colcon Documentation

Dirk Thomas

Jun 01, 2022

User Documentation

colcon is a command line tool to improve the workflow of building, testing and using multiple software packages. It automates the process, handles the ordering and sets up the environment to use the packages.

The code is open source, and available on GitHub.

The documentation exists in two version:

- released: matching the latest released version of all packages
- latest: matching the latest state on the default branch of all packages

The documentation is organized into a few sections:

- *User Documentation*
- *Migrate from other build tools*

Information about development is also available:

- *Developer Documentation*

CHAPTER 1

Installation

The functionality of `colcon` is split over multiple Python packages. The package `colcon-core` provides the command line tool `colcon` itself as well as a few fundamental extensions. Additional functionality is provided by separate packages, e.g. `colcon-cmake` adds support for packages which use CMake. The following instructions install a set of common `colcon` packages.

1.1 Using Debian packages

On platforms which support Debian packages using those is preferred since they will be updated using `apt` together with other system packages.

1.1.1 In the context of the ROS project

The ROS project hosts copies of the Debian packages in their apt repositories. You can choose either of the two following apt repositories.

- ROS 1 repository

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -cs` main
↪" > /etc/apt/sources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key␣
↪C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

- ROS 2 repository

```
$ sudo sh -c 'echo "deb [arch=amd64,arm64] http://repo.ros2.org/ubuntu/main `lsb_
↪release -cs` main" > /etc/apt/sources.list.d/ros2-latest.list'
$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo␣
↪apt-key add -
```

After that you can install the Debian package which depends on `colcon-core` as well as commonly used extension packages (see setup.cfg).

```
$ sudo apt update
$ sudo apt install python3-colcon-common-extensions
```

1.1.2 Outside the ROS project

The Debian packages are also hosted in an apt repository provided by packagecloud:

You can add the GPG key as well as the apt repository using the following command (which is described here).

```
$ curl -s https://packagecloud.io/install/repositories/dirk-thomas/colcon/
↪script.deb.sh | sudo bash
```

After that you can install the Debian package which depends on `colcon-core` as well as commonly used extension packages (see setup.cfg).

```
$ sudo apt install python3-colcon-common-extensions
```

1.2 Using pip on any platform

On all non-Debian platforms the most common way of installation is the Python package manager `pip`. The following assumes that you are using a virtual environment with Python 3.5 or higher. If you want to install the packages globally it might be necessary to invoke `pip3` instead of `pip` and require `sudo`.

```
$ pip install -U colcon-common-extensions
```

Note: The package `colcon-common-extensions` doesn't contain any functionality itself but only depends on a set of other packages (see setup.cfg).

Note: You can find a list of released packages on PyPI using the keyword `colcon`.

1.3 Installing from source

Note: This approach is commonly only used by advanced users.

Commonly this is the case when you want to try or leverage new features or bug fixes which have been committed already but are not available in a released version yet. In order to use the latest state of any of the above packages you can invoke `pip` with a URL of the GitHub repository:

```
$ pip install -U git+https://github.com/colcon/colcon-common-extensions.git
```

1.4 Installing custom branches from source

To try a patch proposed in a pull request you can install the sources of that specific branch by appending the branch name to the URL:

```
$ pip install -U git+https://github.com/colcon/colcon-core.git@branch_name
```

Note: Make sure to uninstall that custom version again using `pip uninstall <name>` to revert back to the previously used version. Otherwise if you use the Debian packages this pip installed package will overlay even newer Debian packages.

1.5 Building from source

Since this is not a common use case for users you will find the documentation in the *developer section*.

1.6 Quick directory changes

1.6.1 Sh (and compatible shells)

On Linux / macOS the above instructions install the package `colcon-cd` which offers a command to change to the directory a package specified by its name is in. To enable this feature you need to source the shell script provided by that package. The script is named `colcon_cd.sh`. For convenience you might want to source it in the user configuration, e.g. `~/.bashrc`:

Depending on which instructions you followed to install the packages the location will vary:

- Debian package: `/usr/share/colcon_cd/function`
- PIP - user specific: `$HOME/.local/share/colcon_cd/function`
- PIP - global: `/usr/local/share/colcon_cd/function`

When building `colcon` from source the generated setup files will automatically include this hook.

1.7 Enable completion

1.7.1 Bash / zsh

On Linux / macOS the above instructions install the package `colcon-argcomplete` which offers command completion for bash and bash-like shells. To enable this feature you need to source the shell-specific script provided by that package. These scripts are named `colcon-argcomplete.bash` / `colcon-argcomplete.zsh`. For convenience you might want to source the one matching your shell in the user configuration, e.g. `~/.bashrc`:

Depending on which instructions you followed to install the packages the location will vary:

- Debian package: `/usr/share/colcon_argcomplete/hook`
- PIP - user specific: `$HOME/.local/share/colcon_argcomplete/hook`
- PIP - global: `/usr/local/share/colcon_argcomplete/hook`

When building `colcon` from source the generated setup files will automatically include this hook.

CHAPTER 2

Quick start

This section gives a high-level overview of how to use the `colcon` command.

2.1 TL;DR

The following is an example workflow and sequence of commands using default settings:

```
$ mkdir -p /tmp/workspace/src      # Make a workspace directory with a src subdirectory
$ cd /tmp/workspace                # Change directory to the workspace root
$ <...>                            # Populate the `src` directory with packages
$ colcon list                      # List all packages in the workspace
$ colcon graph                     # List all packages in the workspace in topological␣
↪order
                                   # and visualize their dependencies
$ colcon build                     # Build all packages in the workspace
$ colcon test                      # Test all packages in the workspace
$ colcon test-result --all         # Enumerate all test results
$ . install/local_setup.bash       # Setup the environment to use the built packages
$ <...>                            # Use the built packages
```

The most commonly used arguments for the `build` and `test` verbs are to only process a specific package or a specific package including all the recursive dependencies it needs.

```
$ colcon build --packages-select <name-of-pkg>
$ colcon build --packages-up-to <name-of-pkg>
```

Note: The log files of the latest invocation can be found in the log directory which is by default in `~/.colcon/log/latest`.

Note: If you want to see the output of each package after it finished you can pass the argument `--event-handlers`

`console_cohesion+`.

2.2 Build ROS 2 packages

The process of building ROS 2 packages is described in the ROS 2 building from source instructions. Using `colcon` instead of the recommended tool `ament_tools` only changes a couple of the steps.

Instead of invoking `ament build` you can invoke `colcon`.

```
$ colcon build
```

In order to use the built packages you need to source the `install/local_setup.<ext>` script mentioned in the instructions.

For detailed information how command line arguments of `ament_tools` are mapped to `colcon` please see the *ament_tools migration guide*.

2.3 Build ROS 1 packages

The process of building ROS 1 packages is described in the distro specific building from source instructions. Using `colcon` instead of the recommended tool `catkin_make_isolated` only changes a couple of the steps.

Note: `colcon-ros` requires at least version 0.7.13 of `catkin` which provides a new CMake option the tool uses.

Instead of invoking `catkin_make_isolated --install` you can invoke `colcon`.

```
$ colcon build
```

Note: `colcon` does by design not support the concept of a "devel space" as it exists in ROS 1. Instead it requires each package to be installed so each package must declare an install step in order to work with `colcon`.

In order to use the built packages you need to source the `install/local_setup.<ext>` rather than the `setup.<ext>` script mentioned in the instructions. For `bash` the command would be:

```
$ source install/local_setup.bash
```

For detailed information how command line arguments of `catkin_make_isolated` are mapped to `colcon` please see the *catkin_make_isolated migration guide*. For detailed information how command line arguments of `catkin_tools` are mapped to `colcon` please see the *catkin_tools migration guide*.

2.3.1 Test ROS 1 packages

As of `colcon-ros` version 0.3.6 the `build` verb builds the test targets for ROS 1 packages implicitly (when available).

In earlier versions you must build the custom `tests` target explicitly:

```
$ colcon build --cmake-target tests
```

2.4 Build Gazebo and the ignition packages

In more recent versions Gazebo has been refactored to split out a lot of the functionality into ignition libraries. While that makes the project more modular it also increases the effort necessary to build all these packages from source. `colcon` can make this process easy again.

In order to build a specific Gazebo version you need the right versions of the ignition libraries. At the time of writing Gazebo 9 is the latest release so we will use that for the purpose of this example. The following steps use a `.repos` to specify the various repositories with specific branches.

```
$ mkdir -p /tmp/gazebo/src && cd /tmp/gazebo
$ wget https://gist.githubusercontent.com/dirk-thomas/
↪6c1ca2a7f5f8c70ce7d3e1ef10a9f678/raw/490aaba72321284af956c9db12f9ef1550ef88cf/
↪Gazebo9.repos
$ vcs import src < Gazebo9.repos
```

Note: The Gist containing the repository list should be replaced with an "official" URL coming from the Gazebo project.

Before building the workspace with `colcon` the steps also fetch some additional metadata for Gazebo from a public repository.

```
$ colcon metadata add default https://raw.githubusercontent.com/colcon/colcon-
↪metadata-repository/master/index.yaml
$ colcon metadata update
$ colcon build
```

To run Gazebo, which requires environment variables for e.g., the model paths, the same commands as for other packages can be used. Using the additional metadata the source script will also automatically source the Gazebo specific file `share/gazebo/setup.sh` which defines these environment variables.

```
$ . install/local_setup.bash
$ gazebo
```

CHAPTER 3

Configuration

Configuration files can provide additional metadata for packages as well as define default command line arguments. All files described below are using the YAML format. Note that all strings are case sensitive.

3.1 colcon.pkg files

A `colcon.pkg` file must be placed in the root directory of a package.

The first level of the configuration file is a dictionary. The key can contain any of the following:

- `name` (string) to declare the package name
- `type` (string) to explicitly declare which colcon extension should process the package.
- `dependencies` (list of strings) to declare additional dependencies on other packages. For more fine grain control it is also possible to set `build-dependencies`, `run-dependencies`, and `test-dependencies`.
- `hooks` (list of relative paths within the install prefix) to declare additional scripts to be sourced.
- Any command line argument. The leading single or double dash must be omitted.

3.1.1 Values for command line arguments

The value type depends on the kind of command line argument:

- For flags which are not followed by a value the value can be either `true` or `false`.
- For options followed by a single decimal / float the value must be a decimal / float.
- For options followed by a single value the value must be a string.
- For options followed by one or more values the value must be a list where each item can be any of the mentioned types.

An example declaring an environment hook which should be sourced for a package:

```
{
    "hooks": ["share/pkgname/hook/something.sh"]
}
```

3.2 .meta files

The first level of the configuration file is a dictionary. The only two supported keys are:

- `names` to provide settings based on the package name.
- `paths` to provide settings based on the package path.

3.2.1 By package name

Note: Providing metadata based on the package name only works if the package can be identified and the name can be determined. Otherwise using a *colcon.pkg* file or the *By package path* configuration is necessary.

The value under the `names` key is again a dictionary.

The key is the name of the package. The value can contain the same package specific settings as described in the *colcon.pkg files* section above. The only exception is that specifying a package name is not supported.

3.2.2 By package path

The value under the `paths` key is again a dictionary.

The key is the path of the package. It can be either absolute or relative to the `.meta` file. The value can contain the same package specific settings as described in the *colcon.pkg files* section above. This can be used if the package name can't be determined automatically and placing a `colcon.pkg` file into the package directory is undesired.

3.2.3 Package specific configuration

The package specific part under the package name or package path has the same content as the package specific configuration files described in the *colcon.pkg files* section above.

