

# CCA Port, Component & Application Build Skeleton Templates

"A new script toolkit for generating CCA  
build skeletons"

Torsten Wilde and James Kohl  
Oak Ridge National Laboratory

CCA Forum Quarterly Meeting  
Knoxville, TN ~ January 2004



# What's this all about?

- Creating your own build environment from scratch is not easy unless you are experienced...
  - Very hard/impossible if you are not (most users ? ☺)
- As the complexity increases (e.g. for rpms), the difficulty level increases as well
- CCA build structure is relatively complicated
  - lots of dependencies
  - additional tool requirements (e.g. Babel)
- Steep learning curve could scare off potential users
- Build “Templates” (Note: NOT C++ Templates) can help you to generate basic CCA code structure in a well defined way



- Designing a build system is a trade-off between standardization & user flexibility
- Ports are installed for use in CCA components (include/cca-ports, lib/cca-ports)
- Components are installed for use in an application (cca/components)
- Applications combine port and component implementations into a full executable (cca/applications)
- Goal: High level template configuration should be flexible, e.g. using configure, automake, libtools & rpm technology, but should be automated and hide the complexity

# Template vs. Build Skeleton

- The template files are installed once on your system
  - There are generic templates for ports, components or applications
- The build skeleton is created automatically from the templates
- The user edits ONE template configuration file and uses a perl script to generate the specific project skeleton from the generic template files (including subdirectories)



- Two parts:
  - How to create/use the skeleton
  - Behind the scenes



# Example: Application Skeleton

- **Task:** Create an application that uses an existing component and an internal driver (driver lives in the application directory)
- Requirements:
- The component and the port used by the driver are already installed on the system
- The application template is installed
- Example will show and explain the steps involved in the creation and use of the build skeleton
- (similar but less complicated procedures are used for the port and component build skeletons)



- [10 step process](#)
- 1. User goes into project directory
- 2. User executes:
  - `create-cca-application --init`
  - This will generate the “template-config” file and a usage documentation file in the current directory
- 3. User edits the template-config file
- 4. User executes:
  - `create-cca-application --all`
  - This will generate the application build skeleton
- 5. User copies SIDL file for the driver into the driversubdir/sidl/
- 6. User executes:
  - `./run_babel.sh`
  - This script enters all subdirs and executes a script that generates the source files from the SIDL file using the Babel compiler



# Creation Process continues

- 7. User edits the Babel `_impl` files
- 8. User edits `.in` files
  - Including info `.in`, framework files (`run_cmdline.in`, `rc.in` files)
- 9. User executes:
  - `./autogen.sh`
  - This will call `autoconf`, `automake` & `libtools` on all `configure.ac` and `Makefile.am` in the current source tree
  - At the end it will call the top level `configure`
    - Any provided `--` options are passed on to it
- 10. Then, user can call:
  - `Make`
  - `Make install`
  - `Make uninstall`
  - `Make rpm`
  - `Make backup`



- Templates vs. Build Skeleton
- Two parts:
  - How to create/use the skeleton
  - Behind the scenes



# Step 2,3 & 4 in more detail

- “create-cca-application –help” shows all available options
  - --init
    - Creates initialization files in the current directory
  - --all
    - Builds templates for all packages defined in “template-config” file
  - --main
    - Builds only the main package templates
  - --package=PACKAGE\_NAME
    - Builds only the specific package template
  - --packageexclude=PACKAGE\_NAME
    - Excludes the specified package from the build tree
  - --packagedelete=PACKAGE\_NAME
    - Deletes the specified package from the build & the directory tree



# Step 2,3 & 4 in more detail

- The “README\_TEMPLATE” file
  - provides step-by-step description of the skeleton creation & usage process
- The “template-config.tempin” file
  - This file is the template configuration file
  - Packages are defined between [package\_name] [/package\_name] blocks
  - Package [MAIN] [/MAIN] is the only required package name
    - Additional packages can be added or delete by the user
    - Any change to this file requires re-execution of “create-cca-application”



