Source code management tools for ALADIN

Filip Váňa, Tomáš Kalibera,
Martin Janoušek
CHMI

Why SCM tools?

- keeping track of the code changes
- enhance communication between developers
- make code versions accessible to all users in a user-friendly and transparent way
- support to geographically distributed teams
- build tools (using make, dependencies)
- do the labor work automatically and let scientists focus on their work

Ideal development environment:

- general and easy-to-learn SMP tool
- automatic building tool
- integrated source-code-level debugger
- analyser
- support team

- difficult to achieve (distribution of tasks to different machines)

SCM systems for ALADIN

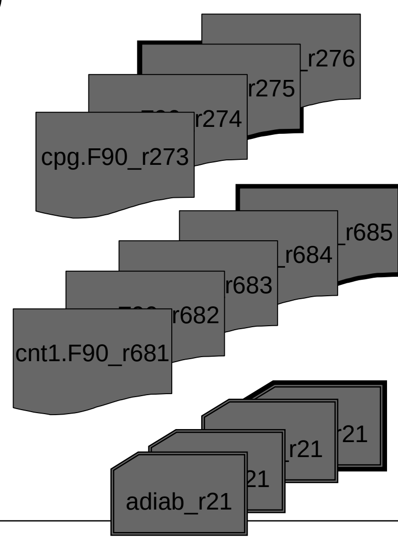
- ClearCase (+TUC scripts and rules)
- CVS (+CVSTUC scripts and rules)
- filesystem + recursive Makefiles (domake)

ClearCase

- Rational Software Corp. (www.rational.com)
- ClearCase: configuration-management system
- files and directories with their versions are versioned objects
- versioned objects are stored in VOB (Versioned Object Base)

ClearCase (2)

- A **branch** is an object that specifies a linear sequence of versions of an element
- A **view** is a selection of a specific version of each object, defined by view configuration specification
- multi-version file system (mvfs)
  - save disk space
  - copy of file only on-demand for modification



ClearCase (3)

- for small teams expensive and hard to manage

Transparent Use of ClearCase (TUC)

- a set of scripts
- an interface between developers and a subset of ClearCase
- easier to learn
- naming conventions for branches and views (branches=views with writing permissions)
- simple dependency resolution (cc_depend – generates a correct order in which modules can be compiled)
- developers are restricted only to use TUC => easier for the administrators to keep the database (VOB) in consistent state

SCM for RC LACE Prague Team

- CC complicated and expensive
- first use files + domake
- difficult to keep track of work
- a simple RCS-based SCM (Váňa)
- complication of cross-compilation
- migration to Linux: CVS-based system (Kalibera, Široká, Váňa)
- CVSTUC
- dependency resolution and GNUMake

Concurrent Versioning System (CVS)

versioned file, revision, branch

+ free, open, many free clients, utilities, tutorials
+ highly portable
+ large open source projects
+ trivial management, simple use

- no "private branches" (ACL support is work in progress, currently only filesystem-based security)
- disk space consumption per user/cvs module (force user to *commit‘n‘delete* work files)

CVSTUC: a CVS system for ALADIN

- large projects can be managed with CVS
- the most commonly used TUC features can be easily mapped to CVS => **CVSTUC**
- general mapping problems
  - no views (only full checkout of a branch)
  - no stable CVS filesystem (project SourceForge: CVS virtual file system for Linux)
- new branches can be created only from a public branch
- lack of ACL => incidental overwriting of other users‘ branches possible

CVSTUC – how it works

- a **view** is a directory tree
- public sources checked out to a directory
  - cc_getview ("cd")
- user views in subdirectories
  - cc_getview ("checkout or update, cd")

CVSTUC commands

- cc_init [-p project]
- cc_getview [-p project] [-r release | -r all] [-b branch] [-u user]
- cc_add [-f file] [-i import] | [-d directory]
- cc_edit -q [-k] -f filename
- cc_list [-u user] [-b branch] [-r revision]
- cc_diff -h -f filename [-w] [-x]
- cc_depend [-f singlefilename] [-l listfilename] [-b branchesToBuildWith] -d directory
- cc_popul -d directory

CVSTUC: a dependency resolution support

cc_depend (can be used without CVSTUC)

- dependencies of projects are stored in databases
- hierarchy of dependency databases
- merge support
- support for generation makefile dependency info files
- extensible

Managing Software Dependencies: Compile-time dependencies

- modules, headers (+recursive)
- modules and includes can be in any directory
- source files not named as the module
- external dependencies
- complex F90 commands (like module procedure) – a real F90 parser necessary
- the dependency transitive closure for compilation
  two transitive closure lookups:
  - is_used_by
  - uses

Creation of project dependency caches

- a huge perl hash
- dependencies: file1 [uses,is_used_by] file2
- generating info:
  - project name
  - project source directory
  - modules directories
  - headers directories
  - map of modules (module name --> source file)
  - merge info
  - names and source directories of the projects
  - externals (modules and headers)

Merging dependency caches

- dependency of 2 projects that use each other resolves externals
- as if the sources were in the "same project", but merge operation is faster than rebuilding dependencies
- no need to access the sources
- (experience note: a shell is not fast enough!)
- the dependency cache is invalidated whenever there is a file newer than the cache
- the cache build time short => no need of incremental cache updates

Building ALADIN with GNU Make

- make is an expert system
- do not rebuild too much, but make sure that everything that has to be rebuilt is rebuilt
- parallel make
- other possible optimizations (compile as much as possible, etc.)
- used for open source + commercial projects
- time-tested

--> **GNU Make** -- free, tested, many features, portable

Dependecy Information Makefiles

- can be pre-generated from CvsTuc dependency caches (tremendously speeds up build process)
- included into makefile(s), auto remake
- modular approach (directories in rules)
- fast: it takes approx. 15 s for GNU/Make to load the dependencies (on 1 GHz PIII)

Project Makefiles

- pattern rules with directories
- project specific compile options
- compile functions for included files

How it works

- cc_depend:
  - generate dependency caches for each project
  - merge caches between projects
  - create dependency information makefiles
  - checkout projects makefiles
- GNU Make
  - loads dependencies, compile (and link)

(Coding) Conventions

- needed for source code management, build, viewer and other tools
- make code readable for humans
- some hints
  - do not copy files between projects, use better dependency/build tools instead
  - do not include system headers into sources (mpif.h...)
  - make names of modules and headers unique within project, include headers without directories
  - place modules into distinct files named same as the module (tools can handle this, but it takes cpu time and is not nice anyway)
  - propagate platform specific ports to source code

Future Development

- support more TUC commands like cc_rmlast...
- support hierarchical branching (still not a full *view* support)
- separate precompiled objects from linking process
- share precompiled modules
- use sandbox GNU Make capability to simplify and speed up building process
- rewrite all code to perl
- a good Fortran 90 parser wanted