Introduction

- What is CMT, its goals
- Operating CMT
- The concepts in CMT, the internal model
- Status, implementation
- Comparisons, references

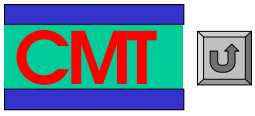
Configuration Management Tool

What is CMT

- A set of tools and conventions
  - structures software development or production
    - concepts of areas, packages, versions, constituents
  - organises software into packages
  - describes package properties
  - describes package constituents
  - operates the software production (management, build, import/export, etc...)
    - by transparently configuring and driving the various conventional tools (CVS, make, MSDev, Web, tar, compilers, linkers, archivers, etc...)

The goals

- Organise software development from a single person up to teams in large projects (*scalability*)
- Organise relationships between entire software bases or between simple packages (*scalability*)
- Manage site, platform or product specific properties
- Automate complex production tools for non software experts (*simplicity*)
- Factor out know-how from project to project, from package to package or from team to team (*process improvement*)



Operating CMT...

①Define software areas –> *define* **CMTPATH**

- private individual work areas
  - primary development
- shared team-wide development areas
  - integration phases
- public production areas
- export areas
- import areas for external software

```
> setenv CMTPATH area1:area2:area3
```

```
HKEY_LOCAL_MACHINE/Software/CMT/path/...
```

...operating CMT...

② Describe and parameterise the configuration of packages –> *fill in the* **`requirements`** *text file*

- relationships with other packages `use Expat v1`
- constituents `library A *.cxx`
- meta-information (`author`, `manager`, ...)
- private or public (exported) properties and tool configuration (macros and symbols)

```
macro cppflags "-g" ...
path_append LD_LIBRARY_PATH "" Linux "..."
```

...operating CMT...

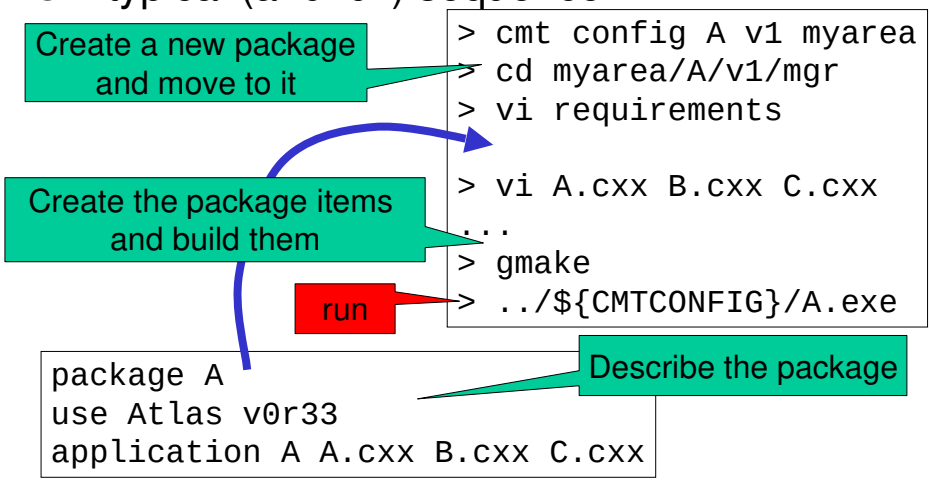
③ Drive the various productivity tools while monitoring the package state –> *run* **`[j]cmt`**

```
> cmt show path
> cmt show uses
> cmt show constituents
> cmt show macros
> cmt show macro xxx
```

- CVS, `make`, `MSDev`, `tar`, `Web`, etc... get their configuration from the `requirements` file (through the `cmt` generic driver)

...operating CMT

④A typical (and full) sequence



The concepts...

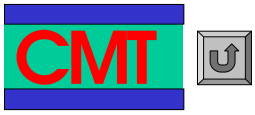
Area	Package search path
Package	The minimal entity
Version	Describes the backward compatibility
Use relationships	Induces inheritance-like properties and version constraints
Constituents	Applications, libraries, documents
Configuration parameters	Macros, environment variables, include search path, etc...
Document generators	Open architecture for any non-standard configuration action

The packages

- Plain standard packages
  `<some area>/<package>/<version tag>/<mgr>`
- Stand alone (*unstructured*) packages
  - simple test applications. Can use any standard area but cannot be used by other packages.
- Glue packages
  - install external software into the CMT conventions
    `LHC++`, `Geant4`, `Objectivity`
- Interface packages
  - generic configuration parameters to a set of correlated packages
    `Simulation`, `Reconstruction`, `Graphics`, `Atlas`

