Understanding and using catkin (and bloom)

May 12, 2013

Dirk Thomas, William Woodall

ROSCon 2013

Workflow

With catkin and bloom from source to (Debian) packages

catkin - build system

- makes it more efficient to build your package(s)
- makes your package more standard compliant, hence reusable by others

bloom - release tool

- supports you in creating (Debian) packages
- makes applying patches during the process easy

Why catkin?

Limitation of rosbuild

- *make* invoking *CMake* invoking *make*?!
  - why does "rosmake MyPkg" take so long...
- difficult to cross-compile
- no "install" target
  - everything (except .* and *.o files) gets packaged in debs
- distribution of packages not compliant to Filesystem Hierarchy Standard (FHS)

Why "catkin"?

"A catkin or ament is a slim, cylindrical flower cluster ... from a willow tree." (see Wikipedia)

History

- Fuerte
  - first iteration of catkin was only used by core packages
- Groovy
  - major overhaul, this version has only a little in common with the Fuerte version
- Hydro
  - mostly identical with Groovy

Goals for the new build system

- Reusing existing tools
  - Don't reinvent the wheel
  - Don't wrap existing API and tools
- Be as standard-compliant as possible
  - Interface well with other (build) tools
  - Enable FHS compliance
- Resolve the mentioned shortcomings of *rosbuild*

catkin = cmake + X

catkin is based on CMake

- catkin utilizes packaging conventions like
  - find_package() infrastructure
  - pkg-config
- extends CMake with some "nice to have" features like
  - enables to use packages after building (without installation)
  - generates find_package() code for your package
  - generates pkg-config files for your package
  - handles build order of multiple packages
  - handles transitive dependencies
  - (optionally) builds multiple packages with a single CMake invocation (for performance)

CMake

Is a widely used standard

- common CMake workflow
  - mkdir build && cd build
  - cmake ..
  - make

Enables a lot of features

- e.g. cross-compilation

But is also very explicit and requires "more" stuff in CMakeLists.txt than *rosbuild*

- e.g. explicitly installing artifacts

Overview of a catkin package

package.xml (as specified in REP 127)

- Contains the meta information of a package
  - name, description, version, maintainer(s), license
  - opt. authors, url's, dependencies, plugins, etc...

CMakeLists.txt

- The main CMake file to build the package
- Calls catkin-specific functions/macros
  - "Read" the package.xml
  - find other catkin packages to access libraries / include directories
  - export items for other packages depending on you

Other common files in a package

Common source layout

include/PkgNamespace/
src/
setup.py

ROS specific subfolders

msg/
srv/
launch/
scripts/

The recommended folder name equals the package name.

What goes in the package.xml? Required tags

```
<?xml version="1.0"?>
<package>
  <name>rospy_tutorials</name>
  <version>0.3.12</version>
  <description>
    This package attempts to show the features of ROS Python API
    step-by-step, including using messages, servers, parameters, etc.
    These tutorials are compatible with the nodes in roscpp_tutorial.
  </description>
  <maintainer email="dthomas@osrfoundation.org">
    Dirk Thomas
  </maintainer>
  <license>BSD</license>
  ...
</package>
```

What goes in the package.xml?
Optional tags

```
<?xml version="1.0"?>
<package>
  ...
  <url type="website">http://www.ros.org/wiki/rospy_tutorials</url>
  <url type="bugtracker">
    https://github.com/ros/ros_tutorials/issues
  </url>
  <url type="repository">https://github.com/ros/ros_tutorials</url>
  <author>Ken Conley</author>
  ...
  <export>
    ...
  </export>
</package>
```

What goes in the package.xml? Different dependencies

<build_depend>

dependencies to build your package

<buildtool_depend>

tools used on the building platform
to build your package, usually catkin

<run_depend>

dependencies other packages need to build
against your package or use your package
- if the dependency provides a shared library
- if the headers of the dependencies are exposed

<test_depend>

additional dependencies to run tests

What goes in the package.xml?
Optional tags - dependencies

```
<?xml version="1.0"?>
<package>
  ...
  <buildtool_depend version_gte="0.5.65">catkin</buildtool_depend>

  <build_depend>message_generation</build_depend>
  <build_depend>rostest</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>message_runtime</run_depend>
  <run_depend version_gte="1.9.44">rospy</run_depend>
  <run_depend>std_msgs</run_depend>
</package>
```

