Exec-helper

Release 0.5.0

Sep 18, 2022

Contents:

CHAPTER 1

Exec-helper

Or How To Get Your Coffee In Peace.

1.1 What

Exec-helper is a meta-wrapper for executing tasks on the command line.

1.2 Why

Exec-helper improves the main bottleneck in your development workflow: you.

It does this by:

- Reducing the number of keystrokes required to execute the same command over and over again
- Chaining multiple commands

All without sacrificing (much) flexibility or repeating useless work.

If this, together with *getting coffee in peace* is not a sufficient rationale for you, the main advantages of exec-helper over (simple) scripts or plain command line commands are:

- Easy permutation of multiple execution parameters (so-called *patterns* in exec-helper).
- Easy selection of a subset of execution parameters.
- Improved DRY: execution parameters are only changed on one spot, in stead of everywhere in your command line.
- Technology-agnostic approach: e.g. running the *exec-helper build* can build a C++ project in one directory structure and a JAVA project in another.
- Enables a self-documented workflow.

- Out of the box support for multi-valued options and default values.
- Searches for a suitable configuration in its parent folders.
- Fast to type using the *eh* alias
- Easy to find and/or list available commands using the *–help* option.
- Easy extensible with your own, first-class citizen, plugins.
- Automatic autocompletion of commands and patterns

1.3 Simple example

This is a simple illustration of the concept behind exec-helper. More extensive information and examples can be found in the *.exec-helper* configuration file for this repository and in the documentation.

1.3.1 Use case

Build a C++ project using g++ and clang++ using cmake in a *Debug* and *RelWithDebInfo* configuration

1.3.2 Configuration file

Copy the following to a file named '.exec-helper':

```
commands:
    init: Initialize build
    build: Build-only + install
    build-only: Build
    install: Install

patterns:
    COMPILER:
        default-values:
            - g++
            - clang++
        short-option: c
        long-option: compiler

    MODE:
        default-values:
            - debug
            - release
        short-option: m
        long-option: mode

build:
    - build-only
    - install

init:
    - command-line-command

build-only:
    - make
```

(continues on next page)

(continued from previous page)

```
install:
    - make

command-line-command:
    init:
        patterns:
            - COMPILER
            - MODE
        command-line: [ cmake, -H., "-Bbuild/{COMPILER}/{MODE}", "-DCMAKE_CXX_
↪COMPILER={COMPILER}", "-DCMAKE_INSTALL_PREFIX=install/{COMPILER}/{MODE}", "-DCMAKE_
↪BUILD_TYPE={MODE}"]

make:
    patterns:
        - COMPILER
        - MODE
    build-dir: "build/{COMPILER}/{MODE}"
    install:
        command-line: install
```

1.3.3 Example output

```
$ exec-helper --help
   -h [ --help ]              Produce help message
   --version                  Print the version of this binary
   -v [ --verbose ]           Set verbosity
   -j [ --jobs ] arg          Set number of jobs to use. Default: auto
   -n [ --dry-run ]           Dry run exec-helper
   -s [ --settings-file ] arg Set the settings file
   -d [ --debug ] arg         Set the log level
   -z [ --command ] arg       Commands to execute
   -c [ --compiler ] arg      Values for pattern 'compiler'
   -m [ --mode ] arg          Values for pattern 'mode'

 Configured commands:
   init                 Initialize build
   build                Build-only + install
   build-only           Build
   install              Install

$ exec-helper init build    # Permutate all combinations of the default values
Executing "cmake -H. -Bbuild/g++/debug -DCMAKE_CXX_COMPILER=g++ -DCMAKE_INSTALL_
↪PREFIX=install/g++/debug -DCMAKE_BUILD_TYPE=debug"
Executing "cmake -H. -Bbuild/g++/release -DCMAKE_CXX_COMPILER=g++ -DCMAKE_INSTALL_
↪PREFIX=install/g++/release -DCMAKE_BUILD_TYPE=release"
Executing "cmake -H. -Bbuild/clang++/debug -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_
↪INSTALL_PREFIX=install/clang++/debug -DCMAKE_BUILD_TYPE=debug"
Executing "cmake -H. -Bbuild/clang++/release -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_
↪INSTALL_PREFIX=install/clang++/release -DCMAKE_BUILD_TYPE=release"
Executing "make --directory build/g++/debug --jobs 8"
Executing "make --directory build/g++/release --jobs 8"
Executing "make --directory build/clang++/debug --jobs 8"
Executing "make --directory build/clang++/release --jobs 8"
Executing "make --directory build/g++/debug --jobs 8 install"
```

(continues on next page)

(continued from previous page)

```
Executing "make --directory build/g++/release --jobs 8 install"
Executing "make --directory build/clang++/debug --jobs 8 install"
Executing "make --directory build/clang++/release --jobs 8 install"

$ exec-helper build-only --compiler g++ --mode release      # Only build the g++␣
↪build in release mode
Executing make --directory build/g++/release --jobs 8

$ exec-helper install --compiler g++ --mode debug RelWithDebInfo    # Install a␣
↪subset - even using ones not listed in the default values
Executing make --directory build/g++/debug --jobs 8 install
Executing make --directory build/g++/RelWithDebInfo --jobs 8 install
```

1.4 Installation

See INSTALL for more information on:

- Using one of the available packages or installers
- (Cross-)build from source

1.5 Documentation

See documentation for the latest documentation.

1.5.1 Usage

see exec-helper for usage information.

1.5.2 Configuration

See exec-helper-config for information on the configuration file format.

1.5.3 Available plugins

See exec-helper-plugins for a list of all available plugins.

1.5.4 Writing custom plugins

See exec-helper-custom-plugins for a guide on writing your own plugins.

1.6 Code quality

The source code of this project is continuously analyzed by multiple tools in an attempt to catch and fix issues and bugs as quickly as possible. Released versions should have passed the analysis from the following tools:

- AddressSanitizer (ASan)
- clang-format
- clang-static-analyzer
- clang-tidy
- cppcheck
- License Scanning (by Gitlab)
- pmd (cpd)
- Static Application Security Testing (SAST by Gitlab)
- Valgrind (memcheck)
- UndefinedBehaviorSanitizer (UBSan)

Check the *.exec-helper* file for detailed information about how these analysis methods are configured and used. The analysis tools can be executed locally using exec-helper with this project.

CHAPTER 2

Installation instructions

2.1 Installing from package

2.1.1 Arch Linux based distributions

Arch linux users can:

*. **Use the pre-built Arch Linux pre-built binary package:** Add to */etc/pacman.conf*:

```
[home_bverhagen_exec-helper_Arch]
SigLevel = Optional TrustAll
Server = https://download.opensuse.org/repositories/home:/bverhagen:/exec-helper/
↪Arch/x86_64/
```

Then:

```
curl -L -O https://download.opensuse.org/repositories/home:/bverhagen:/exec-
↪helper/Arch/x86_64/home_bverhagen_exec-helper_Arch.key
sudo pacman-key --add home_bverhagen_exec-helper_Arch.key
sudo pacman-key --lsign-key C6DA27F1EB5EE305
```

*. Use the exec-helper (AUR) package *. Check out the exec-helper-package project for building the package from source. See the *Building from source* section.

2.1.2 Ubuntu

note: The support of non-LTS versions is rather limited. You are welcome to contribute if one is missing!

Ubuntu users (Bionic and later) can:

*. Add the PPA on Launchpad to your sources *. Check out the exec-helper-package project for building the package from source. See the *Building from source* section.

