$\mathcal{MLC}$++ Environment

Ronny Kohavi
Ronnyk@CS.Stanford.EDU

May 22, 1994

1 Introduction

The purpose of this document is to provide a basic introduction for the $\mathcal{MLC}$++ environment. Related documents include the $\mathcal{MLC}$++ dictionary (dict.*) that defines the terms used throughout the documentation and code, the $\mathcal{MLC}$++ coding standards (mlc-coding.*) that describes the specific $\mathcal{MLC}$++ headers and style, and the general C++ coding standards (cc-coding.*).

[Annotations are enclosed in brackets and printed in a small font. They elaborate the actual text, giving reasons and explaining some of the decisions.]

2 Overview of $\mathcal{MLC}$++ Directory Structure

The project is divided into a few subdirectories.

Root The main directory contains the Descrip.txt file which gives information about how to get the library, permission to use, *etc.* It also contains the "todo" and "longdo" files of things we should do in the short and long term, besides the enhancements described in the code itself.

db/ Databases used for testing.

doc/ Directory containing the documentation relevant to the project.

inc/ Include files. All ".h" files which users may want to include should be here. For technical reasons, ".c" files for templates must be accessed from the same directory, so they are symbolically linked to the actual files in the src/ directory.

src/ Sources. All ".c" files.

[This may be split into more directories according to the paradigms]

src/tests/ Test routines (testers). These are files which tests files in the src/ directory. Each tester has a ".c" file, a ".exp*" files (expected outputs), and optionally an ".cin" file for input.

bin/ Binary files and shell scripts.

The user `mlcadmin` owns the directories, and its home directory is this root directory. Some software is installed under root for convenience (*e.g.*, dot, dotty, C4.5). All files have group "mlc" and members of the $\mathcal{MLC}$++ project should belong to the mlc group in /etc/group. Users who are allowed to delete/add RCS files should be in the .rhosts file for mlcadmin.

3 Makefiles

Each directory that contains source code has a Makefile. The main entry should make sure everything is up to date in the directory. Other entries should be "scratch" which cleans everything and makes things from scratch, and TAGS which builds a TAGS file.

The Makefile in the src/tests/ directory has an automatic rule to transform a .c tester file into an executable test file. The rule deletes the ".out*" files (outputs of the tester), compiles the tester, and compares with the ".exp*" files (expected outputs). When a tester is run, its output is directed to the ".out" file which is compared with the ".exp" file. If there are other expected outputs, they can be termed ".exp1," ".exp2," and ".exp3" and the tester is expected to generate the appropriately named ".out*" files that match. Similarly, if a ".cin" file exists, input to the tester will be taken from this file.

If a tester relies on another tester for a "complete test" (*i.e.*, file A does not test everything and assumes file B does something for the class), then that should be reflected in the Makefile by having one source file "getrd" another (see 6). This is important since changing a class causes recompilation of the appropriate tester and we want both to be recompiled in such a case. An example is t_Attribute.c which relies on t_BagSet.c. Another example is that of an abstract base class (ABC) that cannot easily be tested. The makefile should use one or more of the derived classes to test the ABC as shown below:

```
t_derived.c : t_ABC.c
    getrd t_derived.c
```

The file t_ABC should **not** be in the makefile's list of files to compile.

A tester file should be created for the base class, even though it is not a real tester. The reason is that the automatic "put" mechanism tries to get the tester for the class if it exists. An empty file would be fine, but to avoid misunderstandings, include an error line as follows:

```
#error "Use t_<derived>.c"
```

4 Setting Up Your Environment Variables

$\mathcal{MLC}$++ uses a few environment variables defined below.

MLCDIR The directory where $\mathcal{MLC}$++ root resides.

CCFLAGS The compiler flags to use by default. All makefiles call CC with these flags.

GENTAG The TAG generator program to use. All makefiles call this program.

MLCPATH The path $\mathcal{MLC}$++ uses to open data files after failing to open them in the current directory. Points to src/tests/ by default.

[It is useful to change this to db/ when testing on actual datasets.]

DUMPCORE If defined, causes $\mathcal{MLC}$++ to dump core on fatal_error.

There have been many problems of not getting the desired core dump. Common reasons:

1. You have a limit on coredumpsize. Do "limit" from csh/tcsh to see what your limit is. It should be "unlimited."
2. You do not have permission to write the core. This is more complicated than it seems. You may have write permission to the directory, but the core file may not give you write permission. To see if this is the case, do "touch core" from the shell. If you don't have write permission, you may still be able to remove the core if you have write permission to the directory.

TEMP_FILE_NAME A template for instantiating temporary file names. This name should have 6 training Xs (see Unix mktemp()). Default is tmpXXXXXX.