3.2.4 Using .meta files

Some configuration files are being picked up by default. The following are a few examples (see e.g. `colcon build --help`):

- When `--ignore-user-meta` is not passed any file ending with `.meta` in any recursive subdirectory of `$COLCON_HOME/metadata` is being used.
- When `--metas` is not passed and a file `./colcon.meta` exists it is being used.
- Any file passed with `--metas <path/to/file>` is being used.

Note: The default value for the environment variable `COLCON_HOME` is pointing to the directory `.colcon` within the users home directory.

3.3 defaults.yaml

If the configuration file `$COLCON_HOME/defaults.yaml` exists it is used to customize the default behavior of the CLI. The location can also be modified using the environment variable `COLCON_DEFAULTS_FILE` (see `colcon --help`).

The first level of the configuration file is a dictionary. The key is the `verb` name. In the case of more than one nested verbs the key is the names separated by dots. To specify configuration options *before* the first verb use an empty string key. The value is another dictionary containing the verb specific configuration.

3.3.1 Verb specific configuration

The key can contain any command line argument. The leading single or double dash must be omitted. The value type depends on the command line argument as mentioned in the *Values for command line arguments* section above.

An example to use the symlink install option by default:

```
{
    "build": {
        "symlink-install": true
    }
}
```

CHAPTER 4

How to

This section describe how to perform common tasks.

4.1 Show all output immediately on the console

```
$ colcon <verb> --event-handlers console_direct+
```

Note: If you use the parallel executor (which is the default when that extension is installed) the output of packages processed in parallel will be interleaved.

4.2 Show all output on the console after a package has finished

```
$ colcon <verb> --event-handlers console_cohesion+
```

Note: While this delays the output until a package has finished, it avoids interleaving the output when using the parallel executor.

4.3 Build only a single package (or selected packages)

```
$ colcon build --packages-select <name-of-pkg>
$ colcon build --packages-select <name-of-pkg> <name-of-another-pkg>
```

Note: This assumes that you have built dependencies of the selected packages within the workspace before.

4.4 Build selected packages including their dependencies

```
$ colcon build --packages-up-to <name-of-pkg>
```

4.5 Rebuild packages which depend on a specific package

Assuming you have built the whole workspace before and then made changes to one package. In order to rebuild this package as well as all packages which (recursively) depend on this package invoke:

```
$ colcon build --packages-above <name-of-pkg>
```

4.6 Build packages that create a Python C/C++ Extension

To be able to build a C/C++ extension when using the option `--symlink-install`, you must include the following lines in your package's `setup.py`:

```
sources = ['one.cpp', 'two.cpp']  # This will contain all C/C++ source files
headers = ['a.h', 'b.hpp', 'c.h']  # Any included header files must be listed here

setup(
    ...,
    data_files=[('.', sources + headers)]  # Assuming that your source and header␣
↪files are
                                           # on the same level as setup.py, this will␣
↪copy
                                           # all required files to the colcon build␣
↪folder
)
```

Finally, run `colcon build --symlink-install` as usual.

4.7 Test selected packages as well as their dependents

If you have built the relevant packages before you can run the tests the same way as described in the *previous section*:

```
$ colcon test --packages-above <name-of-pkg>
```

If you haven't built the relevant packages before you can build the to-be-tested packages as well as their recursive dependencies with:

```
$ colcon build --packages-above-and-dependencies <name-of-pkg>
```

4.8 Run specific tests

Depending on the type of the package a different tool is being used to run tests.

4.8.1 Python packages using pytest

```
$ colcon test --packages-select <name-of-pkg> --pytest-args ...
```

Pytest provides multiple ways to select individual tests:

- Tests can be identified by their name:

  ```
  $ ... --pytest-args -k name_of_the_test_function
  ```

- Tests can be identified using markers if the tests have been decorated with markers before:

  ```
  $ ... --pytest-args -m marker_name
  ```

Both approaches also support logical expressions like `or` and `not`. For more information see the pytest documentation.

4.8.2 CMake packages using CTest

```
$ colcon test --packages-select <name-of-pkg> --ctest-args ...
```

CTest provides multiple ways to select individual tests:

- Tests can be selected / excluded using a regular expression matching their name:

  ```
  $ ... --ctest-args -R regex
  $ ... --ctest-args -E regex
  ```

- Tests can be selected / excluded using a regular expression matching their label (which have to be assigned to each test when adding the test in the CMake code):

  ```
  $ ... --ctest-args -L regex
  $ ... --ctest-args -LE regex
  ```

For more information see the CTest documentation.

4.9 Build CMake packages without configuring tests

For CMake packages which use the CMake option `BUILD_TESTING` (which is the standard in the CTest module) you can skip configuring and building tests to improve the build time:

```
$ colcon build --cmake-args -DBUILD_TESTING=OFF
```

4.10 CMake packages generating compile_commands.json

When the CMake option CMAKE_EXPORT_COMPILE_COMMANDS is enabled a `compile_commands.json` file is generated in the package specific build directory containing the exact compiler calls for all translation units of the project in machine-readable form:

```
$ colcon build --cmake-args -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

`colcon-cmake` will additionally generate a workspace-level `compile_commands.json` in the build directory which aggregates the information from all package specific json files.

4.11 Enable additional output for debugging

Beside the output of the actually invoked commands to build or test packages the tool by default only outputs warning or error messages. For debugging purposes you can enable logging messages with other levels (e.g. `info`, `debug`).

```
$ colcon --log-level info <verb> ...
```

CHAPTER 5

What is a Workspace?

Colcon is a command line tool to build and test multiple software packages. It builds and tests those packages in a **colcon workspace**, but what is a workspace?

This page assumes you have `colcon-common-extensions` installed.

5.1 What's in a workspace?

A workspace has these parts:

- Software packages
- Build artifacts
- Logs
- Install artifacts

All these parts are put into different subdirectories of single directory, called the **workspace root**. Lets create a single directory for our workspace.

```
mkdir ws
```

Go into the root of our new workspace.

```
cd ws
```

5.1.1 Software packages

A workspace needs the source code of all the software to be built. Colcon will search all subdirectories of the workspace to look for packages, but an established convention is to put all the packages into a directory called `src`. Let's create a directory for source code.

```
mkdir src
```

We'll need at least one software package in the workspace. Let's create a Python package.

```
mkdir src/foo
touch src/foo/setup.cfg
touch src/foo/setup.py
touch src/foo/foo.py
```

Put this content into `src/foo/setup.py`:

```
from setuptools import setup

setup()
```

Put this content into `src/foo/setup.cfg`:

```
[metadata]
name = foo
version = 0.1.0

[options]
py_modules =
  foo
zip_safe = true

[options.extras_require]
test =
  pytest

[tool:pytest]
junit_suite_name = foo
```

Put this content into `src/foo/foo.py`:

```
def foo_func():
    print('Hello from foo.py')
    return True
```

5.1.2 Build artifacts

The software build process often produces intermediate build artifacts. They are usually not used directly, but keeping them around makes subsequent builds faster. Colcon always directs packages to build out-of-source, meaning the build artifacts are put into a directory separate from the source code. Every package gets its own build directory, but all build directories are put into a single base directory. By default it's named `build` at the root of the workspace.

Note: You can change where build artifacts are put using the `--build-base` option to `colcon build`.

Lets build the software and see its build artifacts.

```
# Make sure you run this command from the root of the workspace!
colcon build
```

You'll see these new directories: `build`, `install`, and `log`.

```
ws
├── build
│   ├── COLCON_IGNORE
│   └── foo/...
├── install/...
├── log/...
└── src
    └── foo
        ├── foo.py
        ├── setup.cfg
        └── setup.py
```

Notice the `build` directory has a subdirectory `foo` and a file `COLCON_IGNORE`. The `foo` subdirectory has all the build artifacts produced when building `foo`. The `COLCON_IGNORE` file tells colcon there are no software packages in this directory.

5.1.3 Logs

If you've built software before you know there can be a lot of console output, but you might have noticed not much was output when you ran `colcon build`. This output was instead written to the `log` directory.

Note: You can change where logs are written to using the `--log-base` option to `colcon`.

Let's look at the `log` directory:

```
log
├── build_2022-05-20_11-50-03
│   ├── events.log
│   ├── foo
│   │   ├── command.log
│   │   ├── stderr.log
│   │   ├── stdout.log
│   │   ├── stdout_stderr.log
│   │   └── streams.log
│   └── logger_all.log
├── COLCON_IGNORE
├── latest -> latest_build
└── latest_build -> build_2022-05-20_11-50-03
```

The directory `log/build_<date and time>` contains all logs from the invocation of `colcon build`. A new directory is created every time `colcon build` is run. The `foo` sub directory contains all logs from building `foo`. There's a more complete description of log files at this page: *Log Files*.

We've only built `foo`, so there are only build logs. Let's add tests to `foo` and see the output.

Make a new file for the test.

```
touch src/foo/test_foo.py
```

Put the following content into `test_foo.py`:

```
import foo

def test_foo():
    assert foo.foo_func()
```

Tell `colcon` to run the tests.

```
# Make sure you run this command from the root of the workspace!
colcon test
```

Lets look in the `log` directory again.

```
log
├── build_2022-05-20_11-50-03/...
├── COLCON_IGNORE
├── latest -> latest_test
├── latest_build -> build_2022-05-20_11-50-03
├── latest_test -> test_2022-05-20_11-50-05
└── test_2022-05-20_11-50-05
    ├── events.log
    ├── foo
    │   ├── command.log
    │   ├── stderr.log
    │   ├── stdout.log
    │   ├── stdout_stderr.log
    │   └── streams.log
    └── logger_all.log
```

There's a new directory `test_<date and time>`. Let's look at `stdout_stderr.log` to see the output of the latest test.

```
cat log/latest_test/foo/stdout_stderr.log
```

Note: Use the command `colcon test-result` to see a summary of test results on the console after tests have been run.

5.1.4 Install artifacts

The last directory to talk about is the `install` directory. It contains both the installed software, and shell scripts that enable you to use it. This is sometimes called the **install space**.

Note: You can change where packages are installed to with the `--install-base` option to `colcon build`.

Let's look inside.

```
install
├── COLCON_IGNORE
├── foo/...
├── local_setup.[bash|bat|ps1|sh|zsh|...]
├── _local_setup_util_[sh|ps1|...].py
└── setup.[bash|bat|ps1|sh|zsh|...]
```

The package `foo` was installed into the directory `install/foo`. By default colcon builds an **isolated workspace** (for more info see *Isolated vs Merged Workspaces*).

The shell scripts set environment variables that allow you to use the the software. Invoking the shell scripts is called **sourcing a workspace**.

Note: Always source a workspace from a different terminal than the one you used `colcon build`. Failure to do so can prevent colcon from detecting incorrect dependencies.

Source the workspace using the appropriate script for your shell.

`sh` compatible shells:

```
# Note the . at the front; that's important!
. install/setup.sh
```

`bash`:

```
source install/setup.bash
```

Windows `cmd.exe`:

```
call install/setup.bat
```

Now you can use `foo`. Open a `python` interactive console and try it out.

```
>>> import foo
>>> foo.foo_func()
Hello from foo.py
True
```

Sourcing a workspace also allows you to build more software that depends on the packages in it. For more info about using colcon to build software that depends on packages in another workspace, see *Using Multiple Workspaces*.

CHAPTER 6

Log Files

You may notice with `colcon-common-extensions` installed there's not much output to the console when you run `colcon build` or `colcon test`. The log output isn't lost; it was written to the `log` directory.

Note: The `--event-handlers` argument can be used to output build logs to the console. For example, `colcon build --event-handlers console_direct+` or `colcon test --event-handlers console_direct+` will output everything in real time.

Let's look at the `log` directory from *What is a Workspace?*.