2.1.3 openSUSE

note: Tumbleweed an Leap 15.4 and later are supported.

openSUSE users can:

*. Check out the binaries from the home:bverhagen:exec-helper project on OBS *. Check out the exec-helper-package project for building the package from source. See the *Building from source* section.

2.1.4 Other distributions

Checkout the *Building from source* section.

2.2 Building from source

2.2.1 Requirements

Build tools

- A C++ 17 compatible compiler. Tested with: *gcc*, *clang* and MSVC 2017 (14.1)
- meson
- ninja
- make for the quick install
- Sphinx for generating man-pages and general documentation
- Doxygen (1.8.15 or newer) for building API documentation (optional)
- gitchangelog for building the changelog (optional)

Build dependencies

- POSIX compliant operating system
- boost-program-options (1.64 or newer) development files
- boost-log (1.64 or newer) development files
- yaml-cpp (0.5.3 or newer) development files (optional, will be downloaded and compiled in statically if missing)
- lua (5.3 or newer) development files (optional, will be downloaded and compiled in statically if missing)
- readline development files (*NIX systems): required if not using the system Lua.

2.2.2 Quick installation

```
$ make
$ sudo make install
```

Use

```
$ make help
```

for an overview of the available quick installation targets and for an overview of the (very limited) available configuration options.

2.2.3 Advanced installation

CMake is the build system. The *Makefile* mentioned in the quick installation is a simple wrapper around a more complex - and more configurable - CMake invocation.

It has the following project-specific configuration options:

USE_SYSTEM_YAML_CPP
: Use the yaml-cpp library installed on the system. If switched off, the library will be automatically installed locally during the build. Default: *auto*

USE_SYSTEM_LUAJIT
: Use the luaJIT library installed on the system. If switched off, the library will be automatically installed locally during the build. Default: *auto*

POSITION_INDEPENDENT_CODE
: Build using position independent code. Default: *ON*

ENABLE_TESTING
: Enable building the testing infrastructure. Default: *ON*

BUILD_MAN_DOCUMENTATION
: Generate the man-pages for this project

BUILD_HTML_DOCUMENTATION
: Generate the HTML documentation for this project

BUILD_XML_DOCUMENTATION
: Generate the XML documentation for this project

2.2.4 Build tests

Testing is enabled by setting the CMake configuration option *ENABLE_TESTING* to *ON*.

The tests require, in addition to all dependencies above, the following dependencies:

- Catch2 unittest framework development files (optional, for building the tests)
- Rapidcheck property based unittest framework development files (optional, for building the tests)

Testing related configuration options:

ENABLE_WERROR
: Enable warning as error during compilation (only supported for *GCC* and *clang*)

LIMITED_OPTIMIZATION
: Build with limited optimization (typically -O1, only supported for *GCC* and *clang*). This is typically used for running tools like valgrind.

TERMINATE_ON_ASSERT_FAILURE
: Explicitly terminate when an assert fires.

USE_SYSTEM_CATCH
: Use the Catch2 library installed on the system. If switched off, the library will be automatically installed locally during the build. Default: *auto*

USE_SYSTEM_RAPIDCHECK
: Use the Rapidcheck library installed on the system. If switched off, the library will be automatically installed locally during the build. Default: *auto*

2.3 Cross compilation

Exec-helper supports both native and cross compilation (including building with a custom sysroot) builds. Cross compilation requires invoking cmake directly and appending **-DCMAKE_TOOLCHAIN_FILE=<toolchain-file>** to the cmake initialization command. Check the *toolchain.cmake.in* file for a template on setting up the toolchain file for cross compilation and the *Makefile* for a template of the cmake initialization command.

CHAPTER 3

exec-helper

3.1 Synopsis

exec-helper <*commands*> [*options*]

eh <*commands*> [*options*]

3.2 Description

The **`exec-helper`** utility is a meta-wrapper for executables, optimizing one of the slowest links in most workflows: you. It enables the user to optimize the existing workflow in multiple minor and major ways:

- It minimizes the amount of typing while eliminating redundancies
- It chains multiple commands, inserting patterns at specified places
- It avoids having to memorize or search for the right invocations for more complicated commands
- It allows to write your system- and project-specific plugins for more advanced optimizations

These optimizations enable efficient users to do what they like to do the most: hang around the coffee machine with peace of mind.

3.3 Options

Mandatory arguments to long options are mandatory for short options too. Arguments to options can be specified by appending the option with '=ARG' or ' ARG'. This manual will further use the '=ARG' notation. Multiple arguments can be specified, if appropriate and without the need to repeat the option, by using spaces in between the arguments.

`-h, --help`
: Display a usage message on standard output and exit successfully.

-v, --verbose
Enable the verbose flag for the command if available.

-z, --command=COMMAND
Execute one or more configured COMMANDs. This is an alias for the <*commands*> mandatory option above.

-s, --settings-file[=FILE]
Use FILE as the settings file for the **exec-helper** configuration. Default: *.exec-helper*. Exec-helper will use the first file it finds with the given FILE name. It will search in order in the following locations:

1. The current working directory
2. The parent directories of the working directory. The parent directories are searched in *reversed* order, meaning that the direct parent of the current working directory is searched first, next the direct parent of the direct parent of the current working directory and so-forth until the root directory is reached.
3. The *HOME* directory of the caller.

-j, --jobs[=JOBS]
Use the specified number of JOBS for each task (if supported). Use *auto* to let **exec-helper** determine an appropriate number. Use a value of *1* for running jobs single-threaded. Default: *auto*.

-n, --dry-run
Print the commands that would be executed, but do not execute them.

-k, --keep-going
Execute all scheduled commands, even if one or more of them fail.

3.4 Configured options

Additional command-line options for **exec-helper** can be configured in the settings file. Refer to the *exec-helper-config(5)* documentation for more information.

3.5 Exit status

When **exec-helper** is called improperly or its plugins are invoked improperly, **exec-helper** will exit with a status of *one*. In other cases, it exits with the same status as the last failed command or *zero* if all commands are executed successfully.

3.6 Auto-completion

Auto-completions are available for the Bash and Zsh shell. Package maintainers receive the tools to automatically enable these completions. If your installation package does not do this, you can enable them yourself by adding *source <install-directory>/share/exec-helper/completions/init-completion.sh* to your profile or bashrc.

3.7 See also

See *Configuration* (5) for information about the configuration file.

See *Plugins* (5) for the available plugins and their configuration options.

CHAPTER 4

Configuration

4.1 Environment

4.1.1 Description

Environment variables can be configured in the configuration file. They will only be set for the particular command(s) defined by the relevant section of the configuration.

Environment variables can *not* be set directly in a command line command. The **`environment`** configuration key needs to be used for this. See section 'environment'.

4.1.2 Environment

The **`environment`** keyword can be set for every plugin that supports the env configuration setting. Check the documentation on a specific plugin to check whether the plugin supports this configuration setting.

The **`environment`** keyword must contain a *map* of key-value pairs, where the key is the name of the **`environment`** variable and the value is the value associated with the specified **`environment`** variable. *Patterns* can be used for the **`environment`** these variable values too.

Note: The *PWD* environment variable, following POSIX convention, is set by the application to the working directory of the task. Therefore, its value cannot be overriden in the configuration.

