CMT : a Software Configuration Management Tool

Christian Arnault[1]

arnault@lal.in2p3.fr LAL, CNRS, France

Abstract

CMT [1] proposes a highly scalable and flexible environment for managing software configuration. Its applications range from individual developers, willing to work on simple test applications up to large software base management such as the ones usually found in large HEP experiments. CMT permits to develop a project-specific organisation for software as well as for team operations, providing simple mechanisms for groupware activities, iterative integrations, public releases, inter-project relationships. CMT provides a high level abstraction for configuration items (packages, libraries, applications, documents, build and run parameters, etc...) which can be specified in a simple language and queried upon for control operations. This approach allowed in particular to transparently make interoperate several different configuration related tools (such as 'make' in the Unix world, CVS or VisualStudio in the PC world) so as to let non-experts users mostly ignore them. For more expert users, powerful mechanisms are provided for extending the CMT knowledge base. A really open architecture has been selected for implementating CMT, which makes it easy to interface to a wide range of environments.

CMT is currently successfully used by several HEP experiments either as a production tool or for evaluation (eg. Virgo, LHCb, Nemo, Auger, Opera, Atlas).

Keywords: Software Configuration Management

1 Introduction

CMT is a set of tools and conventions for :

- structuring software development or production. It defines and implements the concepts of areas, packages, versions, constituents.
- organizing software into packages
- describing package properties
- describing package constituents
- operating (controlling) the software production activities (such as management, build, import or export, etc...) by transparently configuring and driving the various conventional productivity tools (CVS, make, MSDev, Web, tar, compilers, linkers, archivers, etc...)

It was originally created (around 1993) to provide support for horizontal software developments at LAL and rapidly proved its ability to handle larger software projects. It evolved widely since then especially in its implementation (while the original concepts has largely survived these evolutions).

The design strategy has always been based upon a set of goals and user requirements which can be understood as the corner stone of this product. We may list the most important goals appearing now as essential practices in modern software project management. CMT must be able to :

- 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* and *modularity*)

- manage site, platform or product specific properties (*adaptability*)
- automate complex production tools for non software experts (*simplicity*) (and do not do what existing tools do well!)
- factor out know-how from project to project, from package to package or from team to team (*process improvement*)

2 Operating CMT

This section shows, using some typical use cases, how CMT is generally operated and how it helps software management, by characterizing the main activity phases a developer may undertake :

- to define the various software areas where packages are located and searched for.
- to describe and parameterise the package configuration
- to drive the various productivity tools and to monitor the package states

2.1 Defining software areas. The CMTPATH parameter

Software bases generally deal with several sources of packages, some are produced by the project specific teams, some are re-used from companion projects (such as similar HEP experiments), some are obtained from academic sources and some are simply commercial products.

On another hand, teams within a project are split according to many aspects (geographical constraints, traditional local expertise, working groups, etc...) and development phases also yield another decomposition of the software base.

CMT provides an efficient mechanism for handling these situations, by defining a prioritized package search path parameter, implemented as a path-like environment variable (CMTPATH) on Unix platforms or as a Registry entry (HKEY_LOCAL_MACHINE/Software/CMT/path/...) on Windows platforms.

This simple mechanism does not impose any precise semantics upon each entry, and it's up to project managers to assign specific semantics according to specific needs or choice. One may quote typical usage for this search list :

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

2.2 Describing and parameterising the package configuration. The requirements text file

Since, as this will be shown within the section on the concepts, the central concept endorsed by CMT is the package, the main feature of CMT consists in specifying the package configuration. This is obtained by providing in terms of *configuration parameters* information for :

- describing the relationships between packages (use relationships)
- describing the package constituents (in terms of applications, libraries or other kinds of documents)
- obtaining meta-information about the package (author, manager, ...)
- describing private or public (exported) properties and tool configuration (macros and symbols)

These parameters are all specified within one unique textual file (named `requirements`) per package, located in a conventional location in each package (the `mgr` or the `cmt` branch). This convention may be seen as the **core** requirement for a given package to become CMT-capable.

2.3 Driving the various productivity tools while monitoring the package state

One central tool concentrates the mechanisms of understanding the configuration parameters described previously and either generates driving information towards the various productivity tools or monitors the package states.

The `cmt` tool is a command-line driver (mainly a parser for the `requirements` file). It is able to generate (using a wide range of formats) the various input data for tools like `CVS`, `make`, `MSDev`, `tar`, `Web`, etc...