```
log
├── build_2022-05-20_11-50-03
│   ├── events.log
│   ├── foo
│   │   ├── command.log
│   │   ├── stderr.log
│   │   ├── stdout.log
│   │   ├── stdout_stderr.log
│   │   └── streams.log
│   └── logger_all.log
├── COLCON_IGNORE
├── latest -> latest_build
├── latest_build -> build_2022-05-20_11-50-03
├── latest_test -> test_2022-05-20_11-50-05
└── test_2022-05-20_11-50-05
    ├── events.log
    ├── foo
    │   ├── command.log
    │   ├── stderr.log
    │   ├── stdout.log
    │   ├── stdout_stderr.log
    │   └── streams.log
    └── logger_all.log
```

The directory `build_<date and time>` contains all logs from the invocation of `colcon build`. `test_<date and time>` contains all the logs from the invocation of `colcon test`. On platforms that support symbolic links, `latest_build` points to the most recent build, and `latest_test` does the same for tests.

More generally, any verb that generates logs (e.g. `build`, `test`, `test-result`) will put them into a subdirectory named `<verb>_<date and time>`. `latest_<verb>` will point to the logs of the most recent invocation of that verb. The symbolic link `latest` will point to the most recent logs for any colcon verb.

Each `<verb>_<date and time>` directory has these files:

- `events.log` contains all internal events dispatched. This file is mostly for debugging purposes.
- `logger_all.log` contains all logging messages regardless of the log level. The first line of this file contains the exact command line invocation including all the arguments passed.

Notice each `<verb>_<date and time>` directory has a `foo` directory. More generally, a directory for every package is created. Package log directories have these files:

- `command.log` contains the commands invoked for the package, e.g. calls to `python setup.py`.
- `stdout.log` contains all the output printed to `stdout`.
- `stderr.log` contains all the output printed to `stderr`.
- `stdout_stderr.log` contains all the output printed to either `stdout` or `stderr` in the order they appeared.
- `streams.log` combines the output of all log files in the order they appeared.

Note: `colcon` does its best to read concurrently from `stdout` and `stderr` to preserve the order, but it can't guarantee correct order of log messages in all cases.

CHAPTER 7

Isolated vs Merged Workspaces

Assuming you know *what a workspace is*, you know that by default colcon builds an **isolated workspace**. It can also build a **merged workspace**. These terms describe the layout of the **install space**, which is the directory software gets installed into.

This page describes the differences between the two.

7.1 Isolated Workspace

Colcon by default builds an isolated workspace.

Build the workspace from *What is a Workspace?* without any extra arguments.

```
colcon build
```

Now let's look in the `install` folder.

```
install
├── COLCON_IGNORE
├── foo
│   ├── lib/...
│   └── share/...
├── local_setup.[bash|bat|ps1|sh|zsh|...]
├── _local_setup_util_[sh|ps1|...].py
└── setup.[bash|bat|ps1|sh|zsh|...]
```

Colcon created a directory `install/foo` and installed the package `foo` inside of it. Building an isolated workspace just means every software package is installed into its own directory.

An isolated workspace has some advantages. When colcon tests a package in an isolated workspace it will only give the tests access to the install artifacts of the dependencies it declares. This allows users to catch undeclared dependencies. An isolated workspace also allows removing a single package's install artifacts by deleting its directory.

7.2 Merged Workspace

Delete the `install` directory and build the workspace again with the `--merge-install` option.

```
colcon build --merge-install
```

Let's look in the `install` folder again.

```
install
    ├── COLCON_IGNORE
    ├── lib/..
    ├── share/..
    ├── local_setup.[bash|bat|ps1|sh|zsh|...]
    ├── _local_setup_util_[sh|ps1|...].py
    └── setup.[bash|bat|ps1|sh|zsh|...]
```

Notice how there's no longer a `foo` folder. The `lib` and `share` directories are now directly in `install`.

Building a merged workspace just means every software package is installed into the same directory.

A merged workspace is advantageous on platforms where environment variables length is more tightly constrained, such as Windows 10. Environment variables set by the shell scripts, such as `PYTHONPATH`, will be shorter because there only needs to be one entry for all of the installed packages.

CHAPTER 8

Using Multiple Workspaces

Assuming you know *what a workspace is*, you know **sourcing a workspace** allows you to use the installed software. It is possible to use software from multiple workspaces at the same time.

8.1 Using multiple independent workspaces

If you have multiple **independent workspaces**, then they can be used at the same time by sourcing them in sequence. Workspaces are **independent** if neither workspace has a package that depends on a package in the other workspace.

```
source foo_ws/install/setup.bash
source bar_ws/install/setup.bash
```

The first workspace `foo_ws` is called the **underlay workspace**. The second, `bar_ws`, is called the **overlay workspace**. When sourcing more then two workspaces each workspace is said to **overlay** the previous one.

Independent workspaces can usually be sourced in any order.

Note: Be cautious when sourcing multiple workspaces. Undefined behavior can result if packages from one depend on packages from another, meaning they're not actually independent. Chain the workspaces instead if you're unsure.

8.2 Chaining workspaces

Chaining workspaces means making a workspace depend on another workspace. To chain workspaces, build the underlay workspace first. Source the underlay in an new terminal and build the overlay.

```
# Build ping_ws
cd ping_ws
colcon build
# In a new terminal source ping_ws and build pong_ws
```

(continues on next page)

(continued from previous page)

```
source ping_ws/install/setup.bash
cd pong_ws
colcon build
```

In this example `pong_ws` overlays `ping_ws`. `pong_ws` may have a package that depends on packages in `ping_ws`, but `ping_ws` cannot have a package that depends on packages in `pong_ws`.

Only the last workspace in a chain needs to be sourced.

```
# Sourcing pong_ws automatically sources ping_ws first
source pong_ws/install/setup.bash
```

Note: Use `local_setup.[sh|bash|bat|...]` if you want to source a workspace without automatically sourcing the underlays it depends on, such as when you've already sourced the underlay.

You can chain any number of workspaces together by repeating these step with more overlay workspace.

8.3 Extending workspaces versus overriding packages

An overlay workspace **extends** an underlay workspace when it provides new packages. Extending workspaces has no known issues and is the most common use case.

It is also possible for an overlay workspace to contain a different version of a package which has already been built in one of the underlay workspaces. This is called **overriding a package**. Ideally, the version in the overlay workspace should be the one used when the workspace chain is sourced, but that doesn't work in all cases. See *Overriding Packages* for more information about known issues.

CHAPTER 9

Overriding Packages

Assuming you know *what a workspace is* and *how to use multiple workspaces*, you may decide you want to change the version of a package in an underlay without rebuilding the whole chain of workspaces. This can be done by **overriding** the package. It is accomplished by sourcing an existing workspace, then building a different version of that package again in a new overlay workspace.

Overriding a package is not always possible. This page offers tips for avoiding common issues.

Table of Contents

9.1 How to avoid common issues

These are good practices to avoid common issues. If the advice is too restrictive then see the known issue descriptions for more complex advice.

9.1.1 Use isolated workspaces

If you are building a workspace and suspect you may override a package from it in the future, then use an isolated workspace. This avoids include directory search order issues. See *this documentation* if you're unsure what an isolated workspace is.

9.1.2 Override every package that depends on the one you want to override

A **leaf package** is one that has no other packages that depend on it. Overriding a non-leaf package can be problematic. Packages in the underlay were built against the underlay version of the package, but they will be expected to run with the overlay version. If their build process stores some information about the non-leaf package, such as an expected ABI, then the behavior at runtime is unpredictable.

Problems caused by packages remembering information at build time can be avoided by overriding every package that directly or indirectly depends on the one you actually want to override. The group of overridden packages must span all underlays.

Say there are 3 workspaces, **A**, **B** and **C**, where **C** overlays **B** and **B** overlays **A**. **A** contains packages `foo` and `baz` where `baz` depends on `foo`. **B** contains packages `ping` that depends on `foo` and `pong` that depends on `baz`. **C** is the workspace being built. If you want to override `foo` then you should also override `baz`, `ping`, and `pong`.

9.1.3 Make sure overridden Python packages do not change entry point specifications

Python packages may have entry point specifications in a `setup.py` or `setup.cfg` file. Make sure the package in the overlay has identical specifications to the version in the underlay, or only adds new ones. If any specification has been changed or removed then it may not be possible to override this package.

9.2 How to make it easier for your users to override

This section has advice for package authors about how to make easier for your users to use your package and override it or other packages.

9.2.1 Install headers to a unique include directory

Install your package's headers to a unique folder rather than a shared folder. Say you're the author of a package `foo`, and it has a header meant to be included like `#include <foo/foo.hpp>`. Instead of installing the header to `<prefix>/include/foo/foo.hpp`, install it to `<prefix>/include/foo/foo/foo.hpp`.

If you have a CMake package `foo` with the directory structure `include/foo/foo.hpp`, then it can install its headers to a unique directory with this:

```
install(DIRECTORY include/ DESTINATION include/${PROJECT_NAME})
```

All exported targets in your project need to export the unique include directory too.

```
target_include_directories(some_library_in_foo INTERFACE
  "$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>"
  "$<INSTALL_INTERFACE:include/${PROJECT_NAME}>")
```

9.2.2 Dynamically link to libraries outside your package

If your package `foo` statically links to a library from package `bar`, then users can't override `bar` without also overriding yours. Prefer dynamic linking instead.

9.2.3 Handling Python entry point specifications

If your package loads Python entry points and it encounters two specifications with the same name, then it should use the last specification returned by entry_points(). It should also ignore entry points that can't be loaded.

Here's how to do it:

```
from importlib.metadata import entry_points

# Deduplicate entry point specifications before loading
deduplicated_entry_points = {}
# When faced with duplicates, this loop keeps the last entry point found
for ep in entry_points()['your_group_name']:
    deduplicated_entry_points[ep.name] = ep