4.1.3 Example configuration

```
commands:                       # The mandatory commands key
    build: Build the project    # A map of command keys with their explanation
    clean: Clean the project
    rebuild: Build + clean

patterns:                       # Declare the patterns for this configuration file
```

(continues on next page)

(continued from previous page)

```
    COMPILER:                   # Declare the COMPILER pattern
        default-values:         # Default values to use for the pattern
            - g++
            - clang++
        short-option: c         # Declare values for this pattern by using the -c␣
↪[VALUES] option when calling exec-helper
        long-option: compiler   # Declare values for this pattern by using the --
↪compiler [VALUES] option when calling exec-helper
    MODE:                       # Declare the MODE pattern
        default-values:
            - debug
            - release
        short-option: m
        long-option: mode

additional-search-paths:
    - /tmp

# Define the commands listed under 'commands'
build:
    - command-line-command      # Use the command-line-command plugin when using the
↪'build' command

clean:
    - command-line-command      # Use the command-line-command plugin when using the
↪'clean' command

rebuild:
    - clean                     # Call the 'clean' command when calling the 'rebuild'␣
↪command
    - build                     # Call the 'build' command when calling the 'rebuild'␣
↪command

command-line-command:           # Configure the command-line-command
    patterns:                   # Define the default patterns to use
        - COMPILER
        - MODE

    command-line:               # Configure the execution when the specific command␣
↪is not listed. Will be executed from the directory of this configuration file
        - echo
        - building
        - using
        - "{COMPILER}"          # This value will be replaced by the COMPILER pattern␣
↪value
        - in
        - "{MODE}"              # This value will be replaced by the MODE pattern␣
↪value
        - mode.
        - wd=$(pwd)             # This command will be executed in a subshell and␣
↪replaced by its value before the actual command is executed

    clean:                      # Configure the execution of the build command
        patterns:               # Overwrite the parent patterns
            - MODE
            - EH_WORKING_DIR    # Use the EH_WORKING_DIR pattern
        command-line:
```

(continues on next page)

(continued from previous page)

```
            - echo
            - cleaning
            - mode.
            - "{MODE}"          # This value will be replaced by the MODE pattern␣
↪value
            - wd=$(pwd)
        working-dir: "{EH_WORKING_DIR}" # The command will be executed from the␣
↪current working directory rather than from the directory of this configuration file
```

4.1.4 See also

See *Configuration* (5) for information about the configuration file.

4.2 Patterns

4.2.1 Description

Patterns are parts of the configuration that will be replaced by its value when evaluated by **exec-helper**. The *patterns* keyword describes a list of patterns identified by their key. See the 'patterns' section for more information about how to define a pattern.

Patterns can be used to:

- add options to the **exec-helper** command line
- centralize a value in a variable
- allow iterating over multiple configurations
- control the configurations to iterate over

4.2.2 Patterns

A pattern can contain the following fields:

default-values
: A list of default values to use when no values have been defined.

short-option
: The short option on the command line associated with this pattern

long-option
: The long option on the command line associated with this pattern

4.2.3 Predefined patterns

Exec-helper predefines some specific patterns for convenience:

- **EH_ROOT_DIR**: contains the absolute path to the directory where the **exec-helper** configuration is located. Useful for converting relative paths to absolute paths for tools that require it (e.g. when setting your `PATH`)
- **EH_WORKING_DIR**: contains the working directory from where the **exec-helper** executable is called.

4.2.4 Example configuration

```yaml
commands:                           # The mandatory commands key
    build: Build the project        # A map of command keys with their explanation
    clean: Clean the project
    rebuild: Build + clean

patterns:                           # Declare the patterns for this configuration file
    COMPILER:                       # Declare the COMPILER pattern
        default-values:             # Default values to use for the pattern
            - g++
            - clang++
        short-option: c             # Declare values for this pattern by using the -c␣
↪[VALUES] option when calling exec-helper
        long-option: compiler       # Declare values for this pattern by using the --
↪compiler [VALUES] option when calling exec-helper
    MODE:                           # Declare the MODE pattern
        default-values:
            - debug
            - release
        short-option: m
        long-option: mode

additional-search-paths:
    - /tmp

# Define the commands listed under 'commands'
build:
    - command-line-command          # Use the command-line-command plugin when using the
↪'build' command

clean:
    - command-line-command          # Use the command-line-command plugin when using the
↪'clean' command

rebuild:
    - clean                         # Call the 'clean' command when calling the 'rebuild'␣
↪command
    - build                         # Call the 'build' command when calling the 'rebuild'␣
↪command

command-line-command:               # Configure the command-line-command
    patterns:                       # Define the default patterns to use
        - COMPILER
        - MODE

    command-line:                   # Configure the execution when the specific command␣
↪is not listed. Will be executed from the directory of this configuration file
        - echo
        - building
        - using
        - "{COMPILER}"              # This value will be replaced by the COMPILER pattern␣
↪value
        - in
        - "{MODE}"                  # This value will be replaced by the MODE pattern␣
↪value
        - mode.
```

(continues on next page)

(continued from previous page)

```
        - wd=$(pwd)              # This command will be executed in a subshell and␣
↪replaced by its value before the actual command is executed

    clean:                       # Configure the execution of the build command
        patterns:                # Overwrite the parent patterns
            - MODE
            - EH_WORKING_DIR     # Use the EH_WORKING_DIR pattern
        command-line:
            - echo
            - cleaning
            - mode.
            - "{MODE}"           # This value will be replaced by the MODE pattern␣
↪value
            - wd=$(pwd)
        working-dir: "{EH_WORKING_DIR}" # The command will be executed from the␣
↪current working directory rather than from the directory of this configuration file
```

4.2.5 See also

See *Configuration* (5) for information about the configuration file.

4.3 Description

Exec-helper configuration files are written in the YAML 1.2 specification.

4.4 Mandatory keys

A valid configuration file must contain at least the following keys on the root level of the configuration file:

commands

The commands that are configured in the configuration file. It will either contain a list of commands or a list of the commands as keys with an explanation of the command as a value. These formats can not be used interchangeably.

<command-keys>

For every command defined under the *commands* key, the configuration must define this command as a key in the root of the configuration file. The value of the key must either be a registered plugin or another command.

<plugin-keys>

For at least every plugin that is used by a *command* key, configure the specifics of the plugin (if applicable).

4.5 Optional keys

Optionally the configuration file contains the following keys on the root level of the configuration file:

patterns
: Patterns are parts of the configuration that will be replaced by its value when evaluated by exec-helper. The *patterns* keyword describes a list of patterns identified by their key. See the @ref exec-helper-config-patterns for more information about how to define a pattern.

additional-search-paths
: An ordered list of additional search paths to use when searching for plugins. The search paths can be absolute or relative w.r.t. the parent path of the *settings file* in which these paths are defined.

  Defining search paths is useful for extending exec-helper with your own custom plugins or for overwriting or extending the functionality in the provided plugins. See [exec-helper-custom-plugins](@ref exec-helper-custom-plugins)(5) for more information on writing a custom plugin.

  The paths defined in this list take precedence over the system search paths for modules with the same name. A higher position in this list implicates higher precedence.

4.6 Working directory

Configured commands are executed from the so-called *working directory*. Executing commands in a different working directory will not affect your current working directory (e.g. when executing from a shell). Each separately configured command can be executed in a separate working directory.

The *working directory* is the directory that is associated with the first of the following lines whose requirement is met: 1. The *working-dir* configuration setting is configured for the specific command. The value of the *working-dir* configuration key can be an absolute path to the working directory or a relative one w.r.t. the directory of the considered configuration file. If the command should be executed in the actual working directory, use <*working-dir*> as the value in the configuration file. 2. The directory of the considered configuration file.

4.7 Paths

All relative paths in the configuration should be *relative to the directory in which the configuration resides*. While relative paths are convenient for users as they can freely choose the root directory of an application, some applications require an absolute path. In such case, use the *${PWD}* environment variable (both POSIX and non-POSIX systems) to convert a relative path in your configuration into an absolute path for calling these particular applications.

