees necessary for that part. The compilation of each component is enabled by default: it can be explicitly disabled using the appropriate configure option (`--enable-`*`submodulename`*`=no` or `--disable-`*`submodulename`*) or enabling another module that does not depend on it.

Each enabled submodule will also cause the checking for the presence of any other external package related to it.

3.2.1 *Example of enabling/disabling a submodule*

Here we show an example of how we allow the enabling of one of those components and how its dependencies are handled inside the configuration. We have chosen a "simple" one (simple as regard the number of dependency): the `proxyrenewal` (a standalone daemon taking care of obtaining renewed user proxies from an external service).

In the `configure.in` file, we add a conditional variable to guarantee its default abilitation:

```
...
opt_enable_renewal=yes
...
```

Together with it, we add the corrisponding `autoconf` [13] m4 macro in order to have the correct configuration switch in the `configure` script:

```
...
dnl
dnl proxyrenewal option
dnl
AC_ARG_ENABLE(renewal,
    [ --enable-renewal     build proxy
        renewal [default=yes]],
    enable_renewal=``$enableval'',
    enable_renewal=no)
...
```

Note that if the option is not given to the `configure` [13] script, the default value of the `enable_`*`submoduleoption`* is "no".

In the `configure.in` file, after the declaration of all the configure options and variables, there is one big ` ``if'' ` statement used to disable the submodules not explicitely enabled in the `configure` invocation:

```
...
if test ``x$enable_opt1'' = ``xyes'' \
    -o ``x$enable_opt2'' = ``xyes'' \
...
    -o ``x$enable_renewal'' = ``xyes'' \
...
    -o ``x$enable_optN'' = ``xyes'' ; then
    opt_enable_opt1=$enable_opt1
    opt_enable_opt2=$enable_opt2
...
    opt_enable_renewal=$enable_renewal
```

```
...
    opt_enable_optN=$enable_optN
fi
...
```

The net effect of this test is that if one or more options are enabled, only the corrisponding ``opt_enable_*'' variable is set to the value ``yes''.

Once all these variables have the right value, many tests are done in order to check what external dependency has to be tested for (see section 3.3). The values of the ``opt_enable_*'' variables are then copied to another variable:

```
...
have_renewal=$opt_enable_renewal
...
```

These have_* variables are then tested together with the conditions on external dependencies (see section 3.3) in order to understand which of the (selected) submodules have to be built:

```
...
if test ``x$have_myproxy'' = ``xno'' ; then
    have_renewal=no
    have_option=no
    ...
fi
...
```

This bunch of tests will result in having all the have_* variables set to ``yes'' or ``no''. They are then used to enable the appropriate automake [13] conditionals, using constructs like:

```
...
AM_CONDITIONAL(AMC_BUILD_RENEWAL,
    test x$have_ns_daemon = xyes \
    -o x$have_w_manager = xyes \
    -o x$have_renewal = xyes \
    -o x$have_logmonitor = xyes \
    -o x$have_controller = xyes)
...
```

In this example the have_renewal variable may also enable other conditionals, like AMC_BUILD_COMMON, AMC_BUILD_THIRDPARTY and others on which proxyrenewal depends. The system is made in a way that enabling another component depending on proxyrenewal enables all these conditionals too, together with any other one specific for that component.

These conditionals are used in the Makefile.am [13] files by the automake tool to understand which programs, libraries and subdirectories have to be included in the build process. In the main Makefile.am (the one in the workload directory) we will find something like:

```
...
if AMC_BUILD_RENEWAL
WL_RENEWAL = proxyrenewal
endif
...
```

Where the WL_* variables represent the subdirs to be included in the build process:

```
...
SUBDIRS = config m4 \
    $(WL_SUBDIR1) \
    $(WL_SUBDIR2) \
...
    $(WL_RENEWAL) \
...
    $(WL_SUBDIRn)
...
```

Then, in the Makefile.am relative to the proxyrenewal subdir we will find something like:

```
...
if AMC_BUILD_RENEWAL
sbin_PROGRAMS = edg-wl-renewd
bin_PROGRAMS = edg-wl-renew
noinst_LTLIBRARIES = libedg_wl_renewal.la
...
endif
...
```

That is, we will compile the edg-wl-renewd daemon, the edg-wl-renew command line and the (internal) APIs library in order to interface with the daemon.

3.3 M4 files utilizations

As previously mentioned, some of the packages the WMS depends upon come from the DataGrid project, other ones are externally provided:

DataGrid **packages**: `RGMA`, `Java Security`, `GACL`, `VOMS`, and `Replica Manager`.