The version tags

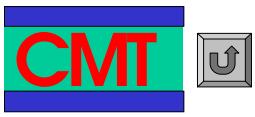
- Handle backward compatibility specifications
  v`<major id>`**r**`<minor id>`**p**`<patch id>`
  Any letter is possible
- Alternate version strategies available
  - best-fit (*the default*), first-choice, last-choice, keep-all
- For the default strategy, the best possible set of versions available from the search path is computed.
  - Overrides are acknowledged
  - Unresolved conflicts are fatal (two different major ids simultaneously referenced). Requires user's action (package synchronisation)



Use relationships

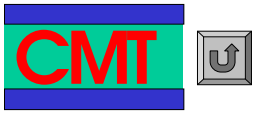
- Set a dependency between two package-version duets.
- Public configuration parameters are inherited through the use.
- The use specification defines a graph of links (possible redundancy)
- `cmt` provides the reduction algorithm of the tree (search path is taken into account)

```
> cmt show uses
```

Constituents

- applications and libraries
  - standard behaviour for make or MSDev parameterised by conventional macros and fragments (make & MSDev)
    - cflags cppflags fflags
    - clinkopts cpplinkopts flinkopts
- documents
  - can be fully tuned and specified for any document filter
- fragments and macros overriding can occur in any client package



Symbols

- Used to specify make macros, environment variables, aliases, path-like variables

```
macro a "aaa"
```
```
alias c "ccc"
```
```
set b "bbb"
```
```
path d "ddd"
```

- alternate value can be specified for different sites, platforms, working conditions

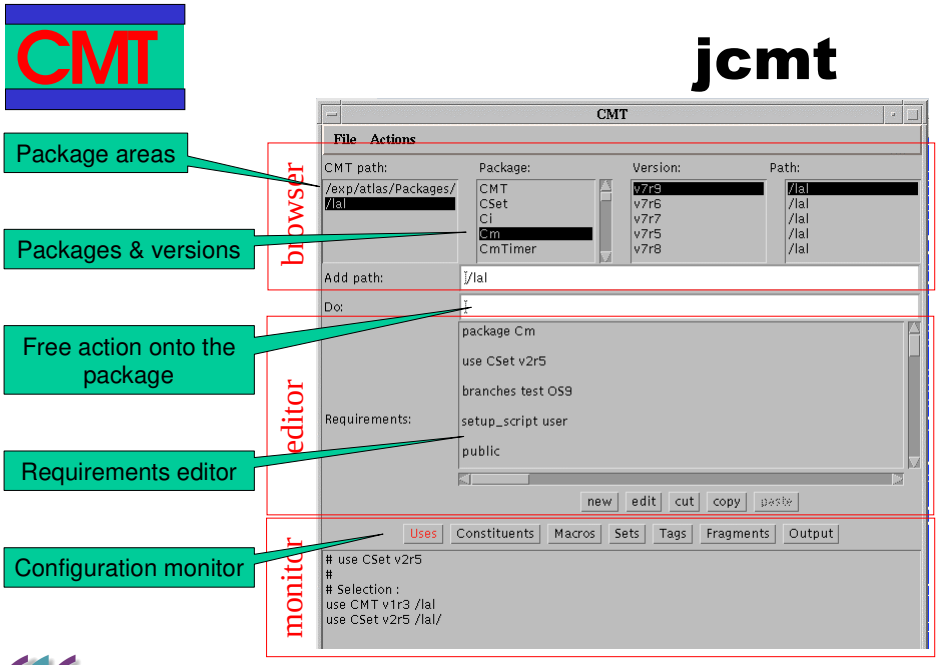
```
macro a "aaa" CERN  "aax" LAL "aay"
macro b "bbb" Linux "bbx" alpha-osf40 "bby"
macro c "ccc" debug "ccx" insure "ccy"
```

- edition

```
macro-append        macro-prepend       macro-remove
set-append          set-prepend         set-remove
path-append         path-prepend        path-remove
```

Implementation

- The main driver
  - `cmt` line-mode driver (Unix & Windows)
  - `jcmt` interactive java driver
- ported to all Unix platforms and native port to Windows 95/98 and NT (only plain C++)
- interfaced to MSDev through calls to `cmt` from the customisation menus
- **`http://www.lal.in2p3.fr/SI/CMT/CMT.htm`**
  - documentation, distribution kits, mailing list, CVSWEB



CMT

Clients

- Production
  - Virgo (historically the first!)
  - LHCb
  - Auger
  - Nemo
- Experimental
  - Atlas
  - Glast
  - Opera
  - ...

Comparisons

Automake → Very complex (only for experts) no semantic for versions, areas

SRT → Based on autoconf, shell scripts (interfaced)

MSDev → Package organisation too limited, need something above. And ... portability! (interfaced)

CVS → Only for source control. (interfaced)

RPS → Similar concepts, good for import/export (to be interfaced??)

Discussion...