4.8 Example configuration

```
commands:                          # The mandatory commands key
    build: Build the project       # A map of command keys with their explanation
    clean: Clean the project
    rebuild: Build + clean

patterns:                          # Declare the patterns for this configuration file
    COMPILER:                      # Declare the COMPILER pattern
        default-values:            # Default values to use for the pattern
            - g++
            - clang++
        short-option: c            # Declare values for this pattern by using the -c␣
↪[VALUES] option when calling exec-helper
        long-option: compiler      # Declare values for this pattern by using the --
↪compiler [VALUES] option when calling exec-helper
    MODE:                          # Declare the MODE pattern
        default-values:
            - debug
            - release
        short-option: m
```

(continues on next page)

(continued from previous page)

```
        long-option: mode

additional-search-paths:
    - /tmp

# Define the commands listed under 'commands'
build:
    - command-line-command      # Use the command-line-command plugin when using the
↪'build' command

clean:
    - command-line-command      # Use the command-line-command plugin when using the
↪'clean' command

rebuild:
    - clean                     # Call the 'clean' command when calling the 'rebuild'␣
↪command
    - build                     # Call the 'build' command when calling the 'rebuild'␣
↪command

command-line-command:           # Configure the command-line-command
    patterns:                   # Define the default patterns to use
        - COMPILER
        - MODE

    command-line:               # Configure the execution when the specific command␣
↪is not listed. Will be executed from the directory of this configuration file
        - echo
        - building
        - using
        - "{COMPILER}"          # This value will be replaced by the COMPILER pattern␣
↪value
        - in
        - "{MODE}"              # This value will be replaced by the MODE pattern␣
↪value
        - mode.
        - wd=$(pwd)             # This command will be executed in a subshell and␣
↪replaced by its value before the actual command is executed

    clean:                      # Configure the execution of the build command
        patterns:               # Overwrite the parent patterns
            - MODE
            - EH_WORKING_DIR    # Use the EH_WORKING_DIR pattern
        command-line:
            - echo
            - cleaning
            - mode.
            - "{MODE}"          # This value will be replaced by the MODE pattern␣
↪value
            - wd=$(pwd)
        working-dir: "{EH_WORKING_DIR}" # The command will be executed from the␣
↪current working directory rather than from the directory of this configuration file
```

4.9 See also

See *Patterns* (5) for more information on defining and using patterns.

See *Environment* (5) for more information on configuring execution environments.

See *exec-helper* (1) for information about the usage of exec-helper.

See *Plugins* (5) for the available plugins and their configuration options.

See *Custom plugins* (5) for the available plugins and their configuration options.

CHAPTER 5

Plugins

5.1 Custom plugins

5.1.1 Where to put your plugins

Exec-helper searches dynamically for (most of) its plugins in all the plugin search paths. It searches in the following locations (earlier listed locations take precedence over later listed locations for plugins with the same name):

1. Using the `--additional-search-path` command-line option. Multiple paths can be passed to it using multiple arguments. Earlier mentioned paths take precedence over later mentioned paths. The paths can be absolute or relative w.r.t. the used **`exec-helper`** configuration file. E.g.:

```
exec-helper build --additional-search-path blaat /tmp
```

will add the relative path *blaat* and the absolute path */tmp* to the plugin search paths.

2. Using the `additional-search-paths` key in the **`exec-helper`** configuration file. The key takes an ordered list containing absolute or relative (w.r.t. the **`exec-helper`** configuration file it is mentioned in) paths. Earlier listed elements take precedence over lower listed elements. E.g.:

```
additional-search-paths:
    - blaat
    - /tmp
```

3. The system plugin paths. These paths contain (most of) the default modules bundled with **`exec-helper`**. It is not recommended to add your custom plugins to any of these paths.

5.1.2 Listing the modules

Exec-helper lists the modules it currently finds by using the `--list-plugins` command-line option.

5.1.3 Writing a lua plugin

Exec-helper supports luaJIT 2.0.5. LuaJIT is a Lua 5.1 implementation with some additional features from Lua 5.2. All LuaJIT functionality is embedded in the **exec-helper** binary, no LuaJIT install is required for running the plugin.

Exec-helper treats all files in the plugin search paths with a *lua* suffix as a compatible lua plugin. The name of the module is derived from the rest of the filename.

5.1.4 The interface

A lua plugin is called within a wider (lua) context containing some objects and (convenience) functions.

Exec-helper specific functions

The following **exec-helper** specific functions are available next to the lua 5.1 functions:

get_commandline()
Returns a list of the command-line arguments set by the `command-line` key in the configuration. Use this to allow users of your plugin to freely set additional, plugin-specific command-line settings that can not be set by other configuration options. These additional command-line settings must be added explicitly by this plugin in the right position. E.g:

```
task:add_args(get_commandline())
```

get_environment()
Returns a two-level Lua table containing the environment in which the task will be executed. The plugin can read and modify this environment. Values set by the *environment* key in the configuration are added automatically to this list before this plugin is called, there is no need to do this explicitly.

Note: The *PWD* environment variable, following POSIX convention, is set by the application to the working directory of the task. Therefore, its value cannot be overriden in a custom module.

get_verbose(string arg)
Add `arg` to the current tasks' command line if verbose mode is activated. This function does nothing if verbose mode is not activated. E.g.:

```
task:add_args(get_verbose('--debug'))
```

register_task(Task task)
Registers the given `task` as a task to execute by the executor(s). Patterns associated with the task will be automatically permutated and substituted. E.g.:

```
register_task(task)
```

register_tasks(array<Task> tasks)
Registers the given `tasks` as multiple tasks to execute by the executor(s). Patterns associated with the task will be automatically permutated and substituted. E.g.:

```
register_tasks(tasks)
```

run_target(Task task, array<string> targets)
Applies the given targets using the given task as their base task. These targets may contain patterns. The result of these applications is returned as an `array<Task>`. The returned tasks must be explicitly registered in order to be executed. E.g.:

```
run_target(task, {'cmake', 'ninja'})
```

user_feedback_error(string message)
Show the given `message` as an error to the user. E.g.:

```
user_feedback_error('You should not do that!')
```

input_error(string message)
Show the given `message` as an error to the user and stop execution of this module. E.g.:

```
input_error('Cowardly refusing to perform that action!')
```

Exec-helper specific types

The following types (classes) are available in your module:

Config
Behaves like an ordinary lua table. Only reading from it using the access operator (*[key]*) is allowed. The access operator takes a string and returns a Lua table.

Task
Contains the task that is being built. It has the following member functions:

- `add_args(array<string> args)`: Append the given arguments to this task.
- `new(Task task)`: Create a new, default task with an empty command line.
- `copy(Task task)`: Returns a copy of the given task.

Pre-defined objects

The following pre-defined objects are automatically present when your module is called:

verbose
A boolean indicating whether the verbose command-line flag was set for this invocation.

jobs
Integer indicating the number of jobs to use for executing this plugin, if the plugin supports parallel job execution. Ignore this if this is not the case.

Example:

```
task:add_args({'--jobs', jobs})
```

Adds `--jobs \<value\>` to the command line of the given task where <*value*> is the value of the configured number of jobs.

config
A pure Lua table containing the configuration of the particular **exec-helper** configuration into one easy-to-navigate syntax tree. The tree may contain multiple levels. Accessing a table value in Lua returns a new Lua table. Use the `one()` and `list()` function to convert the table to a single value or list respectively. These functions will return *nil* when the given key has no value. The functions distinguish between no value (*nil*) and an empty value (e.g. an empty list).