What goes in the CMakeLists.txt?

```
cmake_minimum_required(VERSION 2.8.3)
project(roscpp_tutorials)  # same package name as in package.xml

find_package(catkin REQUIRED COMPONENTS
   message_generation rosconsole roscpp roscpp_serialization rostime)
find_package(Boost REQUIRED COMPONENTS date_time thread)

include_directories(  # consider reasonable include order
   include ${catkin_INCLUDE_DIRS} ${Boost_INCLUDE_DIRS})

add_library(mylib src/file1.cpp src/file2.cpp)  # same for add_executable
target_link_libraries(mylib ${catkin_LIBRARIES} ${Boost_LIBRARIES})

catkin_package(  # information exposed to other packages
   INCLUDE_DIRS include
   LIBRARIES mylib
   CATKIN_DEPENDS message_runtime std_msgs)
```

What goes in the CMakeLists.txt? Installing all artifacts

```
# executables are installed to lib/PKGNAME to be rosrun-able
install(TARGETS myexe
    RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})

# scripts are installed under lib/PKGNAME to be rosrun-able
# only a selected set of core binaries should go into the global bin
install(PROGRAMS myscript
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})

# libraries are installed to lib (or bin for dll's under Windows)
install(TARGETS mylib
    RUNTIME DESTINATION ${CATKIN_GLOBAL_BIN_DESTINATION}
    ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
    LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION})
```

What goes in the CMakeLists.txt? Installing all artifacts

```
# header are installed to include (within their namespaced subfolder)
install(DIRECTORY include/PKGNAME/  # note the trailing slash
    DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
    FILES_MATCHING PATTERN "*.h")  # skip hidden files/folders

# other files/folders are installed to share/PKGNAME
install(FILES otherfile.txt
    DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION})
install(DIRECTORY myfolder  # note the missing trailing slash
    DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION})
```

Packages with Python code

CMakeLists.txt

```
# after catkin_package()
catkin_python_setup()
```

setup.py # don't invoke *python setup.py*, pkg would be incomplete

```
#!/usr/bin/env python
from distutils.core import setup
from catkin_pkg.python_setup import generate_distutils_setup

# this function uses information from package.xml to populate dict
d = generate_distutils_setup(
    packages=['mypackage'],
    package_dir={'': 'src'}
)
setup(**d)
```

Messages and other code generation

package.xml

build_depend on message_generation
run_depend on message_runtime,
build_depend and run_depend on other message packages

CMakeLists.txt

```
find_package(catkin REQUIRED COMPONENTS
  message_generation std_msgs)

add_message_files(DIRECTORY msg FILES
  Foo.msg Bar.msg)
add_service_files(DIRECTORY srv FILES
  Baz.srv)

generate_messages(DEPENDENCIES std_msgs)

catkin_package(CATKIN_DEPENDS message_runtime std_msgs)
```

How to build a package?

Any catkin package is buildable with the standard CMake workflow

- mkdir build && cd build
- cmake ..
- make

- make tests # for building all unit tests
- make run_tests # for running all unit tests

But this is usually not what you want to do...

How to build multiple packages? Workspaces

Arrange them in a *workspace*

```
/ws/  # root of the workspace
/ws/src/  # source space
/ws/src/repoA
/ws/src/repoB
/ws/src/pkgC
/ws/src/pkgD
```

Commonly the repositories are cloned using a .rosinstall file and *wstool*

How to build multiple packages? catkin_make

- Sourcing your ROS environment, commonly

  source /opt/ros/groovy/setup.bash

- Invoking *catkin_make* in the root of the workspace will create the subfolders

  /ws/build/ # build space

  /ws/devel/ # devel space (FHS layout)

- Invoking *catkin_make install* installs the packages to

  /ws/install/ # install space

How to build multiple packages? catkin_make_isolated

Build each package in "isolation"

Workflow for each package, in topological order:

1. source environment of previous package
2. cd build_isolated/pkg_name
3. cmake path/to/pkg [other CMake options]
4. make [other make options (-j, -l, etc.)]
5. make install # to install_isolated
6. repeat with 1. for next package

Works with "plain" CMake packages (e.g. console_bridge)

Using the packages after building

Source the generated environment

(you should do that in a separate shell)

- either from the devel space

  source ws/devel/setup.bash

- or from the install space to verify that your packages install their artifacts correctly

  source ws/install/setup.bash

Overlaying workspaces

- Context of a workspace is defined
  by the environment when invoking *catkin_make*

- Sourcing ws A after building/installing it
  before calling *catkin_make* on ws B
  - makes B an overlay of A
  - respectively A and underlay of B

- Building of workspaces must be triggered explicitly,
  it does not automatically chain

Should I update?
Absolutely! But when?

