The Art of Computational Science

The Kali Code

vol. 7

Code Management: Libraries, Versions, and Compatibility

Piet Hut, Jun Makino

September 13, 2007

Contents

Preface

In this volume, Alice and Bob will search for ways to organize their ever growing body of codes. They hit upon a library structure, and a way to let codes of different ages include different versions of library files with the same name.

We hope to write this volume some time during 2005.

0.1 xxx

We thank xxx, xxx, and xxx for their comments on the manuscript.

Piet Hut and Jun Makino

Chapter 1

Introduction

1.1 xxx

xx

Chapter 2

Memo for code management book

I wrote:

Okay, well, so we should have, for each volume directory

- some way to specify which files are commnds
- some way to specify which files are libraries

And, we need a way to specify

- directory to install commands
- directory to install libraries
- (checking stuff I wrote above)

Hmm, well, this means that we need some startup or initialization script (sourced in .cshrc), which add things like environmental variables and command search path.

Shall I implement some prototype?

Piet wrote:

Yes, please! And in our volumes, we can describe this either in volume 3b (user interface) or in volume 4 (ACS data format). That way, I can then start writing volumes 3b,4,5,etc, using this new convention. We could also write a new volume about it.

What do you think is best: put it in volume 3b (user interface) or in volume 4 (ACS data format), or in a new volume (libraries and code management)?

We need to add: some more in makefile. Current makefile has:

```
SRC =  title.ok pref.ok  ch01.ok  ch02.ok  ch03.ok  ch04.ok \
       ch05.ok  ch06.ok  ch07.ok  ch08.ok  ch09.ok  ch10.ok litt.ok

TARGET = volume1c

include ../etc/Makefile.inc
```

One question is if it is better to use rake, but perhaps better to start with something we are used to.

I guess what is necessary is:

```
 RUBYLIBS = ...
 RUBYCOMMANDS = ...
```

This means we need to specify the directories to copy these.

First, library:

The standard is what is in $LOAD_PATH. I think, however, it is better to add the ACS specific location to this $LOAD_PATH, than to the contaminate standard place with ACS-specific stuff.

So, the simplest solution is $ACSROOT/rubylib.

The name RUBYxxx is not ideal (some things may be csh/bash script or something completely different). Better to have something like

```
  LIBSRCS = ...
  SCRIPTSRCS = ...
```

For the time being, I'd just put things at

```
 $ACSROOT/lib (referred to as ASCSCRIPTS)
 $ACSROOT/bin (referred to as ASCLIBS)
```

One question: Where to put the files which are used to keep the makefiles and others working. If Makefile.inc uses some Ruby scripts, the scripts themselves should live in some volume directory. On the other hand, it seems better to have them in the above ACSSCRIPTS directory, to keep thing simple.

If both files are under svn, this may be a bit inconvenient, since svn would think the files are updated each time source file is copied to the target directory. To avoid this, I simply run "diff" before actually updating file.

Further question: What should be defined as environment valiables and how?

2.1 Much more memo on code management

We want to say, for example,

```
require "acs"
```

and let the system somehow cleverly figure out which version of, for example, clop.rb to actually load. So we have the following questions:

1. How to maintain multiple version of clop.rb and where.
2. How an program specify which version to include.
3. What we do if we want to change some ”old” version.

This is clearly not something easy to explain. ”Solution” we ended up is:

We say something like

```
acsdate "20050301"
require "acs"
```

and clop.rb, in the library directory, has all old versions like

```
clop20040103.rb
clop20041001.rb
clop20041203.rb
....
```

in the directory, well, something like clop_rb

Then, if we say require ”clop”, it would look for the newest among the ones older than the date specified.

Hmm, sounds very complicated....

2.2 Programmer interface

Anyway, we need,

- The way to refer to an include file, which will be

  ```
  acsdate "20050301"
  require "acs"
  ```

Perhaps better not to overload standard "require" method, since that would be too confusing. Also, for the first file (here acs.rb), we cannot use our new method, since that new method is not defined yet.

- The way to install a new version of a library. I think it should just be

  ```
  make libs
  ```

  With the Makefile which for example look like:

  ```
  LIBSRCS = clop.rb
  ```

  That is, the same as the initial version I made.

  make libs will check if the already installed version is newer or not, and if it is older, install the current version in the volume directory to the library directory.

So far, these are implemented.

2.3 Programmer's manual

2.3.1 What is there?

This system offers a way to refer to library files (at least for Ruby library files) with the date of the file it made as **required** argument.

In this way, one can add/remove features or change the behaviour of one library, without worrying about the possibility that changes made will break the old programs which use that library.

The problem here is that there are still things that might change, like the Ruby interpreter itself or C compilter or Operating System. We will need to address these in some future, but not today.

2.3.2 How it works?

Exporting library files

In any (vol) directory, you can export (ruby) libraries by using command

```
check_and_install.rb
```

The standard way to use it is just to add an entry to the makefile like:

```
      LIBSRCS = acsrequire.rb acs.rb
```

Then Makefile.inc in etc directory will run

```
        ruby -S check_and_install.rb ACSLIBS $(LIBSRCS)
```

The first argument is the name of the environment variable (not the value itself) for the name of the directory to store library files, and everything after that are regarded as the file names to install.