The main $\mathcal{MLC}$++ directory contains a script called “setupMLC” which sets these to their default values, and updates the path to include some required directories. GENTAG defaults to Emacs tags (vi users can change this). To use setupMLC, add “source ~mlcadmin/setupMLC” to your .login.

The file “aliasMLC” sets up useful aliases. These include: CC (use CCFLAGS, links with $\mathcal{MLC}$++ library), and oc (objectcenter in motif mode). To use them, add “source ~mlcadmin/aliasMLC” to your .cshrc.

5 Setting your .emacs file

When reading existing files, emacs is forced into C++ mode, but when creating a new file, emacs defaults to C mode if the extension is .c or .h. The following two lines which can be added to your .emacs will change the default behavior. Note that you can always manually say “`M-x c++-mode`” or “`M-x c-mode`.”

```
(setq auto-mode-alist (cons (cons "\\.h$" 'c++-mode) auto-mode-alist))
(setq auto-mode-alist (cons (cons "\\.c$" 'c++-mode) auto-mode-alist))
```

ObjectCenter will open an emacs window to update source files. If you would like to use your existing window, you can redefined EDITOR to be `/local/bin/lemacsclient`, and add the following lines to your .emacs:

```
(setq load-path (cons "/local/CenterLine/lib/lisp" load-path))
(setq exec-path (cons "/local/CenterLine/sparc-sunos4/bin" exec-path))
(require 'clipc))
```

To form the connection with ObjectCenter, you need to execute the command `cl-edit` after ObjectCenter is running.

6 Revision Control

RCS (Revision Control System) is used to keep track of revisions. The user *mlcadmin* is the only one who can run rcs and change the RCS directories (adopted from the ci manual page). This is done to avoid mistakes. Commands which say “requires mlcadmin access” can only be executed by users who have their names in the .rhosts file of *mlcadmin*. These commands are all available in ~mlcadmin/bin which is added to your path by setupMLC.

The following commands are defined in setupMLC:

create Given a file, this commands create a new RCS file in the proper place in the $\mathcal{MLC}$++ tree. It also makes a read-only version in the appropriate place. Requires mlcadmin access.

get Get a file for update (no directory prefix is needed). The write-enabled file is created in the current directory and the file is locked so no one else can *get* it for updates.

To *get* many files, you can write a few lines of csh script like this:

```
foreach i (*.c)
   get $i
end
```

This is useful even if you will not update all files, because when you put them back, those that have not been changed will not get a new RCS version.

getrd Gets a file for reading. The file will be stored in the directory this command is executed from. "co" flags may follow file name. Useful for getting old versions (*e.g.*, `getrd error.c -r1.1`).

unget If the get was not required, this releases the lock. You do not need to have a copy of the file, and the local file is removed.

[The local file is actually copied to /tmp.]

put Put a file back after updating it. A description (log) is required about the change that was done. The local file is removed, but a read-only version is created in the appropriate place in the $\mathcal{MLC}$++ tree. If there were no changes to the file, *put* acts as "unget" (except that there must be a local file).

If the sequence "`@@`" or the sequence "`!fix`" is seen in the code, you will get a warning. One of these sequences should be used in your code when writing temporary comments that you intend to fix before putting the file back.

If *put* sees more than one file, it acts as "multiput," asking for one log description, and putting all files with the same description.

If any of the files do not contain a period (that is, they are a file stem), then *put* looks at the current directory for a .c and .h extensions, plus a t_ prefix with a .c suffix (tester). Each file found is then put with the single log description. This is the common case, since most fixes require modifying all three files.

Put will ask you whether to run "make" or not. You should usually say yes, unless you are storing many files separately and want to run make once. Whether you say "yes" or "no," put will do "getrd" to the appropriate files. If you say "no," you can go to src/ and src/tests and execute "make" manually.

showlog Given a file, it shows the full history log.

To show the log for specific dates, arguments may be given after the file name (see rlog). For example,

```
  showlog Attribute.c "-d>23-July-1993"
```

would show all logs after the given date.

delrevision Given a file and a revision number, this deletes the revision. If you delete the last revision, the file reverts back to the old state. Requires mlcadmin access.

showlocks Shows which $\mathcal{MLC}$++ files are locked by the user issuing the command.

lastdiff Given a file, it shows the difference between the current file and the last version stored in RCS. Useful when you have a locked file and don't know what changes you made.

rename Renames one file to another with all the RCS work involved. Does not update the Makefile, nor does it rename the .exp and .out files which have to be removed manually.

recompile Recompile all files containing the given regular expression *after* preprocessing using the makefile.