Non DataGrid **packages**: `Globus` [6], `Expat` [10], `ARES`, `MySQL`, `MyProxy`, `ClassAd` and `ClassAdj`, `COG`, `CondorG`, `Boost`, `Java`, `Swig`, `Python` and `Perl`.

In order to correctly detect the presence and the position of such packages, it is necessary to create a large number of specific tests and set some variables. These tests may be put directly inside the `configure.in` file, but this would only make this file unreadable. So we decided to create a specific M4 [5] file for every package that doesn't already provide one.

These M4 files define just one macro called something like `AC_`*`PACKAGE`*. In general they take three arguments: the version of the package (when applicable), the action to perform when the right package is found, and the action when it is not.

The generic action of such macros is to check whether (and where) the include files and libraries are there (if the package is a library), or try to understand the path of some kind of executable (for example for `Perl` or `Java`). Sometimes they made both operations (for example for `Swig`).

The first kind of macros defines two (or more, in some special cases) `Makefile` variables, usually called *`PACKAGE`*`_LIBS` and *`PACKAGE`*`_CFLAGS`. The first one will contain the path and the name(s) of the library(ies), while the second will contain the path for the include files (if present). Some of these macros will also define C macros containing other information useful for the compiler.

The second kind of macros, instead, defines `Makefile` variables usually called `RUN`*`PACKAGE`*, which will contain the full path for the required executable.

Inside the `configure.in` file such M4 macros are used as follows (using the same example as of section 3.2):

```
...
if test ``x$opt_enable_w_manager'' = ``xyes'' \
    -o ``x$opt_enable_ns_daemon'' = ``xyes'' \
    -o ``x$opt_enable_logmonitor'' = ``xyes'' \
    -o ``x$opt_enable_controller'' = ``xyes'' \
    -o ``x$opt_enable_renewal'' = ``xyes'' ; then
    AC_MYPROXY([], have_myproxy=yes, have_myproxy=no)
fi
...
```

That is, when one of the components that depend on the `MyProxy` external package is enabled, the test for its presence is carried on. In this specific example, we were interested in this test when the `opt_enable_renewal` variable was set. In order to be sure that the value of the `have_*` variables is something like ``` ``yes'' ``` or ``` ``no'' ``` their values are set by default to ``` ``no'' ``` before actually performing all these tests.

4 Releasing and distributing the code

In the DataGrid project it has been decided to release and distribute the code using the RPM Package Manager. It provides extensive and accessible package management services.

We have had to follow simple DataGrid rules [16] and added some internal procedure to quickly check the WMS RPMs.

In the `configure.in` file, we add the M4 macro `AC_EDG_RPMS` which sets the directory where WMS RPMs must be built. Its default is `‘pwd‘`. We also add another M4 macro `AC_RPM` which defines the variables `RPM_LIBS`, `RPM_CFLAGS`, and `RPM_BIN_PATH`, used in the `Makefile` to build a simple code that reads spec files and returns the files that go in the RPMs.

In the main `Makefile.am` (the one in the `workload` directory) we will find something like:

```
...
rpm:  $(RPM_SPECS)
    make dist
    /bin/mkdir -p \
        @RPM_SOURCES_PATH@ \
        @RPM_SPEC_PATH@ \
        @RPM_BUILD_PATH@ \
        @RPM_RPMS_PATH@ \
        @RPM_SRPMS_PATH@
    /bin/cp -u @PACKAGE@-@VERSION@.tar.gz @RPM_DIR@/SOURCES
    for file in $(RPM_SPECS); do \
        /bin/cp $$file @RPM_SPEC_PATH@; \
    done
    for file in \
        @RPM_SPEC_PATH@/*.spec; do \
        rpm --define "_topdir @RPM_DIR@" -ba $$file; \
    done
...
```

So, it is enough to run the command `make rpm` to build WMS RPMs.

In our package we have organized our code in four spec files. The main one covers the main WMS services and APIs, two of them apply to a cople of external packages that needed modifications, the last one just includes the testsuite description.

We have added another target called `make rpm-check DESTDIR=<install location>`, considering that the WMS package is quite complex, and in order to avoid running the full time-consuming command `make rpm` when there are some errors in the package, e.g. a new header file has been added in the code but not included in the `Makefile.am` .