for ep in deduplicated_entry_points:
    try:
        inst = ep.load()
    except ImportError:
        # Ignore entry point specifications that can't be loaded
        pass
```

9.3 All Known issues

9.3.1 Include Directory Search Order Problem

When overriding a package, it's possible for another package to find that package's headers from the underlay instead of the overlay. This may cause a failure to build or unexpected behavior at runtime depending on the differences between those headers.

Consider an overlay containing package `foo` and `bar`, and an underlay containing `bar` and `baz`. `foo` depends on `bar` and `baz`. Say the underlay is a merged workspace, and both the overridden `bar` and `baz` install their headers to a directory called `include/`. If any libraries or executables in `foo` are configured to search for headers in `baz`'s include directory first, then headers from overridden `bar` will also be found first.

When it can happen

- The underlay workspace is a merged workspace
- The overridden package installs header files (C/C++)
- The overriding package's headers are different from the overridden package's
- Another package in the underlay is not overridden and installs headers to the same directory as the overridden package (such as `include`)
- A package in the overlay depends on both the package being overridden and the aforementioned additional package in the underlay

How to avoid it

Use isolated workspaces

If your underlay is an isolated workspace, then no two packages in it will have the same include directory. Using an isolated overlay workspace won't help if your underlay is already a merged workspace (for example, the default ROS installation when installed from binary packages).

Sort include directories according to the workspace order

One solution is to sort include directories before passing them to the compiler such that headers are found in overlay workspaces before underlays. This requires making the build system for every package in an overlay aware of workspaces and the order they were sourced. The only known implementation of this is the `find_package(catkin COMPONENTS ...)` CMake feature in ROS 1.

Only override packages that install headers to unique include directories

If every package in the underlay installs their headers to unique directories, then packages in the overlay cannot accidentally find headers when depending on other packages in the underlay.

9.3.2 Unpredictable behavior when overridden package breaks API

Consider an overlay containing `bar`, and an underlay containing `bar` and `baz`. `baz` depends on `bar`. If `bar` in the overlay changed an API used by `baz`, then the behavior of `baz` at runtime can't be predicted.

When it can happen

- The overriding package removed or changed APIs compared to the overridden package
- A package in the underlay depends on the overridden package

How to avoid it

Build everything above the overridden package from source

If an API has changed, then every package in the underlay which depends on the overridden package (directly or indirectly) must be overridden too. You will need to find versions of those packages that are compatible with the API changes.

9.3.3 Undefined behavior when overridden package breaks ABI

Consider an overlay containing `bar`, and an underlay containing `bar` and `baz`. `baz` depends on `bar`. If `bar` in the overlay changed ABI, then it is unpredictable what will happen when `baz` is used at runtime.

When it can happen

- The overridden package uses a compiled language (C/C++, etc.)
- The overriding package is ABI incompatible with the overridden one

How to avoid it

Build everything above the overridden package from source

If ABI has changed, then every package in the underlay which depends on the overridden package (directly or indirectly) must be overridden too. If only ABI has changed, then no changes to the packages are needed. Compiling them again is enough.

9.3.4 Renamed or deleted Python modules still importable

Consider an overlay containing a Python package `pyfoo` and an underlay containing a Python package `pyfoo`. `pyfoo` in the underlay installs the Python modules `foo`, `foo.bar`, and `baz`. `pyfoo` in the overlay installs only the Python modules `foo`.

When the overlay is active, users will still be able to import `baz` from the underlay version of `pyfoo` However, they will not be able to import `foo.bar` because Python will find the `foo` package in overlay first, and that one does not contain `bar`.

When it can happen

- The package being overridden is a Python package
- The overridden package installs top level modules not present in the overriding package

How to avoid it

No workaround is known yet, but it's unlikely to cause problems unless combined with another issue.

9.3.5 One-definition rule violations caused by static linking

Consider an overlay containing packages `foo` and `bar`, and an underlay containing packages `bar` and `baz`. `foo` depends on `bar` and `baz`. `baz` depends on `bar` and has a library that statically links to another library in `bar`. `foo` has a library depending on both the mentioned library in `baz` and in `bar`.

When `foo` is used there are two definitions for symbols from `bar`: the ones from the underlay version of `bar` via `baz`, and the one from the overlay version of `bar`. At runtime, the implementations from the underlay version may be used.

When it can happen

- a package in the underlay statically links to the overridden package
- a package in the overlay depends on the overriding package and the package in the underlay

How to avoid it

Build everything above the overridden package from source

All packages directly or indirectly depending on the overridden package must be added to the overlay. No changes to the packages are needed. Compiling them again is enough.

9.3.6 Python entry_points are duplicated

Consider a package `pyfoo` that has an entry point specification `foobar = pyfoo.bar:baz`. If `pyfoo` is overridden and the overridden version has same specification, then the entry point will be listed twice. Whether or not it is a problem depends on how those entry points are loaded.

If the code loading entry points loads all of them without checking for duplicates, then the same entry points may be used twice.

When it can happen

- A python package providing entry points is overridden with a version that provides the same specification.

How to avoid it

There is no known workaround.

9.3.7 Deleted Python entry_points may still be loaded

Consider a package `pyfoo` that has an entry point specification `foobar = pyfoo.bar:baz`. say `pyfoo` is overridden and the overridden version does not have that specification.

If the specification is still importable, then entry points from the underlay may be run undesirably. If the specification is not importable, then the code loading them must gracefully handle import errors.

When it can happen

- A python package providing entry points is overridden with a version that omits an entry point available in the underlay.

How to avoid it

There is no known workaround.

CHAPTER 10

`build` - Build Packages

The `build` verb is building a set of packages. It is provided by the `colcon-core` package.

10.1 Command line arguments

These common arguments can be used:

- *executor* arguments
- *event handler* arguments
- *discovery* arguments
- *package selection* arguments
- *mixin* arguments

Additionally, the following specific command line arguments can be used:

–build-base BUILD_BASE The base path for all build directories. The default value is `./build`. Each package uses a subdirectory in that base path as its package specific build directory.

–install-base INSTALL_BASE The base path for all install prefixes. The default value is `./install`.

–merge-install Use the `--install-base` as the install prefix for all packages instead of a package specific subdirectory in the install base.

Without this option each package will contribute its own paths to environment variables which leads to very long environment variable values.

With this option most of the paths added to environment variables will be the same, resulting in shorter environment variable values.

The disadvantage of using this option is that it doesn't provide proper isolation between packages. For example declaring a dependency on one package also allows access to resources from other packages installed in the same install prefix (without requiring a declared dependency).

Note: on Windows using `cmd` this argument should be used for workspaces with many packages otherwise the environment variables might exceed the supported maximum length.

–symlink-install Use symlinks instead of copying files from the source and build directories where possible.

–test-result-base TEST_RESULT_BASE The base path for all test results. The default value is the `--build-base` argument. Each package uses a subdirectory in that base path as its package specific test result directory.

–continue-on-error Continue building other packages when a package fails to build. Packages recursively depending on the failed package are skipped.

10.1.1 CMake specific arguments

The following arguments are provided by the `colcon-cmake` package:

–cmake-args [* [* ...]] Pass arbitrary arguments to CMake projects. Arguments matching other options must be prefixed by a space, e.g. `--cmake-args " --help"`.

–cmake-target CMAKE_TARGET Build a specific target instead of the default target. To avoid packages which don't have that target causing the build to fail, also pass *–cmake-target-skip-unavailable*.

–cmake-target-skip-unavailable Skip building packages which don't have the target passed to *–cmake-target*.

–cmake-clean-cache Remove the CMake cache file `CMakeCache.txt` from the build directory before proceeding with the build. This implicitly forces a CMake configure step.

–cmake-clean-first Build the target `clean` first, then proceed with a regular build. To only invoke the clean target use *–cmake-target clean*.

–cmake-force-configure Force CMake configure step.

10.1.2 ROS `ament_cmake` specific arguments

The following arguments are provided by the `colcon-ros` package:

–ament-cmake-args [* [* ...]] Pass arbitrary arguments to ROS packages with the build type `ament_cmake`. Arguments matching other options must be prefixed by a space, e.g. `--ament-cmake-args " --help"`.

10.1.3 ROS `catkin` specific arguments

The following arguments are provided by the `colcon-ros` package:

–catkin-cmake-args [* [* ...]] Pass arbitrary arguments to ROS packages with the build type `catkin`. Arguments matching other options must be prefixed by a space, e.g. `--catkin-cmake-args " --help"`.

–catkin-skip-building-tests By default the `tests` target building the tests in `catkin` packages is invoked. If running `colcon test` later isn't intended this can be skipped.

CHAPTER 11

`edit` - Edit File

The `edit` search and edit a file inside a package using command line. It is provided by the `colcon-ed` package.

When a correct package name and file name is provided, it will open that file with an editor.

```
$ colcon edit <package_name> <file_name>
```

Note: To enable auto-complete, check out the *installation guide*.

Hidden files and `.pyc` files are ignored. If multiple files under the same name are present, they will all be listed and you can type in the index number to select which file to edit.

By default, `vim` is the editor that will be used, but you can also use other editors by specifying `$EDITOR` environment variable to the ones preferred. For example, to use **Visual Studio Code**, simply run

```
$ export EDITOR=code
```

11.1 Command line arguments

These common arguments can be used:

- *discovery* arguments
- *mixin* arguments

The following specific positional arguments are used:

PKG_NAME Explicit package name to specify which package the file is in.

FILE_NAME File name to specify which file to edit.

CHAPTER 12

`graph` - Visualize Dependencies

The `graph` verb generates a visual representation of the package dependency graph. It is provided by the `colcon-package-information` package.

For each package, the path, name and type is shown. By default it outputs a topologically ordered list of packages using ASCII art to indicate direct as well as transitive dependencies.

Optionally the verb can generate DOT code to render a graphical representation.

12.1 Command line arguments

These common arguments can be used:

- *discovery* arguments
- *package selection* arguments
- *mixin* arguments

Additionally, the following specific command line arguments can be used:

–density Output the density of the dependency graph. This option is only supported without `--dot`.

–legend Output a legend for the dependency graph.

–dot Output topological graph in DOT. Commonly the output should be piped to `dot`, e.g. `| dot -Tpng -o graph.png`.

The differently colored edges represent the dependency types: blue=build, red=run, tan=test. Dashed edges are indirect dependencies which only appear if some packages are being skipped.

–dot-cluster Cluster packages by their file system path. This option only affects `--dot`.

–dot-include-skipped Include skipped packages in the rendered graph with a gray color. This option only affects `--dot`.

12.2 Example Output

12.2.1 ASCII

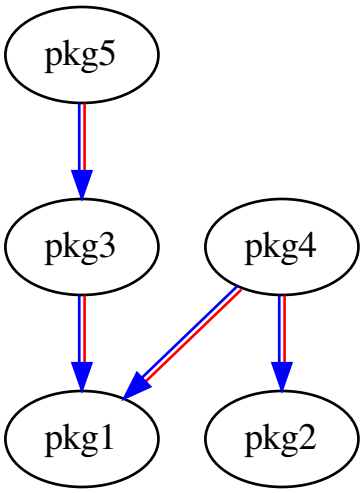
Example output of invoking `colcon graph` with `--legend` (which adds the first paragraph describing the symbols).

```
+ marks when the package in this row can be processed
* marks a direct dependency from the package indicated
  by the + in the same column to the package in this row
. marks a transitive dependency