Example:

```
task:add_args({'--directory', one(config['build-dir']) or '.'})
```

Adds `--directory \<value\>` to the task command line, where <*value*> is one value set by the *build-dir* key or . when no such key exists in the configuration of this plugin.

task
A Task object containing the current context for executing the task, this may include prefixes from other plugins. It is *not* possible to erase these prefixes. If your module requires pre- or post-tasks, you can create one or more new tasks and register it. Similarly, it is possible to create new tasks with the same context as the given *task* variable by copy constructing it. Use the Lua *:* operator for calling member functions of a task.

For example, to create a module that calls *echo hello* on its invocation, use:

```
task:add_args({'echo', 'hello'})
```

5.1.5 Example

A module for a directly callable tool

Let's implement a simple module for calling *make* called `make`:

make.lua:

```
task:add_args({'make'})

task:add_args({'--directory', one(config['build-dir']) or '.'})
task:add_args(get_verbose('--debug'))
task:add_args({'--jobs', one(config['jobs']) or jobs})
task:add_args(get_commandline())

register_task(task)
```

This module adds *make* with some additional arguments from the config and the options to the existing `task` task. At the end, it registers the task for execution.

The relevant section in the users' **exec-helper** configuration may look like:

```
commands:
    build: Build the project

patterns:
    MODE:
        default-values:
            - debug
            - release
        short-option: m
        long-option: mode

build:
    - make

make:
    patterns:
        - MODE

    build:
        build-dir: "build/{MODE}"
        jobs: 3
        command-line: [ --dry-run, --keep-going]
```

Running `eh build --mode release --verbose` will execute the command-line:

```
make --directory build/release --debug --jobs 3 --dry-run --keep-going
```

A module calling an other command

Let's implement a simple module for *clang-static-analyzer*. Per the docs, this analyzer is used by prepending `scan-build <options> <build command>` to the build command line. Obviously, users will already have configured a command (e.g. *build*) for building the project without any analysis. For maintenance and convenience purposes, we do not want the user to replicate this build command for this plugin, but rather, we want our plugin to add some arguments to the tasks' command line and call the configured build-command for extending the task with the actual build configuration.

Let's implement this module, called under the name `some-analyzer`:

some-analyzer.lua:

```
task:add_args({'scan-build'})
task:add_args(get_verbose('-v'))
task:add_args(get_commandline())

local build_commands = list(config['build-command'])

if type(build_commands) == 'nil' then
    input_error('Clang-static-analyzer: one must define at least one build command')
end

if type(next(build_commands)) == 'nil' then
    user_feedback_error('Clang-static-analyzer: one must define at least one build␣
↪command')
    input_error('Clang-static-analyzer: one must define at least one build command')
end

register_tasks(run_target(task, build_commands))
```

This module adds `scan-build` and some additional arguments to the command line of the task. Next, it takes the `build-command` configuration values, does some validity checks on it, and requests **`exec-helper`** to extend the command with the arguments of the given command values.

The relevant section in the users' **`exec-helper`** configuration (combined with the module above for implementing the build command) may look like:

```
build:
    - make

make:
    build-dir: build

some-analyzer:
    build-command: build
    command-line:
        - --keep-going
```

Running `eh some-analyzer --jobs 4` would execute the command line:

```
scan-build --keep-going make --directory build --jobs 4
```

5.2 Bash plugin

5.2.1 Description

The bash plugin is used for executing commands in the *bash* shell, rather than executing the command right away. This is very useful for executing command lines that need special shell characters like `&&`, `|`, `;`, `>`.

5.2.2 Mandatory settings

Mandatory settings for all modes

command
: Command to execute in the shell, as a string. See the `-c` option of `bash` for more information.

5.2.3 Optional settings

The configuration of the bash plugin may contain the following additional settings:

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

5.2.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: run the bash example

patterns:                       # Define the patterns that can be used
    EXAMPLE_PATTERN:            # Define the EXAMPLE_PATTERN.
        default-values:         # Define the default value
            - world!

example:
    - bash                        # Use the bash plugin when running the 'example'␣
↪command
```

(continues on next page)

(continued from previous page)

```
bash:                               # Sh plugin configuration settings
    example:                       # Settings specific to the 'example' command
        environment:               # Define the environment
            EXAMPLE_ENVIRONMENT: hello
        patterns:                  # Define the patterns that are used
            - EXAMPLE_PATTERN
        command: 'echo ${EXAMPLE_ENVIRONMENT} && echo {EXAMPLE_PATTERN} && echo
↪"working directory is $(pwd)"'      # Define the shell command
        command-line: [ -ex]      # Pass additional command line arguments
        working-dir: /tmp         # Set the working directory to a predefined value
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.2.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.3 Bootstrap plugin

5.3.1 Description

The bootstrap is used for executing bootstrap files. This is often used in build chains.

5.3.2 Mandatory settings

There are no mandatory settings for the bootstrap plugin.

5.3.3 Optional settings

The configuration of the bootstrap plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

filename
: The name of the bootstrap script. Default: `bootstrap.sh`.

5.3.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: run the bootstrap example

patterns:                       # Define the patterns that can be used
    EXAMPLE_PATTERN:            # Define the EXAMPLE_PATTERN.
        default-values:         # Only define the default value
            - world!

example:
    - bootstrap                 # Use the command-line-command plugin when running␣
↪the 'example' command

bootstrap:                      # Bootstrap configuration settings
    example:                    # Settings specific to the 'example' command
        patterns:               # Define the patterns that are used
            - EXAMPLE_PATTERN
        filename: src/bootstrap-mock.sh     # Set the name of the bootstrap script
        command-line:           # Define 2 additional command line flags
            - "hello"
            - "{EXAMPLE_PATTERN}"
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.3.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.4 Clang-static-analyzer plugin

5.4.1 Description

The clang-static-analyzer plugin is used for executing the clang-static-analyzer static code analysis tool.

5.4.2 Mandatory settings

The configuration of the clang-static-analyzer plugin must contain the follwing settings:

build-command
: The **exec-helper** build target command or plugin to execute for the analysis.

5.4.3 Optional settings

The configuration of the clang-static-analyzer plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

5.4.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the make example
    build: Build the files
    clean: Clean the build

patterns:                       # Define the patterns that can be used
    MAKE_TARGET:                # Define make targets for building
        default-values:         # Only define the default value
            - hello
            - world

example:
    - clean
    - clang-static-analyzer     # Use the clang-static-analyzer plugin when running␣
↪the 'example' command

build:
    - make

clean:
    - make

clang-static-analyzer:          # Configure clang-static-analyzer
    build-command: build        # Execute the 'build' command for building and␣
↪analyzing the project
    command-line:               # Add additional arguments to the clang-static-
↪analyzer invocation
        - -enable-checker
        - alpha.clone.CloneChecker

make:
    build:
        patterns:
```

(continues on next page)

(continued from previous page)

```
            - MAKE_TARGET
        command-line:
            - "{MAKE_TARGET}"
    clean:
        command-line:
            - clean
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

Makefile:

```
CXX=g++
CXXFLAGS+=-O0 -g --coverage
LDFLAGS+=
SRC_DIR=src
BUILD_DIR=build

hello:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/hello $(SRC_DIR)/hello.cpp

world:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/world $(SRC_DIR)/world.cpp

clean:
        rm -rf $(BUILD_DIR)

.PHONY: clean
```

src/hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

src/world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.5 Clang-tidy plugin

5.5.1 Description

The clang-tidy plugin is used for executing the clang-tidy static code analysis tool.

5.5.2 Mandatory settings

There are no mandatory settings for the clang-tidy plugin.