This target builds the program `checkfiles` which reads the spec files and extracts the list of files that goes in the RPMs, putting it in a file called `rpmfiles.tmp`. Then it runs the commands `make apidoc`, and `make install DESTDIR=<install location>`. The following piece of code produces two files: `installedfiles_notinthespecfile.txt` and `specfiles_notinstalled.txt`.

```
...
grep "^< " $(DIFFFILES) \
    | cut -d' ' -f2 \
    > specfiles_notinstalled.txt grep "^> " $(DIFFFILES) \
    | cut -d' ' -f2 \
    > installedfiles_notinthespecfile.txt
...
```

Once obtained such files someone can edit the file `specfiles_notinstalled.txt` to check if the spec files contain some old files. By viewing the file `installedfiles_not-inthespecfile.txt` it is possible to verify if we need to upgrade the spec files.

5 A bit of sociology

As already mentioned, WMS is developed by a group of persons working for different institutions in different european countries. This is not just addressed by an appropriate choice of tools, as described in the previous sections, but also by appropriate organisation. In our case, the role of the *packager* was introduced, with the task of organising the code tree structure, providing templates for the packaging of new components, overseeing on the application of project-wide rules [16] and on the uniformity of build procedures. While this role entailed being called to solve many build and installation issues for any component, it allowed all developers to converge towards common formats for the `Makefile.am` and M4 files, and a `configure.in` organized for all needed tasks. In addition, it allowed developers to concentrate just in code development instead of its organization.

6 Deployment procedure

A specific procedure was implemented to allow developers to keep committing changes to CVS HEAD without disrupting the test procedures related to the cut of new releases.

For each new release, the set of bugs and new features to be addressed (as extracted by the project bug-reporting system) is defined, and communicated via e-mail or on the IRC channel where most of the communication within the work-package WP1 is occurring. When all development issues are resolved (this is again known by interaction via

IRC), an e-mail is sent to the work-package WP1 mailing list, communicating the start time of the test session. Before this time, developers have to commit all pending changes with respect to the upcoming release. When the announced time arrives, a CVS branch called `test_<version>` is created. Software is entirely rebuilt starting from that branch and various tests are performed, including the execution of the work-package WP1 specific regression test suite. If errors are found, these are fixed and committed to the branch. When the test results are satisfactory, the release is tagged on the branch and all the applied fixes are merged to the main trunk.

7 Conclusion

We have summarised our experiences, the limits found and the extensions added to standard code management and packaging tools adopted by the EU DataGrid project, in order to accomodate the practical needs of the Workload Management work-package. The latter is characterized by a distributed development team, taking part in a centrally-managed project, and by the specific need to integrate a very complex set of dependencies, either external to the project, or external to the work-package WP1, or specific to the WMS component relationship. The tools that were used are CVS as repository, *GNU autotools* as manager of the building package, RPM as manager of the package. The missing functionality is mainly centered around the description of dependencies at the *GNU autotools* level, for which a dedicated, yet-to-be-developed "autodep" tool was felt to be useful.

8 Acknowledgements

The authors wish to thank to all the WMS team for their help and support in developing a better organization of the WMS package. We also thank the EU and our national funding agencies for their support of this work.

The DataGrid project is funded by the European Commission under contract IST-2000-25182.

9 Availability

EU DataGrid is distributed under a BSD-based license, its text is available at:

http://eu-datagrid.web.cern.ch/eu-datagrid/license.html

The source code can be downloaded in several formats from:

http://marianne.in2p3.fr/datagrid/repository/

References

[1] *The DataGrid Project.* http://eu-datagrid.web.cern.ch/eu-datagrid

[2] *DataGrid Workload Management Work Package.* http://server11.infn.it/workload-grid

[3] *GNU Manuals Online.* http://www.gnu.org

[4] *Bugzilla Bug Tracking System.* http://www.bugzilla.org

[5] *GNU m4.* http://www.gnu.org/software/m4/m4.html

[6] *The globus alliance.* http://www.globus.org/

[7] *Bypass.* http://www.cs.wisc.edu/condor/bypass/

[8] *Trio - portable and extendable printf and string functions.* http://daniel.haxx.se/trio/

[9] *loki.* http://sourceforge.net/projects/loki-lib/

[10] *The Expat XML Parser.* http://expat.sourceforge.net/

[11] *RPM Package Manager.* http://www.rpm.org/

[12] *LCFG (ng).* www.lcfg.org

[13] G.V. Vaughan *et al*, *GNU Autoconf, Automake, and Libtool.* New Riders, (October 2000)

[14] K. Fogel *et al*, *Open Source Development with CVS.* Coriolis Group, (October 2001)

[15] G. Avellino *et al*, *The EU DataGrid Workload Management System: towards the second major release*, 2003 Conference for Computing in High-Energy and Nuclear Physics (CHEP03), La Jolla, California, (24-28 Mar 2003)

[16] Quality Assurance Group, *European DataGrid Developers' Guide*, (2003)