pkg1  + **.
pkg2   + *
pkg3    + *
pkg4     +
pkg5      +
```

12.2.2 DOT

Example output of invoking `colcon graph` with `--dot` rendered into an image. In this example the packages have the same build (blue edges) and run (red edges) dependencies.



CHAPTER 13

`info` - Show Package Information

The `info` verb shows detailed information about packages. It is provided by the `colcon-package-information` package.

For each package a set of lines is shown starting with the path of the package. Additionally the package type, name, build/run/test dependencies and metadata like the package version are enumerated.

By default the information for all packages is shown. Optionally, single or multiple package names can be passed to only show their information.

13.1 Command line arguments

These common arguments can be used:

- *discovery* arguments
- *package selection* arguments
- *mixin* arguments

Additionally, the following specific command line arguments can be used:

PKG_NAME [PKG_NAME ...] Explicit package names to only show their information. This positional argument is redundant with the package selection argument `--packages-select` and only exists for ease of use.

CHAPTER 14

`list` - List Packages

The `list` verb enumerates a set of packages. It is provided by the `colcon-package-information` package.

For each package a line is shown containing the path, name and type separated by tabs. By default the list is ordered alphabetically by the package name. Optionally, it can order the packages topologically based on their dependencies.

14.1 Command line arguments

These common arguments can be used:

- *discovery* arguments
- *package selection* arguments
- *mixin* arguments

Additionally, the following specific command line arguments can be used:

–topological-order, -t Order output based on topological ordering (breadth-first). For more detailed visualizations of dependencies see the *graph* verb.

–names-only, -n Output only the name of each package but not the path and type

–paths-only, -p Output only the path of each package but not the name and type

CHAPTER 15

`metadata` - Manage metadata

The `metadata` verb offers multiple subverbs to manage the registration, introspection and update of metadata repositories.

15.1 metadata `add`

Add the URL of a metadata repository. Afterwards, the metadata information needs to be fetched at least once by invoking `colcon metadata update <name>`.

name The name to identify the to-be-added repository.

url The URL pointing to a yaml file containing an index enumerating `.meta` files.

15.2 metadata `list`

Enumerate the name and URL of the registered metadata repositories as well as their metadata files.

[name] An optional name to only enumerate the metadata files of a specific repository.

15.3 metadata `remove`

Remove the URL of a metadata repository and all its metadata files.

name The name to identify the to-be-removed repository.

15.4 metadata `update`

Fetch the metadata files from a specific repository. The output indicates which metadata files have been added, updated, are unchanged or are obsolete. Obsolete metadata files are not deleted but renamed (changing the `.meta` extension to

`.obsolete`).

[name] An optional name to only update the metadata files of a specific repository.

CHAPTER 16

`mixin` - Manage mixins

The `mixin` verb offers multiple subverbs to manage the registration, introspection and update of mixin repositories.

16.1 mixin `add`

Add the URL of a mixin repository. Afterwards the mixin information needs to be fetched at least once by invoking `colcon mixin update <name>`.

name The name to identify the to-be-added repository.

url The URL pointing to a yaml file containing an index enumerating `.mixin` files.

16.2 mixin `list`

Enumerate the name and URL of the registered mixin repositories as well as their mixin files.

[name] An optional name to only enumerate the mixin files of a specific repository.

16.3 mixin `remove`

Remove the URL of a mixin repository and all its mixin files.

name The name to identify the to-be-removed repository.

16.4 mixin `update`

Fetch the mixin files from a specific repository. The output indicates which mixin files have been added, updated, are unchanged or are obsolete. Obsolete mixin files are not deleted but renamed (changing the `.mixin` extension to `.obsolete`).

[name] An optional name to only update the mixin files of a specific repository.

CHAPTER 17

`test` - Test Packages

The `test` verb runs the tests for a set of packages. It is provided by the `colcon-core` package.

17.1 Command line arguments

These common arguments can be used:

- *executor* arguments
- *event handler* arguments
- *discovery* arguments
- *package selection* arguments
- *mixin* arguments

Additionally, the following specific command line arguments can be used:

–build-base BUILD_BASE The base path for all build directories. The default value is `./build`. Each package uses a subdirectory in that base path as its package specific build directory.

–install-base INSTALL_BASE The base path for all install prefixes. The default value is `./install`.

–merge-install Use the `--install-base` as the install prefix for all packages instead of a package specific subdirectory in the install base. See *here* for more information.

–test-result-base TEST_RESULT_BASE The base path for all test results. The default value is the `--build-base` argument. Each package uses a subdirectory in that base path as its package specific test result directory.

–retest-until-fail N Rerun tests up to N times if they pass.

–retest-until-pass N Rerun failing tests up to N times.

–abort-on-error Abort after the first package with any errors. Failing tests are not considered errors in this context.

–return-code-on-test-failure Use a non-zero return code to indicate any test failure.

17.1.1 CMake specific arguments

–ctest-args [* [* ...]] [colcon-cmake] Pass arbitrary arguments to CTest projects. Arguments matching other options must be prefixed by a space, e.g. `--ctest-args " --help"`.

17.1.2 Python specific arguments

–python-testing {pytest,setuppy_test} Choose the Python testing framework to use. The default value is determined based on the packages `tests_require` option.

- `pytest`: Use pytest to test Python packages.
- `setuppy_test`: Use `setup.py test` to test Python packages.

–pytest-args [* [* ...]] Pass arbitrary arguments to Python packages using `pytest`. Arguments matching other options must be prefixed by a space, e.g. `--pytest-args " --help"`.

–pytest-with-coverage Generate coverage information in the package specific build directory. By default, coverage information is only generated for Python packages that declare a test dependency on `pytest-cov`.

CHAPTER 18

`test-result` - Summarize Test Results

The `test-result` verb summarizes the results of previous run tests. It is provided by the `colcon-test-result` package.

18.1 Command line arguments

These common arguments can be used:

- *mixin* arguments

Additionally, the following specific command line arguments can be used:

–test-result-base TEST_RESULT_BASE The base path for all test results. The default value is `./build`. Each package uses a subdirectory in that base path as its package specific test result directory.

–all Show all test result files including the ones without errors / failures.

–verbose Show additional information for each error / failure.

–result-files-only Print only the paths of the result files. Use with `--all` to include files without errors / failures.

–delete Delete all result files. This might include additional files beside what is listed by `--result-files-only`. An interactive prompt will ask for confirmation.

–delete-yes Same as `--delete` without an interactive confirmation.

CHAPTER 19

Global arguments

19.1 Command line arguments

The following command line arguments can be used *before* every verb. For each argument the name in brackets indicates which package contributes it.

–log-base LOG_BASE [colcon-core] The base path for all log directories. The default value is `./log`. To completely disable logging, pass `/dev/null` (on POSIX) / `nul` (on Windows).

Within the base path all log files from a specific invocation are places in a subdirectory named `<verb>_<timestamp>` (in the following named `log-dir`). If the system supports creating symlinks, shortcuts with the names `latest_<verb>` and `latest` are created (overwriting previous symlinks).

Within the `log-dir` the following files are generated:

- `events.log` [`colcon-output`]: All generated events - only used for debugging.
- `logger_all.log` [`colcon-core`]: All log messages, independent of the chosen log level for the console output.
- For each processed package a subdirectory with the following files is generated:
  - `command.log` [`colcon-output`]: All invoked external commands including their return code.
  - `stderr.log` [`colcon-output`]: All output from external commands to `stdout`.
  - `stdout.log` [`colcon-output`]: All output from external commands to `stderr`.
  - `stdout_stderr.log` [`colcon-output`]: A combination of `stdout.log` and `stderr.log` in the order the two streams were processed. Since both streams are read concurrently the order is not fully deterministic.
  - `streams.log` [`colcon-output`]: A combination of the command log as well as both output logs. Each line is being prefixed with the elapsed time since the package started being processed.

–log-level LOG_LEVEL [colcon-core] Set the log level for the console output. The log file `<log-dir>/logger_all.log` always contains messages of all levels. The value can either be a numeric value or a string as defined in the Python logging module (the names are case insensitive). The default value is `warning`.

19.2 Environment variables

The following environment variables are supported. The name in brackets indicates which package contributes it.

CMAKE_COMMAND [colcon-cmake] The full path to the CMake executable. By default the executable `cmake` is being searched for on the `PATH`.

COLCON_ALL_SHELLS [colcon-core] Flag to enable all shell extensions. If the environment variable is set and not empty all shell extensions are being used even if they are supposed to be skipped based on the current platform.

COLCON_ANIMATION_PROGRESS [colcon-graphviz-anim] Flag to generate an animation of the task progress. If the environment variable is set and not empty the file `./graphviz_anim_build.gif` is generated.

COLCON_COMPLETION_LOGFILE [colcon-argcomplete] Set the path of a logfile to append the completion time to. If the environment variable is set and not empty a line is appended to the specified file. The line contains the duration it took to compute the completion choices as well as the value of the environment variable `COMP_LINE` which was passed to determine the completion choices.

COLCON_DEFAULTS_FILE [colcon-defaults] Set the path to the yaml file containing the default values. By default the path *$COLCON_HOME*`/defaults.yaml` is being used.

COLCON_DEFAULT_EXECUTOR [colcon-core] Select the default executor extension for verbs which support it. By default the executor with the highest priority is used. The executor can also be chosen using the `--executor` argument.

COLCON_EXTENSION_BLOCKLIST [colcon-core] Block extensions which should not be used. The value uses a path separator to enumerate entry point group names or full entry point names.

COLCON_HOME [colcon-core] Set the base path of configuration files. By default the path `~/.colcon` is being used.

COLCON_LOG_LEVEL [colcon-core] Set the log level for the console output. See the *–log-level* arguments for details about the possible values. Using the environment variable instead of the command line argument allows setting the log level *before* the argument parsing has been performed.

COLCON_WARNINGS [colcon-core] Set the warnings filter similar to PYTHONWARNINGS except that the module entry is implicitly set to `colcon.*`

CTEST_COMMAND [colcon-cmake] The full path to the CTest executable. By default the executable `ctest` is searched for on the `PATH`.

POWERSHELL_COMMAND [colcon-powershell] The full path to the PowerShell executable. By default the executable `PowerShell` (on Windows) / `pwsh` (on non-Windows) is searched for on the `PATH`.

CHAPTER 20

Executor arguments

The executor is responsible for processing jobs in verbs which have more than one item of work. For each argument the name in brackets indicates which package contributes it.

–executor EXECUTOR [colcon-core] The executor to process all jobs. The default is chosen based on the priorities of all available executor extensions. To see a complete list invoke `colcon extensions colcon_core.executor --verbose`.

- `sequential` [`colcon-core`]

  Process one package at a time.

- `parallel` [`colcon-parallel-executor`]

  Process multiple jobs in parallel.

–parallel-workers NUMBER [colcon-parallel-executor] The maximum number of jobs to process in parallel. The default value is the number of logical CPU cores as reported by os.cpu_count(). This option only affects `--executor parallel`.

CHAPTER 21

Event handler arguments

The event handlers are used by several verbs to generate any kind of output based on the progress of the invocation. For each argument the name in brackets indicates which package contributes it.

–event-handlers [name1+ [name2- ...]] [colcon-core] A list of event handlers to enable (trailing +) or disable (trailing −). The default is chosen by each available event handler (see `colcon <verb> --help`).

- `console_cohesion` [`colcon-output`]:

  Pass job output at once to `stdout` after it has finished.

- `console_direct` [`colcon-core`]:

  Pass output directly to `stdout` / `stderr`. When processing multiple jobs in parallel the output will likely interleave.

- `console_package_list` [`colcon-output`]:

  Output a list of queued job names.

- `console_start_end` [`colcon-core`]:

  Output each job name when starting / ending.

- `console_stderr` [`colcon-output`]:

  Output all `stderr` of a job at once after it has finished.

- `desktop_notification` [`colcon-notification`]:

  Enable desktop notification of the summary indicating the result of the invocation.

- `event_log` [`colcon-output`]:

  Log all events to a global log file `events.log`. See *here* for more information about the location of that log file.

- `graphviz_anim` [`colcon-graphviz-anim`]:

  Generate a `.gif` file of the job progress after the invocation has finished. See *here* for more details.

- `log` [`colcon-output`]:

  Output job specific log files containing all the output of invoked commands. See *here* for more information about the location of these log files.

- `log_command` [`colcon-colcon`]:

  Log a debug message for each invoked command. Either *set a custom log level* to show debug messages or check the generated log files.

- `status` [`colcon-notification`]:

  Show a status line at the current cursor in the terminal title and update it continuously.

- `store_result` [`colcon-package-select`]:

  Persist the result of a job in a file in its build directory which allows the information to be used in subsequent invocation to select packages based on the previous result.

- `summary` [`colcon-output`]:

  Output a single line summarizing the result of all jobs after the invocation has finished.

- `terminal_title` [`colcon-notification`]:

  Show the ongoing status in the terminal title.

CHAPTER 22

Discovery arguments

Discovery arguments determine which locations should be checked if they contain packages. For each argument the name in brackets indicates which package contributes it.

–paths [PATH [PATH ...]] [colcon-core] The (non recursive) paths to check for a package. Use shell wildcards (e.g. `src/*`) to select all direct subdirectories of `src`.

–base-paths [PATH [PATH ...]] [colcon-recursive-crawl] The base paths to recursively crawl for packages. The default value is `..` In a workspace root the subdirectories other than `src` (commonly `build`, `install`, `log`) contain a `COLCON_IGNORE` marker file which causes them to be ignored.

–metas [PATH [PATH ...]] [colcon-metadata] The directories containing a `colcon.meta` file or paths to arbitrary files containing the same meta information. The default value is `./colcon.meta`.

–ignore-user-meta [colcon-metadata] Ignore `*.meta` files in the configuration directory *$COLCON_HOME*`/metadata/`.

–packages-ignore [PKG_NAME [PKG_NAME ...]] [colcon-package-selection] Ignore packages by name as if they were not discovered. In contrast to being skipped using *package selection* arguments, ignored packages aren't considered in the dependency graph.

–packages-ignore-regex [PATTERN [PATTERN ...]] [colcon-package-selection] Ignore packages where any of the patterns match the package name. In contrast to being skipped using *package selection* arguments, ignored packages aren't considered in the dependency graph.

CHAPTER 23

Package selection arguments

Package selection arguments determine the set of packages which should be processed by a given verb. If multiple arguments are passed the resulting set is a logical `AND` combination. For each argument the name in brackets indicates which package contributes it.

–build-base BUILD_BASE The base path for all build directories. The default value is `./build`. Each package uses a subdirectory in that base path as its package specific build directory.

For some verbs this argument only exists since some of the package selection arguments use state information from previous builds to select / skip packages.

–packages-up-to [PKG_NAME [PKG_NAME . . .]] Select the packages with the passed names as well as their recursive dependencies.

–packages-up-to-regex [PATTERN [PATTERN . . .]] Select the packages that match any of the passed patterns as well as their recursive dependencies.

–packages-above [PKG_NAME [PKG_NAME . . .]] Select the packages with the passed names as well as packages which recursively depend on them.

–packages-above-and-dependencies [PKG_NAME [PKG_NAME . . .]] Select the packages with the passed names, packages which recursively depend on them as well as their recursive dependencies.

–packages-above-depth DEPTH [PKG_NAME . . .] Select the packages with the passed names as well as packages which recursively depend on them up to the given depth.

–packages-select-by-dep [DEP_NAME [DEP_NAME . . .]] Select packages which (recursively) depend on the given packages.

–packages-skip-by-dep [DEP_NAME [DEP_NAME . . .]] Skip packages which (recursively) depend on the given packages.

–packages-skip-up-to [PKG_NAME [PKG_NAME . . .]] Skip the packages with the passed names as well as their recursive dependencies.

–packages-select-build-failed Select packages which have failed to build previously. Packages which have been aborted previously are not included.

–packages-skip-build-finished Skip packages which have finished to build previously.

–packages-select-test-failures Select packages which had test failures previously.

–packages-skip-test-passed Skip packages which had no test failures previously.

–packages-select [PKG_NAME [PKG_NAME . . .]] Select the packages with the passed names.

–packages-skip [PKG_NAME [PKG_NAME . . .]] Skip the packages with the passed names.

–packages-select-regex [PATTERN [PATTERN . . .]] Select the packages where any of the patterns match the package name.

–packages-skip-regex [PATTERN [PATTERN . . .]] Skip the packages where any of the patterns match the package name.

–packages-start PKG_NAME Skip packages before the given package name in flat topological ordering. This option only makes sense when using the *sequential executor*.

–packages-end PKG_NAME Skip packages after the given package name in flat topological ordering. This option only makes sense when using the *sequential executor*.

CHAPTER 24

Mixin arguments

Mixins can be used by several verbs to contribute command line arguments defined in external files.

The following arguments are provided by the `colcon-mixin` package:

–mixin-files [FILE [FILE . . .]] Read additional mixin files and make the mixin names specified in them available in the `--mixin` argument.

–mixin [mixin1 [mixin2 . . .]] The names of mixins to be used. The list of mixin names and their command line arguments depends on which ones are available. To enumerate the available verb specific mixins and their command line arguments invoke `colcon mixin show <verb>`.

An example mixin provided in the colcon-mixin-repository repository:

- **debug**:
  - `cmake-args`: `['-DCMAKE_BUILD_TYPE=Debug']`

When multiple mixins are used they are interpreted in order with later mixin values replacing or extending previous values. In case of lists the items are being concatenated. In all other cases the latter value replaces the former value.

Warning: At the moment the logic concatenating lists has no semantic knowledge of the data. Therefore the joined list might not have the desired semantic meaning. E.g. for the following two lists:

- `cmake-args`: `['-DCMAKE_C_FLAGS=-fPIC']`
- `cmake-args`: `['-DCMAKE_C_FLAGS=-g']`

the joined list will be just a concatenation:

- `cmake-args`: `['-DCMAKE_C_FLAGS=-fPIC', '-DCMAKE_C_FLAGS=-g']`

But passing these arguments to CMake would result in the latter value of `CMAKE_C_FLAGS` overwriting the former even though the user likely wanted both compiler options to be used.

CHAPTER 25

Design

25.1 Goals for colcon

A few high level goals are used to guide the overall development.

- The tool should make building, testing and using multiple packages easy.
- It should be possible to add support for any kind of build system using extensions. `colcon-core` only bundles Python support in order to bootstrap itself.
- It should be possible to build any set of packages without requiring changes to their sources. If necessary missing information can be provided externally.
- After building packages they must be immediately usable which includes setting up necessary environment variables etc.

25.1.1 Explicitly out of scope

The tool does not aim to address any of the following tasks. Those should be left for other tools to take care of them.

- Fetch the source of the packages which should be processed by `colcon`.
- Install dependencies of the packages which should be processed by `colcon`.
- Perform packaging tasks like creating Debian packages.

Note: While these items are specifically not targeted by `colcon` it is still possible to implement support for any of these features (or helpful functionality to integration with existing tools) in an extension.

25.2 Software design

Additionally some software design goals are stated:

- All the functionality provided should be exposed in a way that it can be reused by other extensions.
- The separation into multiple Python packages is being used to encourage modularity and loose coupling (Law of Demeter). It is also used to demonstrate extensibility and show that certain features are not "special" but can be contributed externally.
- Each component should have responsibility over a single part of the software (Single responsibility principle).
- Each functionality added should follow the principle "you don't pay for what you don't use".

CHAPTER 26

Bootstrap from source

When developing `colcon` you want to have a local checkout of all involved packages.

Note: The following steps use the command line tool vcstool to fetch a set of repositories. You can e.g. install it using `pip install vcstool`.

Note: While the following instructions use a Linux shell the same can be done on other platforms like Windows with slightly adjusted commands.

26.1 Virtual environment

While not strictly necessary it is recommended to use a virtual environment for developing Python packages.