# Template-config.tempin in detail

```
[MAIN]
# The name of your application (also autoconf package name)
XXX_MY_NAME_XXX = app-Test-getSumPort

# package version
XXX_VERSION_XXX = 0.1

XXX_CONTACT_EMAIL_XXX = wildet@ornl.gov

# RPM info
XXX_RPM_SUMMARY_XXX = "bla bla."
XXX_RPM_DESCRIPTION_XXX = "bla bla"
XXX_PACKAGELICENSE_XXX = LGPL
XXX_RPM_GROUP_XXX = Development
XXX_RPM_REQUIRES_XXX =

# component type
XXX_CCA_COMPONENT_TYPE_XXX = babel

# additional autoconf macros (list separated by spaces)
XXX_ADD_AUTOCONF_MACROS_XXX =
```

```
[/MAIN]
```



- ```
# your component name (will be name of subdir as well)
[Apps-Driver]
# Name of your sidl file including subdirs
XXX_SIDL_FILE_NAME_XXX = sidl/driver.sidl
XXX_SIDL_IMPLEMENTATION_CLASS_NAME_XXX = Drivers.Driver
XXX_SIDL_SERVER_TYPE_XXX = C++
# additional linker flags
XXX_ADD_LINK_FLAGS_XXX =
# additional compiler flags
XXX_ADD_COMPILER_FLAGS_XXX =
# list one client port library for each port used and provided
# installed client port libs are usually in: usr/local/lib/cca-
ports
XXX_CLIENT_PORTS_LIB_XXX = -lgetSumPort-client-Cxx
# additional needed user library directories & user libraries
# add -l in front of each library and -L for each library
directory
XXX_ADD_LIB_DIRS_XXX =
XXX_ADD_LIBS_XXX =
# additional needed user header directories
# add -I in front of each directory
XXX_ADD_HEADER_DIRS_XXX =
[/Apps-Driver]
```



# Step 6 “run\_babel.sh” in more detail

- Script that goes into subdirs and will call “create-client-interface.sh” if there
  - Usage : “./run\_babel.sh” [OPTION] [OPTION ARGUMENT] ...
  - Options:
    - -[h] or -[-help]  
Print out command line help.
    - -[sbcd] [CCA Spec Babel Config Dir]  
Points to the directory where the cca-spec-babel-config file is installed.  
(e.g. -sbcd /usr/local/bin)
    - -[bd] [BABEL EXECUTABLE DIR]  
Points to the directory where the babel executable is installed.  
(e.g. -bd /usr/local/bin)



# Step 8 "autogen.sh" in more details

- Script file that will call automake, autoheader and autoconf on all files recursively (if needed)
  - Usage: `./autogen.sh` [OPTION] [OPTION ARGUMENT] [ARGUMENTS] ...
  - Options:
    - `-[h]` or `-[-help]`  
Print out command line help.
    - `-[acmd]` [autoconf macro directory]  
Points to the directory where the standard skeleton autoconf macros are installed. (e.g. `-acmd /usr/local/cca/components/autoconf-macros-dir`)
    - [ARGUMENTS]  
whatever you would like to let autogen.sh call all configure with at the end (e.g. `-prefix=MY_INST_DIR -with-babel=BABEL_DIR`)



# What happens if you change stuff?

- User can edit all build system files by hand after skeleton was created if he needs to (no re-run of the create-cca-xxx command is required)
- If the user changes/deletes or adds entries in the template-config.tempin file, create-cca-xxx has to be re-run
- User can specify the desired behavior with command line options
  - Using --package, --all or --main will overwrite only required files (it will make a backup of each file that was changed by the user)
  - Using --force in addition will make no backups
  - Using --cleanup before rebuilding will erase subsequent build information, after that all files are rebuild



- Relative easy to use (provided you use standard install locations)
  - only one file to edit
  - 3 commands to call
- Hides build system complexity from the normal user
  - Can specify additional configure macros and can use their variables inside the template-config file
  - Can specify specific compile/link options inside the template-config file
- Leaves all flexibility in (for advanced users)
  - If needed, configure.ac & Makefile.am can be edited directly (changes will be preserved using backup before overwriting)
- Should make the transition to developing for CCA much easier by saving the time and effort needed to create a CCA compatible build environment



- Probably more to do than apparent
  - Enable multiple autoconf macro directories
  - Cleanup standard compile/link flags
  - Streamline and improve documentation
- Just finished test process last Friday
  - Testers/CCA Developers needed to test design and get feedback
  - Maybe additional variables or other structuring is needed
  - Maybe converting shell script files to perl scripts
  - Making skeletons a more generic build tool?
    - could be done by taking out CCA specific configure macros and prefixes & putting them into a internal specification file
    - For babel or classic mode
- Demo will now follow