- rosbuild packages can depend on catkin packages seamlessly
  - but not the other way around!
  - transition from rosbuild to catkin must happen from bottom up
  - catkin packages are called *wet*, and rosbuild *dry*; analogy is a raising water line

- rosbuild for building-from-source will be available ... "forever"
  - but the build farm for generating Debian packages, running continuous integration and documentation will likely go away soon... (after the Hydro release?)

How to release catkin package?

First, the source repository must be prepared:

- increment the version number of all packages in the repository
- tag the repository with the release version

This task is automated by *catkin_prepare_release*.

The next step is to use *bloom* to create a git-buildpackage compatible repository.

What is a GBP repository?

A git-buildpackage repository is a *git* repository which serves as a staging place for released packages.

In a GBP you can:

- Split up a repository by package
- Add ROS or platform specific patches
- Generate distribution files (debian files)
- Create tags for specific versions of packages

Enter bloom

bloom is a release automation tool, which uses the platform agnostic package.xml meta-data to generate platform specific artifacts.

bloom

- Splits up multiple packages into branches
- Allows the releaser to patch the upstream
- Generates platform specific release files

  - like Debian files or Fedora files
- Generates git-buildpackage compatible tags

bloom also has scripts to help further automate this process within the ROS infrastructure.

How does bloom work?

bloom's workflow (for debs) is as follows:

- Export the upstream repository at a specific version (vcs tag or reference) to an archive
- Import this archive into an *upstream* branch
- Create a *release* branch for each package
  - Patches from previous releases are applied
- Create a *debian* branch for each package
  - Previous debian patches are applied
  - These are patches to the debian files, not sources
- Creates *tags* for each debian distribution

Great, now what?

Now your package is "released", but no one knows about it until you put it in a ROS distribution. A distribution is defined by the repositories in the release file (REP 137):

e.g. https://github.com/ros/rosdistro/blob/master/groovy/release.yaml

This file defines what packages make up the distribution, what version of those packages are released, and where those packages have been released (location of the GBP), along with other meta information.

A typical release file entry

```
repositories:
  ...
  driver_common:  # name of the repository (not a package)
    packages:  # list of packages and their relative location in the repo
      driver_base:  # no value means pkg is in subfolder with same name
      driver_common:
      timestamp_tools:
    status: end-of-life  # can also be developed, maintained, unmaintained
    status_description: Will be released only as long as required
                        for PR2 drivers (hokuyo_node, wge100_driver)
    tags:  # tags are used by the build farm infrastructure
      release: release/{package}/{upstream_version}
    url: https://github.com/ros-gbp/driver_common-release.git
    version: 1.6.5-0  # including the deb-inc number
```

Getting into the release file

In order to get your package into the release file, you must open a pull request to the release file you wish to put your package in.

What does being in the release file do?

- It is used by the rosinstall generator
  - users building "from source" will use this
- The build farm will pickup your package and build a sourcedeb and binarydebs of your package

What about wiki documentation?

Being in the release file gets your package into the system for the build farm and "from source" installations, but it does not get you into the documentation system.

Why do I have to put my package information in multiple places?

- some packages are documented but not released
- some packages are released but not documented
- also different information is required for documentation,
  commonly documenting a specific branch

Getting documented

In order for our documentation indexer to process your repo, there needs to be an entry in the documentation file:

e.g. https://github.com/ros/rosdistro/blob/master/groovy/doc.yaml

```
repositories:
  ...
  driver_common:  # name of the repository (not a package)
    # VCS type, can be git, hg, svn, or bzr
    type: git
    # upstream VCS url
    url: https://github.com/ros-drivers/driver_common.git
    # branch, tag, or reference (usually development branch)
    version: groovy-devel
```

What about third-party libraries?

We don't want to modify them to use catkin, but we want to build them seamlessly with other catkin packages.

To accomplish this there are several options, see REP 136.

Recommendation:

- Place package.xml upstream
  - contains the dependency information
  - install package.xml in *${prefix}/share/${pkg_name}*
- Mark package with build type cmake:
  - builds the package with standard cmake invocation

```
<export>
    <build_type>cmake</build_type>
</export>
```

Are we done?

Not really, there are always things to improve...

- configure without tests?
- running tests after installation?
- creating packages w/o debug symbols?
- separating packages into runtime/dev?
- simplifying usage of catkin/CMake?
- ...

A lot of ideas, but what do we want to realize (and how)?

Any feedback and suggestions
(and contributions) are welcome.