> This works by running CC -E, then *egrep,* and doing *getrd* to all the files that contain the regular expression. Every source file that is recompiled triggers a *getrd* for the matching tester if it exists. A make is then executed in the src/ and src/tests/ directories.
> Since a file that goes through the preprocessor includes `#include` directives, this is useful for compiling all relevant files after changing an include file which requires recompilation of all files that directly or *indirectly* include it.

break-lock Breaks the lock on a given file name. It asks you for a description of why you're breaking the lock, and e-mails it to the owner of the lock. After breaking a lock you can get the file in the usual ways. Please do not use this option much.

For small changes, it is better to change the .c file in the directory without updating the RCS, and sending e-mail to the lock owner about the change. The lock owner should then make the change before storing the file back.

historylog Show a log of updates from a given date in all of $\mathcal{MLC}$++. This is useful to review code before meetings. The syntax is `historylog "Wed 4-Aug-1993 12:00-LT"` and it is usually useful to pipe this to a file.

[It takes a LONG time to run so be patient.]

[Note that RCS stores the time stamp in UTC (Coordinated universal time) which is 7 or 8 hours ahead, depending on daylight saving. This time stamp appears in the log files. See "`man co`" for more explanations.]

7 Setting up ObjectCenter

When working with ObjectCenter, a few variables must be setup to allow default includes to look in the $\mathcal{MLC}$++ directory, and to link with the library. In your ~/.ocenterinit file, please insert the following line:

```
source /u/mlc/ocenterinit
```

The $\mathcal{MLC}$++ ocenterinit file sets up a few defaults:

1. Some errors which are know to be ObjectCenter bugs are suppressed. Suppression of errors is done at the line level (*i.e.*, the same error will be shown if it happens in a different file or in a different line number).
2. The default include directory points to the $\mathcal{MLC}$++ include directory.
3. The $\mathcal{MLC}$++ library file is loaded.
4. A few aliases are defined, the most important are:

   r run

   cdmlc Moves to the $\mathcal{MLC}$++ source directory.

   cdmlci Moves to the $\mathcal{MLC}$++ include directory.

   memcheck Start memory leak checks.

   nomemcheck Stop memory leak checks.

   reloadlib Unloads and reloads the library. If you need to load a source program in source code, but have already linked with the library, you will usually get an error message about objects being defined twice. This call unloads the library and reloads it (but does not link). You can now load your source file and run will relink. If you loaded a source file without doing `reloadlib` and got errors, you can just execute `reloadlib` and run, because the source file will be reloaded automatically.

A famous problem with ObjectCenter is that line numbers do not appear in the source window. If this is the case, remove the resource setting for borderwidth in your .Xdefaults or .Xresources.

8 Mailing Lists

Two mailing lists are setup for $\mathcal{MLC}$++. `smallmlcpp@cs` sends mail to the people working on $\mathcal{MLC}$++. It is intended for discussions of specific problems, sometimes in detail.

The second mailing list, `mlcpp@cs` sends to `smallmlcpp@cs`, but also to other people interested in $\mathcal{MLC}$++.

9 Libraries and Changing Header Files

9.1 Making Major Changes

Usual changes are first done locally in the user's directory, and then when everything compiles well and tests, they are transferred back to the $\mathcal{MLC}$++ directories.

In some cases, a change must be made that requires recompilation of a large portion of the $\mathcal{MLC}$++ library (*e.g.*, adding a member to a basic class like LabelledInstanceBag).

Note that adding a virtual function to a class changes the size of the virtual table and requires recompilation of every function that uses this class (unless the member is the last one). Similarly, changing the order of the virtual functions requires recompilation of all functions.
In such cases, it is sometimes best to add the member as non-virtual and change it to virtual late at night, or add the function as the last member. The problem with these two approaches is that people forget to make the final change before the night job.

Suppose a header file is changed and it requires recompilation of many files. Instead of interrupting everyone, the following procedure should be used:

1. Type "tmplib" from the csh. This alias changes your default library and changes your compilation flags so that header files will be searched in tmpinc/ before inc/.
2. Copy your header files to tmpinc/.
3. Build the temporary library from scratch by executing "make scratch" in src/. Remember that this removes all the .o files in src/, but does not affect the current library.
4. Test your change. You may make changes, run "make" (both in src/ and in src/tests) and everything will use the temporary library.
5. When the change is complete, warn everyone that you are renaming the main library (preferably when few people are working). *put* the header files (and all other .c files), effectively storing them back to the inc/ directory.

6. Type “renlib” from the csh. This alias will rename the temporary library to the active one.
7. Alternatively (not as nice) is to do “orglib” and execute “make scratch” from the *root* directory (~mlcadmin). This will compile the library from scratch and execute all tests.
8. People can start working once the library is complete (before all tests are executed), but they should erase their ptrepository and ocrepository files and recompile all object files in their directories. Also, template files may be generated.

9.2 Fast Library

When fast performance is required, the library can be compiled without all debug code and assert code.