5.5.3 Optional settings

The configuration of the clang-tidy plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

sources
: A list of sources that must be checked by the clang-tidy plugin. The sources may contain wildcards.

checks
: A list of checks that should be enabled or disabled. Enabling or disabling checks is done the same way as they are enabled on the clang-tidy command line. Default: no checks will be enabled or disabled on the command line, meaning the default checks enabled by clang will be checked.

warning-as-errors
: Threat warnings as errors. The value associated with this key is either:

  - A list of checks, defining which warnings will be threated as errors. See **checks** for the format.
  - The single keyword *all*: means that all enabled checks will be threated as errors.

Note: This options is only supported if the clang-tidy binary supports the `-warnings-as-error=<string>` option.

5.5.4 Example

Configuration

```yaml
commands:                       # Define the commands that can be run
    example: Run the make example

patterns:                       # Define the patterns that can be used
    TARGET:                     # Define targets to check
        default-values:         # Only define the default value
            - hello
            - world

example:
    - clang-tidy                # Use the clang-tidy plugin when running the 'example
↪' command

clang-tidy:
    patterns:
        - TARGET
    sources:
        - "src/{TARGET}.cpp"
    checks:
        - "*"
        - "cppcoreguidelines-*"
        - "modernize-*"
        - "performance-*"
        - "readability-*"
        - "-fuchsia-*"
        - "-llvmlibc-*"
    command-line:
        - -fix
```

Additional files

In order for the above example to work, the following files need to be created in the *src* directory:

hello.cpp:

```cpp
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```cpp
#include <cstdlib>
#include <iostream>
```

(continues on next page)

(continued from previous page)

```
auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.5.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.6 CMake plugin

5.6.1 Description

The cmake plugin is used for generating, building and installing software using the CMake build generator system.

5.6.2 Mandatory settings

There are no mandatory settings for this plugin, though it is recommended to configure the *mode* setting explicitly.

5.6.3 Optional settings

The configuration of the make plugin may contain the following settings:

Settings for all modes

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

mode
Set the mode of the CMake call for the specific command. Default: *generate*.

Supported modes are:

- **Generate**: For generating a build directory based on the CMake configuration in the source. This is often callend the *configure* or *build init* step.
- **Build**: Build the generated project
- **Install**: Install the generated project

build-dir
The path to the build directory. This is either an absolute path are a path relative to the location of this file. Default: . (the directory of the **exec-helper** configuration).

Settings for the *generate* mode

source-dir
The directory containing the root CMakeLists.txt file of the sources. Default: . (the directory of the **exec-helper** configuration).

generator
The generator to use for generating the build directory. See the CMake documentation on which generators are supported for your platform and the value(s) to explicitly set them. Default: the default one for your system and environment. See the CMake documentation on the details.

defines
A map of the *build generator settings* for configuring the generator.

Settings for the *build* mode

target
The specific CMake target to build. Default: the default target. See the CMake documentation for more details.

config
The configuration for multi-configuration tools. Default: the default configuration. See the CMake documentation for more details.

Settings for the *install* mode

config
The configuration for multi-configuration tools. Default: the default configuration. See the CMake documentation for more details.

prefix
Override the configured prefix set during the *generate* mode. Default: the default installation prefix. See the CMake documentation for more details.

component
Limit installation to the given component. Default: all installation targets.

5.6.4 Example

Configuration

```yaml
commands:                       # Define the commands that can be run
    example: Run the cmake example
    clean: Clean the build
    run: Run the files that were built

patterns:                       # Define the patterns that can be used
    CMAKE_TARGET:               # Define the CMAKE_TARGET pattern.
        default-values:         # Only define the default value
            - hello
            - world

example:
    - build                     # Use the cmake plugin when running the 'example' command
    - run

build:
    - generate
    - build-only
    - install

generate: cmake
build-only: cmake
install: cmake

clean:                          # Use the cmake plugin when running the 'clean' command
    - cmake

run:
    - command-line-command

cmake:
    environment:                # Define additional environment variables
        WORLD: "world!"
    patterns:                   # The patterns that are used by the cmake plugin
        - CMAKE_TARGET
    source-dir: .               # Set the source dir for all cmake targets that do not further specialize this
    build-dir: build            # Set the build dir for all cmake targets that do not further specialize this

    generate:                   # Specific settings for the 'generate' command
        mode: generate          # Set the mode
        defines:                # Set some defines
            CMAKE_BUILD_MODE: RelWithDebInfo
        command-line:           # Define additional command line arguments
            - -Wno-dev          # An example argument passed to cmake

    build-only:                 # Specific settings for the 'build-only' command
        mode: build             # Set the mode

    install:                    # Specific settings for the 'install' command
        mode: install           # Set the mode
        prefix: /tmp            # Set the prefix
```

(continues on next page)

(continued from previous page)

```
        component: runtime  # Limit to installing 'runtime' components

    clean:
        mode: build
        target: clean

command-line-command:
    patterns:
        - CMAKE_TARGET
    command-line:
        - build/{CMAKE_TARGET}
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(cmake-example CXX)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

add_executable(hello src/hello.cpp)
add_executable(world src/world.cpp)
install(TARGETS hello world DESTINATION bin COMPONENT runtime)
```

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.6.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.7 Command-line-command plugin

5.7.1 Description

The command-line-command plugin is used for executing arbitrary command lines. This plugin can be used for constructing the command line for commands that do not have a corresponding plugin available.

5.7.2 Mandatory settings

The configuration of the command-line-command must contain the following settings:

command-line
: The command-line to execute. There are two different usages:

  - **No identification key**: Set one command line as a list of separate arguments. This form is only usable if only one line needs to be executed.
  - **With identification key**: Make a map with arbitrary keys, where each associated value is one command line, described as a list of separate arguments. This form is usable if one or more lines need to be executed. Multiple commands are executed in the order the identification keys are defined.

  Note: see the documentation of **wordexp** (3) for the limitations on what characters are not allowed in the command-line command.

5.7.3 Optional settings

The configuration of the command-line-command plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

5.7.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: run the command-line example

patterns:                       # Define the patterns that can be used
    EXAMPLE_PATTERN:            # Define the EXAMPLE_PATTERN.
        default-values:         # Define the default value
            - world!

example:
    - command-line-command      # Use the command-line-command plugin when running␣
↪the 'example' command

command-line-command:           # Command-line-command configuration settings
    example:                    # Settings specific to the 'example' command
        environment:            # Define the environment
            EXAMPLE_ENVIRONMENT: hello
        patterns:               # Define the patterns that are used
            - EXAMPLE_PATTERN
        command-line:           # Define 2 command lines
            - hello: [echo, "${EXAMPLE_ENVIRONMENT}"]
            - world:            # The same as [echo, "{EXAMPLE_PATTERN}"]
                - echo
                - "{EXAMPLE_PATTERN}"
            - workingdir: [ echo, working, directory, is, "$(pwd)"]     # Print out␣
↪the current working directory
        working-dir: /tmp       # Set the working directory to a predefined value
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.7.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.8 Cppcheck plugin

5.8.1 Description

The cppcheck plugin is used for executing the cppcheck static code analysis tool.

5.8.2 Mandatory settings

There are no mandatory settings for the cppcheck plugin.

5.8.3 Optional settings

The configuration of the cppcheck plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

enable-checks
: A list of checks that should be enabled or disabled. Check the documentation of cppcheck for a list of all the available checks. Default: *all*.

src-dir
: The base directory containing all the files to check. Default: `.` (the current working directory).

5.8.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the cppcheck example

patterns:                       # Define the patterns that can be used
    TARGET:                     # Define targets to check
        default-values:         # Only define the default value
            - hello
            - world

example:
    - cppcheck                  # Use the cppcheck plugin when running the 'example'␣
↪command

cppcheck:                       # Cppcheck configuration for the 'example' command
    example:
        patterns:               # Define the patterns to use
            - TARGET
        src-dir:                # Define the source dir to look in
            - src
        target-path:            # The target path to look in
            - "{TARGET}.cpp"
        enable-checks:          # The list of additional checks to enable
```

(continues on next page)

(continued from previous page)

```
            - warning
            - style
            - performance
            - portability
            - information
        command-line:           # Set additional arguments
            - --error-exitcode=255
```

Additional files

In order for the above example to work, the following files need to be created in the *src* directory:

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.8.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.9 Docker plugin

5.9.1 Description

The Docker plugin is used for running or attaching to a Docker container.