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

An interactive and graphical Java version of the same tool (`jcmt`) adds to the `cmt` features an efficient package browser, able to navigate the set of search paths (specified using the `CMTPATH` parameter). It is in addition able to launch freely defined actions in the context of any reached package (provided access rights are granted to users).

3 The concepts developped in `CMT`

`CMT` lies upon a precise set of management or configuration concepts. Having such a conceptual approach permitted a very efficient and high-quality realisation of the tools since they are strongly based on an object oriented design model. The main concepts are (this is a somewhat simplified view) :

- Area. This is already described in a previous section.
- the Package. This is the very minimal autonomous (self-contained) entity. One may distinguish different usages or kinds of packages :
  - Plain standard packages. This represents the vast majority of packages worked out for a project. They are installed in a conventional structure taking the general form of `<some area>/<package>/<version-tag>/<mgr>`.
  - Stand alone (unstructured) packages. They generally contain simple test applications. They can use packages from any standard area but cannot themselves be used by other packages (since they are unstructured).
  - Glue packages. They install external software such as `LHC++`, `Geant4`, `Objectivity` into the CMT conventions
  - Interface packages. They provide generic configuration parameters to a set of correlated packages providing high level abstract packages such as `Simulation`, `Reconstruction`, `Graphics`, or even `Atlas` (for getting the global software base setup)
- Version. The version tags handle backward compatibility specifications using the conventional major-id/minor-id paradigm. Alternate version strategies are available such as *first-choice*, *last-choice* or *keep-all*. The default strategy computes the best possible set of versions available from the search path, taking overrides and unresolvable conflicts (where user action would be required) into account.
- Use relationship. It induces inheritance-like properties and version constraints, by setting a dependency between two package-version duets. Public configuration parameters are inherited through it. The use specification defines a graph of links (with possible redundancy) and the `cmt` tool provides the reduction algorithm for the tree (taking search paths into account) by the `cmt show uses` command.

- Package Constituents. Applications and libraries exploit standard behaviour for make or MSDev, through conventional make macros and fragments (make and MSDev). Whereas *documents* can be fully tuned and specified for any document filter through an open-architecture-based mechanism. Fragments and macros overriding can occur in any client package.
- the Configuration parameters. Macros, environment variables, include search path, etc... can be specified using the `CMT` syntax and automatically interpreted with the `cmt` tool (eg. in setup scripts or in makefiles)

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

  Alternate value can be specified for different sites, platforms, and working conditions edition

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

  General edition features (to append, prepend or remove text elements) are also provided.

4 Status and implementation

The main `cmt` line-mode driver has been ported to all Unix and Windows-based platforms (including LynxOS). This is a plain stand-alone C++ application (no use of external libraries besides `iostream`). It is interfaced with MSDev environment through calls to `cmt` from the customisation menus.

The `jcmt` interactive interface is written in standard Java (using `JDK1.1`).

4.1 Comparison with other tools

Several well known products are used for some configuration management activities. They can be compared with `CMT` in this respect, although they may not cover exactly the same range of usage :

`automake`[3]	Very complex (only for experts) no semantic for versions, areas
`SRT`[4]	Based on autoconf, shell scripts (low portability and flexibility)
`MSDev`	Limited package organisation, need something above it. And ... portability!
`CVS`	Only for source control. It is well interfaced with `CMT`
`RPM`[2]	Similar concepts, good for import/export. An interface to RPM features is envisioned.

5 Conclusions and acknowledgements

`CMT` is currently successfully used by several physics experiments either as a production tool or for evaluation (eg. Virgo, LHCb, Nemo, Auger, Opera, Atlas). This already results in continuous improvements both on the reliability side and on the feature evolutions (especially recently those related with the interfacing with MSDev).

I must particularly thank Florence Ranjard and the LHCb software team for their huge patience and extremely valuable help and advices. The LAL Virgo software team was the first large and realistic context where `CMT` grew up.

References

1 "The CMT Web page", `http://www.lal.in2p3.fr/SI/CMT/CMT.htm`
2 "The RPM Web page", `http://www.rpm.org/`
3 "The automake Web page", `http://www.gnu.org/software/automake/automake.html`
4 "The SRT Web page", `http://www.cern.ch/Atlas/GROUPS/SOFTWARE/DOCUMENTS/srt_html/index.html`