To compile the library this way, type “fastlib” from the csh, and “make fastlib” in src/. The alias “fastlib” changes your default library and compiler flags. After a fast library has been created, you can use it simply by typing “fastlib.” The alias “orglib” returns you to the original library and original compiler flags.

Note that once a fastlib has been created, you can regularly link to it by doing fastlib before your compilation.

9.3 Profiled Library

A profiled version of the fast library can be generated by typing “proflib” and “make proflib.” The behavior is analogous to fastlib. After you execute the executable, do “profile a.out” (or whatever executable you have) to get the profiled results.

If you see spontaneous calls which are important, they may come from libraries compiled without the -pg appropriate flag. For GnuLib, you can simply link with `$MLCDIR/CClibg++/pgsrc/*.o` which will link to a version of GnuLib which is profiled.

9.4 Testing Templates

Templates are hard to work with and require a lot of experience. If you have a problem, the first thing to do is to remove the ptrepository and ocrepository (our repository for objectcenter) and recompile. Sometimes, especially when control-c’s are pressed, the repository gets corrupted.

If you are modifying a template, you must be careful not to link with the regular library which contains many instantiations of templates, because the library will be preferred over new instantiations. To solve the problem, do

```
setenv MLCLIB ~/mylib.a
(cd $MLCDIR/src; make lib)
```

This will create a library without any template instantiations.

[Note that if you do not execute `setenv MLCLIB` to some name, it will act on the actual library, with some possible side effects to your life from other users who will need to instantiate dozens of template files.]

Clearly, if you wish to actually store the template, you should do `make lib` on the real library, and immediately compile some testers that will create template instantiations. Generation of some common template instantiations make be done by executing `make rep`.

10 System Administration

This section may be skipped by "users" of $\mathcal{MLC}$++. It explains how users can be added to the project, and other system administration chores.

10.1 Adding a New User

To add a new user do the following:

1. Open an account, assign quota.
2. Add the login name to `/etc/group` on all machines.
3. To give permission to update RCS files, do
   ```
   rcs -a<login> files
   ```
   In general, files should be `*.c in src` and src/tests/, `*.h` in inc/, and `u_*.c` in util/.
4. If the user is allowed to create/delete files, add the name to the .rhosts of mlcadmin and to the `MLCUSERS` environment variable defined in mlcadmin's .login.

10.2 Global Replaces

In case a global replace is needed, you can do the following:

```
foreach i (*.c)
   get $i
   mv $i /tmp
   sed -e '1,$s/oldword/newword/g' /tmp/$i > $i
   put $i <<!
Changed oldword to newword
.
n
!
end
```

Known problems:

1. If a question is asked by "put," it is answered as "n" and usually does not cause the files to be stored. This is common for the `@@` signs.
2. In the header directory, don't forget to run this with *.h, not *.c. If you run with *.c, relink the template files (*e.g.*, Array.c and Array2.c from src/)

10.3 Daily Execution and Log

˜mlcadmin/bin/daily is executed every day early in the morning (currently 6:30AM). The script recompiles everything in $\mathcal{MLC}$++. The results of this job are stored in ˜mlcadmin/daily.log. Errors in the compilation should be preceded by three asterisks. These lines are then mailed to the users in mlcadmin/.forward.

10.4 Aliases vs Scripts

Users must execute `setupMLC` in their .login, and `aliasMLC` in their .cshrc. The `setupMLC` script adds bin/ to their path, giving them access to many scripts like *put, get.* The `aliasMLC` script gives them aliases which cannot be done in a script because they change environment variables. These aliases should be kept to minimum, as they incur an overhead for every script executed.

10.5 Code Review

To prepare a code review, the script bin/codeReview can be used. It adds line numbers, and output nice enscripted postscript. Usage is

```
codeReview tmp/codeReview0209.ps file1 file2 ...
```

The first argument is the output file and should be an absolute path name, or it will end in /tmp. If the files do not have a directory prefix, the header, source, and test file will be printed.

Code reviews may also include historylog output, but that is usually read on the screen due to the humongous amount of output with little information.

10.6 Troubleshooting Guide

Here are common setup problems that you may encounter.

MLCC MLCC sometimes doesn't work. Specifically, running make in the $\mathcal{MLC}$++ directories gives `MLCC: command not found`.

The problem is that your `.cshrc` does not execute `source aliasMLC`. Most common reason is that you have a statement such as `if (! $?prompt) exit(0)` above the aliasMLC.

Make / MLCC I wrote a Makefile, but it doesn't understand MLCC, or I'm getting strange make messages.

Make sure you have `SHELL=`usr/bin/csh/ at the top of the makefile. Since MLCC is an alias, you must enclose it in parenthesis so that make will run it inside the shell.

E N D