5.9.2 Mandatory settings

Mandatory settings change depending on which mode is selected. See *mode* for more information.

5.9.3 Optional settings

The configuration of the make plugin may contain the following settings:

Settings for all modes

patterns
A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

mode
Set the mode of the Docker call for the specific command. Default: `exec`.

Supported modes are:

- **run**: Create a new container based on the given *image* and runs the given command. Note: use `--rm` as an additional command line argument to automatically clean up the created container.
- **exec**: Run the command in the given, actively running, *container*.

env
A map of environment key/value pairs set *inside* the container. Default: an empty map.

interactive
Boolean indicating whether to run interactively inside the container. Check the Docker documentation for more information. Default: same as the used Docker default.

tty
Boolean indicating whether to use a pseudo-tty inside the container. Check the Docker documentation for more information. Default: same as the used Docker default.

privileged
Boolean indicating whether to run the container in privileged mode. Check the Docker documentation for more information. Default: `no`.

user
Set the given user *inside* the container. Check the Docker documentation for more information. Default: the container default.

Settings for the *run* mode

volumes
List of volumes to be mounted into the container. Eeach value maps directly to a Docker volume configuration. Check the Docker documentation for all the options and formats that can be used. Default: an empty list.

image
The Docker *image* to use as the base image for creating a new container. This configuration option is *mandatory* when the plugin is in *run* mode.

Settings for the *exec* mode

container
The Docker *container* to execute the command in. Note that the container *must* already be running when this command is called. This configuration option is *mandatory* when the plugin is in *exec* mode.

5.9.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the docker example
    run: Show the contents of the /example folder

patterns:                       # Define the patterns that can be used
    IMAGE:                      # Define the IMAGE
        default-values:         # Define the default value(s)
            - ubuntu:rolling
        short-option: i         # Define the short option for overriding the default␣
↪value
        long-option: image      # Define the long option for overriding the default␣
↪value

    COMMAND:
        default-values:
            - ls
            - echo

example:
    - docker                    # Use the docker plugin when running the 'example'␣
↪command

ls:
    - command-line-command      # Use the 'command-line-command' plugin for␣
↪constructing the 'ls' command

echo:
    - command-line-command      # Use the 'command-line-command' plugin for␣
↪constructing the 'echo' command

docker:
    example:
        patterns:               # Define the patterns we will use for this command.
            - IMAGE             # Use the IMAGE pattern => all occurences of '{IMAGE}
↪' will be replaced by the actual value
```

(continues on next page)

(continued from previous page)

```
            - COMMAND
        mode: run                # Use the 'run' mode
        image: "{IMAGE}"         # Set the image. The quotes "" are required due to␣
↪the YAML specification and its JSON compatibility.
        envs:                    # Define additional environment variables inside the␣
↪container
            SHELL: xterm-color   # Use a YAML dictionary to define all kay-value pairs
        interactive: yes         # Run an interactive shell in the container
        tty: no                  # Do not attach to a pseudo-tty in the container
        privileged: no           # Do not run a privileged container
        user: root               # Explicitly run as the root user
        volumes:
            - "${PWD}:/examples"  # Mount the folder of this configuration file in␣
↪the container on the /examples path
        targets: "{COMMAND}"     # Run the 'run' task in the configured container

command-line-command:
    ls:                         # Configure the 'run' command
        command-line: [ ls, -la, /root]  # Run 'ls -la /root'

    echo:                        # configure the 'echo' command
        command-line: [ echo, Hello world]   # Run 'echo Hello world'
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.9.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.10 Execute plugin

5.10.1 Description

The execute plugin is used for executing specific plugins or, if no associated plugin is found, following commands defined in the configuration. This plugin is mainly used by other plugins that want to execute other commands.

5.10.2 Mandatory settings

There are no mandatory settings for this plugin.

5.10.3 Optional settings

There are no optional settings for this plugin.

5.10.4 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.11 Fish plugin

5.11.1 Description

The fish plugin is used for executing commands in the *fish* shell, rather than executing the command right away. This is very useful for executing command lines that need special shell characters like `&&`, `|`, `;`, `>`.

5.11.2 Mandatory settings

Mandatory settings for all modes

command
: Command to execute in the shell, as a string. See the `-c` option of `fish` for more information.

5.11.3 Optional settings

The configuration of the fish plugin may contain the following additional settings:

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

5.11.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: run the fish example

patterns:                       # Define the patterns that can be used
    EXAMPLE_PATTERN:            # Define the EXAMPLE_PATTERN.
        default-values:         # Define the default value
            - world!

example:
    - fish                        # Use the fish plugin when running the 'example'␣
↪command

fish:                             # Sh plugin configuration settings
    example:                    # Settings specific to the 'example' command
        environment:            # Define the environment
            EXAMPLE_ENVIRONMENT: hello
        patterns:               # Define the patterns that are used
            - EXAMPLE_PATTERN
        command: 'echo {$EXAMPLE_ENVIRONMENT} && echo {EXAMPLE_PATTERN} && echo
↪"working directory is" (pwd)'      # Define the shell command
        command-line: [ --debug=exec-fork ]     # Pass additional command line␣
↪arguments
        working-dir: /tmp       # Set the working directory to a predefined value
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.11.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.12 Lcov plugin

5.12.1 Description

The lcov plugin is used for executing code coverage analysis using lcov.

5.12.2 Mandatory settings

The configuration of the lcov plugin must contain the following settings:

run-command
: The **exec-helper** command or plugin to use for running the binaries for which the coverage needs to be analyzed.

5.12.3 Optional settings

The configuration of the lcov plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

info-file
: The lcov .info file to use for the analysis. Default: `lcov-plugin.info`.

base-directory
: The base directory to use for the lcov analysis. Check the lcov documentation on the `--base-directory` option for more information. Default: `.` (the current working directory).

directory
: Use the coverage data files in the given directory. Check the lcov documentation on the `--directory` option for more information. Default: `.` (the current working directory).

zero-counters
: Set this option to *yes* to reset the coverage counters before starting the analysis. All other values are threated as *no*. Default: `no`.

gen-html
: Set this option to *yes* to enable HTML report generation of the coverage data. Default: `no`.

gen-html-output
: Set the output directory of the generated HTML report. Does nothing if **gen-html** is not enabled. Default: `.` (the current working directory).

gen-html-title
: Set the title of the generated HTML report. Does nothing if **gen-html** is not enabled. Default: `Hello`.

gen-html-command-line
: Set additional command line options for the gen html stage. Default: no additional command line options.

excludes
: A list of directories and files to excluse from the coverage report. The paths are relative to the current working directory. Default: an empty list.