```
$ mkdir colcon-venv
$ python3 -m venv colcon-venv
$ . colcon-venv/bin/activate
```

Note: On Windows the Python 3 executable is likely named `python` and the activation script is invoked with `colcon-venv\Scripts\activate`

You might want to make sure that the venv is using up-to-date versions of the some foundational packages.

```
$ pip install -U pip setuptools
```

26.2 Fetch the sources

```
$ mkdir colcon-from-source && cd colcon-from-source
$ curl --output colcon.repos https://raw.githubusercontent.com/colcon/colcon.
↪readthedocs.org/main/colcon.repos
$ mkdir src
$ vcs import src < colcon.repos
```

Note: Depending on your platform you might not want to use all cloned packages. On Windows you must ignore or remove `colcon-argcomplete`, and may want to do the same for `colcon-bash`. If you don't use PowerShell you might want to ignore / remove the package `colcon-powershell`. To ignore a package add an empty file named COLCON_IGNORE to the folder.

Ignore `colcon-argcomplete` and `colcon-bash` on Windows.

```
> type nul > src\colcon-argcomplete\COLCON_IGNORE
> type nul > src\colcon-bash\COLCON_IGNORE
```

26.3 Dependencies

Make sure the dependencies are available:

```
$ curl --output requirements.txt https://raw.githubusercontent.com/colcon/colcon.
↪readthedocs.org/main/requirements.txt
$ pip install -r requirements.txt
```

26.4 Build the sources - first time

In the first build we will use the minimal features provided by `colcon-core` to build the set of cloned packages.

```
$ ./src/colcon-core/bin/colcon build --paths src/*
```

Note: On Windows the command needs to be prefixed with `python`.

The build of the packages will run sequentially and for each package the output will be printed directly to the console. The install directory will contain a `local_setup.sh` (or `.bat` on Windows).

In order to generate scripts for additional shells the set of packages have to be built a second time but this time using all extension provided by the various cloned packages.

26.5 Build the sources - second time

```
$ . install/local_setup.sh
$ colcon build
```

Note: On Windows the setup file ends with `.bat` and is just being called. Also the `colcon` executable can't be invoked directly here since while it is being used it can't be overwritten by the build. Instead invoke the following command: `python install\colcon-core\Scripts\colcon-script.py build`.

Note: The second build will process packages in parallel as long as their dependencies are finished. Also the output of all packages is not shown on the console (until there are errors) but is being redirected to log files. Depending on the platform you might also notice a status line during the build, a continuously updated title of the shell windows, and a desktop notification at the end of the build.

To use the full functionality you can source the generated script for your shell:

```
$ . install/local_setup.bash
```

Note: With bash you should now also have completion for all arguments if you have the Python package argcomplete installed. Try typing `colcon <tab>` to see the completion of global options and verbs.

CHAPTER 27

Environment setup

Using a package after it has been built or building a package on top of its dependencies might require updating environment variables. For the former, the best option for the user is to have a script perform the environment update for ease of use. In the latter case, automating the process is necessary to build packages in topological order without user interaction. E.g. if a package installs an executable which should be invocable by name, the directory containing the executable needs to be part of the `PATH` environment variable.

27.1 Entry points

To update environment variables:

- On non-Windows platforms a shell script needs to be `source`-ed (e.g. `.sh`, `.bash`, `.zsh`)
- On Windows a shell script needs to be `call`-ed (e.g. `.bat`, `.ps1`)

Note: Executables are unable to change the environment of the current shell. Therefore this needs to be done from a shell script.

27.1.1 Package-level

Note: Each package installs its files under its install prefix. This corresponds to `install/<package_name>` except when `colcon build --merge-install` is used. In that case, the install prefix is `install/`.

For each built package `colcon` generates a set of package-level scripts (one for each supported shell type): `share/<package_name>/package.<ext>`. These script files update the environment with information specific to this package.

`colcon` supports multiple different approaches to update environment variables:

- A heuristic checking the installed files for known file types or known destinations. A few of the supported heuristics:
  - An executable under `bin` adds that directory to the `PATH`
  - An existing Python library directory adds that directory to the `PYTHONPATH`
  - A file ending in `-config.cmake` or `Config.cmake` adds the directory to the `CMAKE_PREFIX_PATH` (provided by `colcon-cmake`)
  - A file starting with `Find` and ending in `.cmake` adds the directory to the `CMAKE_MODULE_PATH` (provided by `colcon-cmake`)
  - A shared library adds the directory to one of the following environment variables (depending on the platform): `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH`, `PATH` (provided by `colcon-library-path`)
  - A file ending in `.pc` in a directory `lib/pkgconfig` adds the directory to the `PKG_CONFIG_PATH` (provided by `colcon-pkg-config`)
- Specific package types providing their own scripts to setup the package specific environment. E.g. `ament_cmake` packages (supported by `colcon-ros`) provide a file named `share/<package_name>/local_setup.<ext>`.

27.1.2 Workspace-level

In the root of the install prefix path, `colcon` generates two kinds of scripts:

- `local_setup.<ext>`: updates the environment with information from all packages installed under this prefix path. Since package-level scripts rely on information from their dependencies the package-level scripts must be invoked in topological order. In order to determine the topological order of all packages under the prefix path, `colcon` stores all run dependencies of a package in a file named `share/colcon-core/packages/<package_name>`.
- `setup.<ext>`: first invokes the `local_setup.<ext>` files from all parent prefix paths and then invokes the sibling `local_setup.<ext>` file.

27.2 Different shells

Each shell has a potentially different syntax and supports a different set of features. Some environment changes like extending the `PATH` are applicable to all shells while others like providing completion functionality are only applicable to some. If one shell (e.g. `bash`) provides a superset of functionality and syntax of another shell (e.g. `sh`) the logic of the `.sh` scripts doesn't have to be duplicated for `bash` but can be invoked from the package-level setup files. The latter shell is called a *primary shell*.

27.2.1 Primary shells

All environment changes necessary to use a package should be expressed in script with the extension of a primary shell. Primary shell extensions are:

- `.sh`: plain shell
- `.bat`: Windows cmd
- `.ps1`: PowerShell

27.2.2 Non-primary / additional shells

Other shells which are able to invoke primary shell scripts within their context commonly don't duplicate their content and logic. Instead they first invoke the primary shell script and then add additional logic to provide specific functionalities like completion.

27.3 Avoid duplicate entries

Several environment variables store multiple values separated by a delimiter. Commonly, duplicate values are not useful and only increase length and decrease readability. Therefore, shell scripts try to avoid adding duplicate values where applicable.

With the number of values in such an environment variable the cost to check if a given value is already in the collection increases. This significantly affects the time it takes to `source` / `call` the workspace-level setup file since for each attempt to update an environment variable the collection needs to be split and each existing value compared to the to-be-added value.

Therefore `colcon` provides an alternative way to update the environment.

27.3.1 .dsv files

Since shell scripts can contain arbitrary logic it is opaque from the outside how they affect the environment. That makes it impossible to optimize their execution.

Most scripts perform one of the following common operations:

- Set an environment variable to a value.
- Add a value to an environment variable (using a platform specific delimiter) if the value is not already in the collection.
- Source / call another script file.

A `.dsv` file contains the descriptive information about the intended environment change (instead of shell specific logic). The content of such a file uses a semicolon as the delimiter and contains a single line. The first value is the `type` of the operation followed by a variable number of arguments specific to the operation.

The following list enumerates the supported types and their arguments:

- `prepend-non-duplicate;<name>;<value>`: Prepend a value `<value>` to an environment variable `<name>` (using a platform specific delimiter) if the value is not already in the collection. The value is considered to be a path. If the value is not an absolute path the prefix path of the `.dsv` file is prepended to the value. An empty value therefore represents the prefix path.
- `prepend-non-duplicate-if-exists;<name>;<value>`: Same as `prepend-non-duplicate` but only if the path described by the value exists.
- `set;<name>;<value>`: Set an environment variable `<name>` to a value `<value>`. If the value is an existing relative path in the install prefix the install prefix is prepended to the value. Otherwise the value is used as is.
- `set-if-unset;<name>;<value>`: Same as `set` but only if the environment variable is not yet set (or empty).
- `source;<path>`: Source / call another script file `<path>`. If the value is not an absolute path the prefix path of the `.dsv` file is prepended.

27.4 Implementation

Implementing the logic to determine the topological order of packages in every primary shell would be a lot of effort and (depending on the shell) cumbersome. Also parsing and interpreting `.dsv` files would likely not be much faster than invoking the native scripts.

Therefore both parts are implemented in a Python script located in the root of the install prefix: `_local_setup_util_<ext>.py`. The Python script itself can't change the environment. However, it is able to efficiently interpret the operations described by the `.dsv` files and generate the shell specific commands necessary to update the environment. The Python file is templated with information specific to the primary shell it's used from, hence the `<ext>` in the filename.

27.5 Tracing

When sourcing / calling a workspace-level setup file the number of evaluated scripts and / or interpreted `.dsv` files can be significant. To debug what files are being considered in which order and what environment changes are being performed you can prepend the invocation with `COLCON_TRACE=1`. As a result each recursively invoked script as well as every generated command will be printed to the terminal.

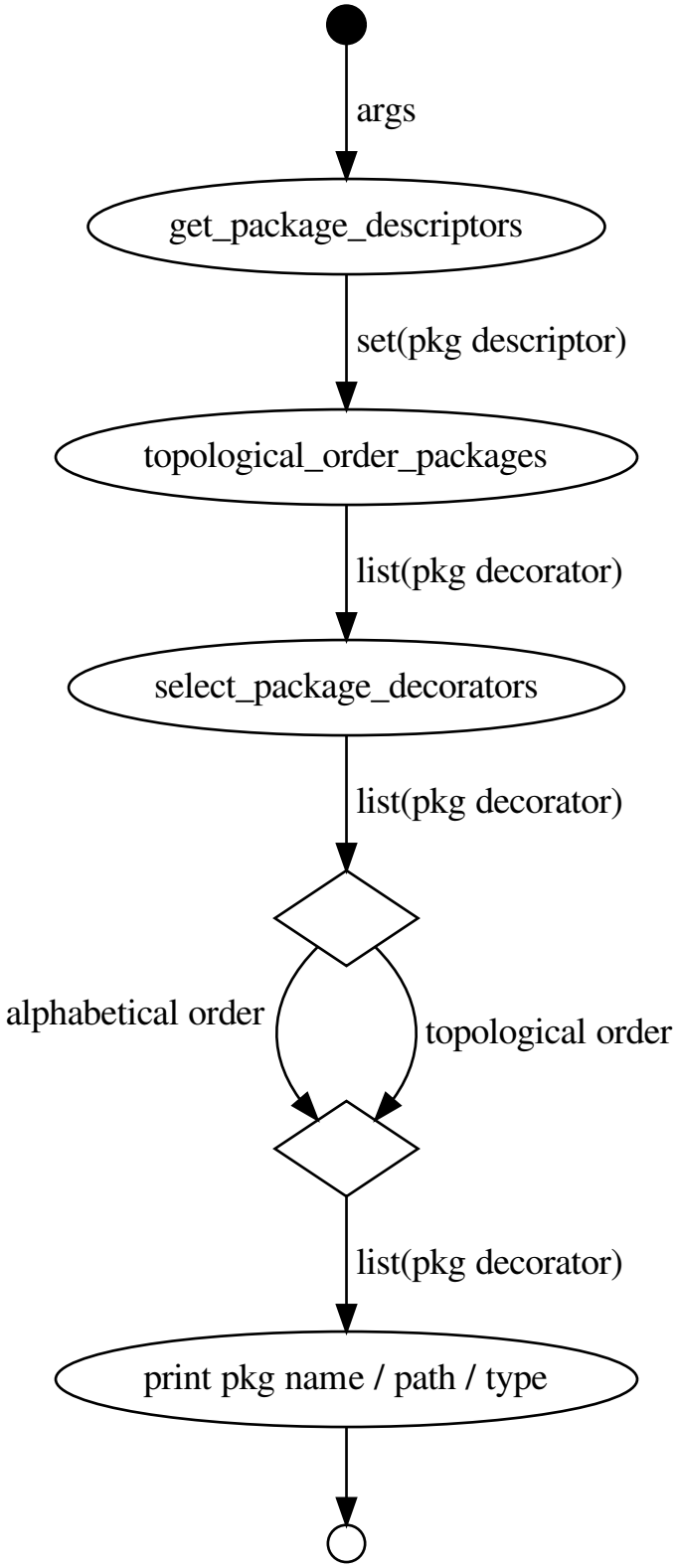
CHAPTER 28

Program flow

The following depicts the high level program flow of a couple of verbs.

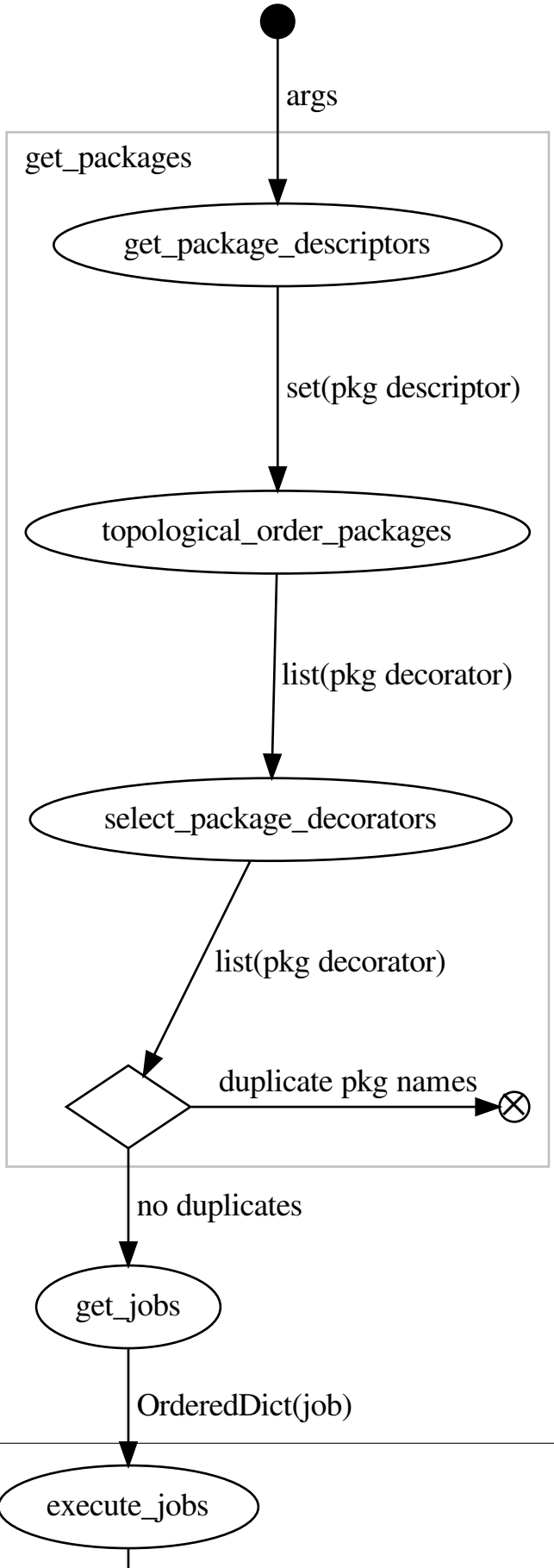
28.1 list

This verb is provided by the `colcon-package-information` package.


args
get_package_descriptors
set(pkg descriptor)
topological_order_packages
list(pkg decorator)
select_package_decorators
list(pkg decorator)
alphabetical order
topological order
list(pkg decorator)
print pkg name / path / type

28.2 build

This verb is provided by the `colcon-core` package.


args
get_packages
get_package_descriptors
set(pkg descriptor)
topological_order_packages
list(pkg decorator)
select_package_decorators
list(pkg decorator)
duplicate pkg names
no duplicates
get_jobs
OrderedDict(job)
execute_jobs

CHAPTER 29

Extension points

The following describes some of the extension points and how they are used.

29.1 VerbExtensionPoint

This extension point is used by the `colcon` command. Each verb extension defines custom logic which can be invoked by calling `colcon <verb>`. Each verb extension can add command line arguments to the verb specific subparser in `argparse`.

29.2 PackageDiscoveryExtensionPoint

This extension point is used by verb extensions. Each package discovery extension can add command line arguments which are grouped in the "Discovery arguments" section of the usage help. Each extension returns a set of package descriptors which identify a package with the tuple path, name, and type. While a package discovery extension only provides candidate paths it uses the package identification extension point to determine if a path contains a package.

If explicit arguments are provided for any package discovery extensions only those extension are being used. Otherwise the package discovery extensions which provide a default value are used.

29.3 PackageIdentificationExtensionPoint

This extension point is used by package discovery extensions. A package identification extension determines if a given path contains a package and returns the tuple path, name, and type.

Package identification extensions are grouped by their priority. The extensions are invoked in order of their priority and the first extension successfully identifying a package stops further extensions from being considered.

If multiple extensions have the same priority they are all invoked. If more than one extension within the priority group identifies the package in the given path the result must be identical or a warning is printed and the path is skipped.

29.4 PackageAugmentationExtensionPoint

This extension point is used by verb extensions after packages have been discovered and identified. A package augmentation extension can add arbitrary information to a package descriptor, like additional dependencies.

29.5 PackageSelectionExtensionPoint

This extension point is used by verb extensions after packages have been discovered, identified, and augmented. Each package selection extension adds command line arguments which are grouped in the "Package selection arguments" section of the usage help.

By default all packages are selected. Each extension can select / unselect matching package decorators based on the passed arguments. If arguments for multiple extensions are being passed the combined boolean flag is a logical `AND` combination.

29.6 TaskExtensionPoint

This extension point is used by verb extensions after packages have been discovered, identified, and augmented. Each task extension implements the logic for the combination of a `verb` and a `package type`, e.g. building a Python package.

In order to be invoked a task extension needs parameters provided by a context object. A job object brings those two together.

29.7 ExecutorExtensionPoint

This extension point is used by some verb extensions to execute a set of jobs. The executor extension with the highest priority is used by default.

29.8 ShellExtensionPoint

This extension point is used by some task extensions to generate shells scripts to setup the environment or create environment hooks for each supported shell. Each extension implements the logic for a specific shell. For the primary shells each extension also provides the environment to run commands.

29.9 EnvironmentExtensionPoint

This extension point is used by some task extensions as well as shell extensions to create environment hooks for each supported shell. Each extension creates environment hooks for a specific environment variable and uses the shell extensions to generate scripts for each supported shell.

29.10 EventHandlerExtensionPoint

This extension point defines how events are handled. An event can be any kind of object, like a line of output, a job starting, progressing or ending, or a command executing. Every event is dispatched to all extensions in order of their priority.

CHAPTER 30

Contributions

There are already many great contribution guidelines available online. Therefore only a few important bullets are enumerated here. Please read for example the Open Source Guide How to contribute for more detailed information which was created and is curated by GitHub.

30.1 Bug reports

- Make sure you are using the latest version.
- Search the project's issue tracker and/or the internet for similar reports.
- Perform basic troubleshooting steps:
  - Try to reproduce the problem "from scratch".
  - If you are deviating from any instructions try to following the instructions and see if the problem persists.
  - If it seems to work for others ask yourself what is different in your case.
- Consider to provide a pull request if possible.
- And as a last step report a bug you can't solve yourself:
  - Describe the expected as well as the actual behavior.
  - Give enough context (e.g. platform, versions, environment).
  - Provide easily reproducible steps and/or a SSCCE.

30.2 Pull requests

- Keep each pull request focused on one aspect, create separate ones for separate aspects.
- For bug fixes make sure to reproduce the problem before and after applying the patch ensure that the problem has been addressed.

- For larger patches consider to create a feature request ticket to discuss the proposal ahead of time.
- Ensure that new code is covered by tests to prevent regressions in the future.
- Make sure that the code changes pass the existing tests including the linters.
- And as a last step create a pull request.

30.3 Documentation

Since documentation is stored in a git repository any changes are made through pull requests and therefore the above bullets for pull requests apply.

The documentation follows the one-sentence-per-line style to minimize the diff (preventing reflow in a paragraph on changes).

30.4 New packages / extensions

Using Python entry points it is easy to contribute extensions in separate packages. To ease discoverability and ensure long term maintenance if individual maintainers move on it is encouraged to host the code in a repository under the *colcon* organization unit on GitHub. Please open a ticket to either ask for the creation of a repository which you will have *admin* level access to or for moving an existing repository to this organization unit.

30.4.1 Use keyword in package metadata

When creating a package containing `colcon` extensions please consider declaring a keyword to help discovering extensions through e.g. PyPI. When using a `setup.cfg` file for the metadata of the package it is as simple as including these lines:

```
[metadata]
keywords = colcon
```

CHAPTER 31

Make Releases

If you are developing a Python package containing colcon extensions you might want to make a release of it. Commonly that involves bumping the version number and tagging the commit with that version.

Commonly Python packages are released to PyPI following the packaging instructions for Python projects. For colcon packages the recommendation is to use publish-python which not only uploads a wheel to PyPI but also creates and uploads Debian packages.

31.1 First-time setup

Before using `publish-python` you need to do the following:

- Install the prerequisites of `publish-python`
- Configure credentials for twine / PyPI
- Configure credentials for packagecloud.io and request to be added as a contributor to the colcon repository
- Create a configuration file for your package

For details please see the publish-python documentation.

Example configuration file `publish-python.yaml`:

```
artifacts:
  - type: wheel
    uploads:
      - type: pypi
  - type: stdeb
    uploads:
      - type: packagecloud
        config:
          repository: dirk-thomas/colcon
          distributions:
            - ubuntu:xenial
```

(continues on next page)

(continued from previous page)

```
            - ubuntu:bionic
            - ubuntu:focal
            - debian:stretch
            - debian:buster
```

The list of targeted distributions might need to be updated to match the configuration files mentioned in the following subsection.

31.2 Publishing a release

After having tagged the repository you only need to invoke `publish-python --upload` to create the configured artifacts as well as upload them to the configured destinations.

In the case the Debian package is used by the ROS community you need to open a ticket in the GitHub repository ros-infrastructure/reprepro-updater. The pull request should update the two files colcon.ubuntu.upstream.yaml and colcon.debian.upstream.yaml file with a bumped version number for the package. Someone from the ROS team will then review and merge the pull request and import the latest artifacts into the apt repositories of ROS.

CHAPTER 32

Changelog

Most colcon repositories are hosted on GitHub and contain a single Python package. The following describes the process used to provide a detailed changelog for each release. The main goal is to keep the necessary effort minimal and reuse the information available in the GitHub tickets.

32.1 Milestone

For each upcoming release a milestone is being created with the tentative version number as the title. Every issue and pull request targeting that upcoming release should have that milestone set. That allows to list all open / closed tickets for a specific milestone using the GitHub interface.

32.2 Release

When making a release and tagging a new version the due date of the matching milestone should be edited to contain the date of the release. Afterwards the milestone should be closed and a new milestone with the next tentative version number should be created.

32.3 Link

To make the changelog information easily accessible the package manifest should provide a link like this in the `setup.cfg`:

```
[metadata]
project_urls =
    Changelog = https://github.com/colcon/<reponame>/milestones?direction=desc&
↪sort=due_date&state=closed
```

CHAPTER 33

ament_tools

The following describes the mapping of some `ament_tools` options and arguments to the `colcon` command line interface.

33.1 ament build | test

[BASEPATH] `--base-paths BASEPATH`

--build-space PATH `--build-base PATH`

--install-space PATH `--install-base PATH`

--build-tests CMake configures tests by default. To skip configuring tests use `--cmake-args -DBUILD_TESTING=OFF`.

-s, --symlink-install `--symlink-install`

--isolated The colcon option `--merge-install` has the inverse logic.

--start-with PKGNAME `--packages-start PKGNAME`

--end-with PKGNAME `--packages-end PKGNAME`

--only-packages PKGNAME1 ... PKGNAMEn `--packages-select PKGNAME1 ... PKGNAMEn`

--skip-packages PKGNAME1 ... PKGNAMEn `--packages-skip PKGNAME1 ... PKGNAMEn`

--parallel colcon uses the parallel execution by default. To build packages sequentially use `--executor sequential`.

33.2 ament build

```
colcon build ...
```

--cmake-args -D... -- `--cmake-args -D...` The closing double dash is not necessary anymore. Any CMake arguments which match colcon arguments need to be prefixed with a space. This can be done by quoting each argument with a leading space.

--force-cmake-configure `--cmake-force-configure`

--make-flags When using this option to pass a target name the substitution is: `--cmake-target TARGET`. When using this option to control the parallel execution with arguments like `-jN` the substitution is to use the environment variable `MAKEFLAGS`.

--use-ninja `--cmake-args -G Ninja`

33.3 ament test

`colcon test ...`

--ctest-args ... -- `--ctest-args ...` Any CTest arguments which start with a dash need to be prefixed with a space (see `--cmake-args`).

--retest-until-fail N `--retest-until-fail N`

--retest-until-pass N `--retest-until-pass N`

--abort-on-test-error `--abort-on-error`

33.4 ament test_results

`colcon test-result ...`

[BASEPATH] `--build-base BASEPATH`

--verbose `--all`

33.5 Behavioral changes

The colcon `test` verb performs only the action of running tests. It does not build any packages.

`--retest-until-fail` with `colcon` uses pytest-repeat which runs individual tests of a package N+1 times each (the first test N+1 times, then the second test N+1 times, etc). With `ament_tools` the entire test suite of a package was run up to N+1 times. As a consequence `colcon` provides a more accurate result since each test that passed has actually run N times. Note that with `pytest-repeat`, `pytest` tests are repeated N times regardless of the result of the previous runs; if a test fails it will be repeated N times anyway. This is different from the behavior of a CTest test that will stop being repeated as soon as it fails once.

The location of JUnit test results file for `ament_python` packages tested with `colcon` is in `<pkg-build>/pytest.xml`, whereas with `ament_tools` it is in `<pkg-build>/test_results/<pkgname>/pytest.xunit.xml`.

CHAPTER 34

catkin_make_isolated

The following describes the mapping of some `catkin_make_isolated` options and arguments to the `colcon` command line interface.

`--source PATH` `--base-paths BASEPATH`

`--build PATH` `--build-base PATH`

`--devel PATH` colcon doesn't support the concept of a "devel" space. Instead you can choose the path of the devel space as the install base and perform an normal installation.

`--install-space PATH` `--install-base PATH`

`--merge` `--merge-install`

`--use-ninja` `--cmake-args -G Ninja`

`--use-nmake` `--cmake-args -G "NMake Makefiles"`

`--install` colcon always performs an installation. It doesn't support the concept of a "devel" space.

`--cmake-args ...` `--cmake-args ...` The closing double dash is not necessary anymore. Any CMake arguments which match colcon arguments need to be prefixed with a space. This can be done by quoting each argument with a leading space.

`--force-cmake` `--cmake-force-configure`

`--pkg PKGNAME1 ... PKGNAMEn` `--packages-select PKGNAME1 ... PKGNAMEn`

`--from-pkg PKGNAME` `--packages-start PKGNAME`

`--only-pkg-with-deps PKGNAME1 ... PKGNAMEn` `--packages-up-to PKGNAME1 ... PKGNAMEn`

CHAPTER 35

catkin_tools

The following describes the mapping of some `catkin_tools` options and arguments to the `colcon` command line interface.

35.1 catkin build

`[PKGNAME1 ... PKGNAMEn]` `--packages-up-to PKGNAME1 ... PKGNAMEn`

`--no-deps` `--packages-select PKGNAME1 ... PKGNAMEn`

`--start-with PKGNAME` `--packages-start PKGNAME`

`--force-cmake` `--cmake-force-configure`

`--cmake-args ... --` `--cmake-args ...` The closing double dash is not necessary anymore. Any CMake arguments which match colcon arguments need to be prefixed with a space. This can be done by quoting each argument with a leading space.

`-v, --verbose` `--event-handlers console_cohesion+`

`-i, --interleave-output` `--event-handlers console_direct+`

`--no-status` `--event-handlers status-`

`--no-summarize, --no-summary` `--event-handlers summary-`

`--no-notify` `--event-handlers desktop_notification-`