If we have

```
   ACSLIBS = /home/foo/acs/lib
   LIBSRCS = aaa.rb bbb.rb
```

check_and_install.rb does the following things for each of files in LIBSRC. Here, we call aaa.rb

1. First, check_and_install.rb looks if there is a file

   ```
   /home/foo/acs/lib/.sourcenames/aaa.rb
   ```

   If the file is there, then it contains the full path name of the file aaa.rb which was copied into /home/foo/acs/lib/aaa.rb in previous time check_and_install.rb was used to install aaa.rb. If that name is different from the full pathname of aaa.rb, this command aborts. If the names are the same, or if the file under .sourcenames does not exist, goes to step 2 (it creates the file under .sourcenames if it does not exist)

2. It compares the original aaa.rb to /home/foo/acs/lib/aaa.rb. If

   ```
   they are the same, there is no more thing to do. If they are
   different, it copies the original aaa.rb to
   /home/foo/acs/lib/aaa.rb, and then to
   /home/foo/acs/lib/aaa_rb/YYYYMMDD_aaa.rb, where YYYY, MM, DD are
   the current year, month and date.
   ```

Loading library files

In order to use this versioning mechanism, you need to load the library file "acsrequire.rb", using usual load or require. If you load "asc.rb" in standard way, "acsrequire.rb" is also loaded.

Once acsrequire.rb is loaded, a top-level method acsrequire is defined, which you can use as

```
    acsrequire "aaa"
```

You should omit ".rb" for acsrequire.

By default, acsrequire behaves similarly with standard "require", loading the newest version. To specify the version date, you need to write

```
   $acsdate = "20050401"
   acsrequire "aaa"
```

Then, acsrequire tries to find the most recent aaa.rb, which is older than 20050401.

If the current directry has aaa.rb, and if current directory is in the environmental variable RUBYLIB, that one is used.

In order to find "acs.rb" in the first place, you need to define RUBYLIB appropriately. This is currently done by sourcing the file

```
  acsstart.cshrc
```

In the directory $ACSROOT/bin. So you need to source this file in your .cshrc or whatever file to use ACS as

```
   set ACSROOT=/home/foo/acs
   source $ACSROOT/bin/acsstart.cshrc
```

Hope this is understandable...

nil

Chapter 3

Four Ways to Invoke Ruby

3.1 standard way:

In order to run a program `some_code.rb`, type

```
ruby some_code.rb
```

This will work for running files that are present in the current directory.

The drawback is that invoking `ruby` does not give you access to files that are installed in the source libraries such as `$ACSROOT/bin` and `$ACSROOT/kali/bin`.

3.2 the acs and kali commands:

As an abbreviation for `ruby -S`, we have introduced the alternatives:

```
acs some_code.rb
```

and

```
kali some_code.rb
```

The command `acs` includes a search for files in the general source directory `$ACSROOT/bin`. In addition to that, the command `kali` also includes a search for files in the kali specific source directory `$ACSROOT/kali/bin`.

Note that, when using either of these two commands, you can leave out the `.rb` suffix. The two commands

```
  acs some_code
```

and

```
  kali some_code
```

give the same results as the two commands listed above.

3.3 executing the command directly:

In most cases, the Ruby files have been made executable (and if not, you can make them executable by typing `chmod +x some_code.rb`). You can then run a code by directly type its file name:

```
  some_code.rb
```

If you want to run files in the source directories in this way, you have to add the appropriate directories, other than `$ACSROOT/bin` (such as `$ACSROOT/kali/bin`) explicitly to your command search path.

3.4 using a wrapper:

Another way to execute the file `some_code.rb` is to write a one-line shell script, with the name `some_code` and containing just one line:

```
  kali $*
```

which can then be invoked simply as

```
  some_code
```

The presence of `$*` in the one-liner will guarantee that all command line options will be passed on correctly, as in:

```
  kali some_code --help some_option other_option
```

3.5 Conclusions

The first way doesn't work for files in source libraries. The last two ways are dangerous, in the sense that they introduce the name of all our Ruby files into the global name space, where they can easily collide with other packages. Therefore, our prefered way to run Ruby files in the Kali project is to use the second way:

```
  kali some_code
```

Note that the command `kali` can be used everywhere where one can use `acs` or just plain `ruby`.

Note, however, that there are still occasions to use `ruby` instead of `kali` in some situations. For example, when you are editing the file `some_name.rb` in a particular directory, and you want to test it out, you want to run it locally, ignoring the older official version in the kali source directory. In this case you can type:

```
  ruby some_code.rb
```

However, you can also type:

```
  kali some_code.rb
```

or alternatively just

```
  kali some_code
```

since the `kali` command is guaranteed to first look in the current directory for a file to execute.

So it is up to the user, which one to use. If you don't want to think about what file is located where, you can always use the `kali` command, since that is always guaranteed to do the right thing. However, if the `ruby` command can find the intended file (and the required files specified by `require`) it would be perfectly fine to use the standard `ruby` command instead. And if you do not mind the potential hazards of using the global name space, you can of course use the third or fourth way listed above.

nil nil nil nil nil nil

Chapter 4

Literature References

[to be provided]