5.12.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the lcov example
    build: Build the files
    clean: Clean the build
    run: Run the built binaries
```

(continues on next page)

(continued from previous page)

```
patterns:                          # Define the patterns that can be used
    MAKE_TARGET:                   # Define make targets for building
        default-values:            # Only define the default value
            - hello
            - world

example:
    - build
    - lcov                         # Use the lcov plugin when running the 'example'␣
↪command

build:
    - make

clean:
    - make
    - command-line-command

run:
    command-line-command

lcov:                              # Configure lcov
    run-command: run               # Execute the 'build' command for building, running␣
↪and analyzing the project
    info-file: build/coverage.info  # Create and use the coverage.info file in the␣
↪build dir
    base-directory: .              # LCOV's base-directory functionality
    directory: .                   # LCOV's directory functionality
    zero-counters: yes             # Zero the counters before executing the analysis
    gen-html: yes                  # Generate a HTML coverage report
    gen-html-output: build/coverage # Output the HTML coverage report to build/
↪coverage
    gen-html-title: "LCOV-example" # Set the title of the HTML coverage report
    excludes:                       # Set which entries to exclude from the report
        - /usr/include/*

make:
    build:
        patterns:
            - MAKE_TARGET
        command-line:
            - "{MAKE_TARGET}"
    clean:
        command-line:
            - clean

command-line-command:
    patterns:
        - MAKE_TARGET
    run:
        command-line: ["build/{MAKE_TARGET}"]
    clean:
        command-line:
            remote-gcda-file: [ rm, -rf, "{MAKE_TARGET}.gcda"]
            remote-gcno-file: [ rm, -rf, "{MAKE_TARGET}.gcno"]
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

Makefile:

```
CXX=g++
CXXFLAGS+=-O0 -g --coverage
LDFLAGS+=
SRC_DIR=src
BUILD_DIR=build

hello:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/hello $(SRC_DIR)/hello.cpp

world:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/world $(SRC_DIR)/world.cpp

clean:
        rm -rf $(BUILD_DIR)

.PHONY: clean
```

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.12.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.13 Make plugin

5.13.1 Description

The make plugin is used for executing Makefiles.

5.13.2 Mandatory settings

There are no mandatory settings for this plugin.

5.13.3 Optional settings

The configuration of the make plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

build-dir
: The path to the Makefile. This is either an absolute path are a path relative to the location of this file. Default: `.` (the current working directory).

5.13.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the make example
    clean: Clean the build
    run: Run the files that were built

patterns:                       # Define the patterns that can be used
    MAKE_TARGET:                # Define the EXAMPLE_PATTERN.
        default-values:         # Only define the default value
            - hello
            - world

example:
    - clean
```

(continues on next page)

(continued from previous page)

```
    - make                          # Use the make plugin when running the 'example'␣
↪command
    - run

clean:                              # Use the make plugin when running the 'clean' command
    - make

run:
    - command-line-command

make:
    environment:                    # Define additional environment variables
        WORLD: "world!"
    example:                        # Specific settings for the 'example' command
        patterns:                   # The patterns that are used by the make plugins
            - MAKE_TARGET
        build-dir: $(pwd)           # Set the build dir
        command-line:               # Define additional command line arguments
            - --keep-going          # An example argument passed to make
            - "{MAKE_TARGET}"       # Define the make target to execute
    clean:
        command-line:
            - clean

command-line-command:
    patterns:
        - MAKE_TARGET
    command-line:
        - build/{MAKE_TARGET}
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

Makefile:

```
CXX=g++
CXXFLAGS+=-O0 -g --coverage
LDFLAGS+=
SRC_DIR=src
BUILD_DIR=build

hello:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/hello $(SRC_DIR)/hello.cpp

world:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/world $(SRC_DIR)/world.cpp

clean:
        rm -rf $(BUILD_DIR)

.PHONY: clean
```

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.13.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Custom plugins* (5) for the available plugins and their configuration options.

See *Configuration* (5) for information about the configuration file format.

5.14 Meson plugin

5.14.1 Description

The meson plugin is used for setting up, compiling, installing and testing software using the Meson build generator system.

5.14.2 Mandatory settings

Mandatory settings for all modes

mode
: Set the mode of the Meson call for the specific command. Default: *setup*.

  Supported modes are:

  - **setup**: For setting up the build directory based on the Meson configuration in the source. This is often callend the *configure* or *build init* step.
  - **compile**: Compiles (or builds) the generated project

- **test**: Run the configured test suite using Meson
- **install**: Install the generated project

5.14.3 Optional settings

The configuration of the meson plugin may contain the following additional settings:

Settings for all modes

patterns
A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

build-dir
The path to the build directory. This is either an absolute path are a path relative to the location of this file. Default: . (the directory of the **exec-helper** configuration).

Additional settings for the *setup* mode

source-dir
The directory containing the root `meson.build` file of the sources. Default: . (the directory of the **exec-helper** configuration).

build-type
Set the Meson build type explicitly. See the `--buildtype` parameter of `meson setup` for more information.

cross-file
Set the Meson cross-file. See the `--cross-file` parameter of `meson setup` for more information.

prefix
Set the Meson installation prefix. See the `--prefix` parameter of `meson setup` for more information.

options
A map of the options to set for setting up the build. See the `-D` parameter of :code‘meson setup‘ for more information.

Additional settings for the *compile* mode

jobs
Fix the number of jobs to use. Default: *auto* or the number of jobs set on the **exec-helper** invocation.

Additional settings for the *test* mode

suites
Set the test suites to run. By default, this parameter is omitted.

targets
Set the targets to run. By default, this parameter is omitted.

5.14.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the meson example
    run: Run the files that were built

patterns:                       # Define the patterns that can be used
    MESON_TARGET:               # Define the MESON_TARGET pattern.
        default-values:         # Only define the default value
            - hello
            - world

example:
    - build                     # Use the meson plugin when running the 'example'␣
↪command
    - run

build:                          # Subdivide the 'build' command into three␣
↪consecutive commands
    - generate
    - build-only
    - install

generate: meson                 # Define the subcommands. These commands can be␣
↪called directly to.
build-only: meson
install: meson

run:
    - command-line-command  # Use the command-line-command plugin for the 'run'␣
↪command

meson:
    environment:            # Define additional environment variables
        WORLD: "world!"

    prefix: /tmp            # Set the installation prefix
    source-dir: .           # Set the source dir for all meson targets that do not␣
↪further specialize this
    build-dir: build        # Set the build dir for all meson targets that do not␣
↪further specialize this

    generate:               # Specific settings for the 'generate' command
        mode: setup         # Set the mode
        options:            # Set some defines
            test: true
```

(continues on next page)

(continued from previous page)

```
        command-line:        # Define additional command line arguments
            - --strip        # An example argument passed to make

    build-only:              # Specific settings for the 'build-only' command
        mode: compile        # Set the mode
        jobs: 1              # Always compile with one thread

    install:                 # Specific settings for the 'install' command
        mode: install        # Set the mode

command-line-command:
    run:
        patterns:                # The patterns that are used by the 'run' command
            - MESON_TARGET
        command-line:
            - build/{MESON_TARGET}
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

meson.build:

```
project('example', 'cpp',
  version: '0.1.0',
  default_options: [
    'cpp_std=c++17',
  ]
)

hello = executable('hello', ['src/hello.cpp'],
  install : true,
)

world = executable('world', ['src/world.cpp'],
  install : true,
)
```

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
```

(continues on next page)

(continued from previous page)

```
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.14.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.15 Ninja plugin

5.15.1 Description

The ninja plugin is used for executing Makefiles.

5.15.2 Mandatory settings

There are no mandatory settings for this plugin.

5.15.3 Optional settings

The configuration of the ninja plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
: The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

build-dir
: The path to the build directory. This is either an absolute path are a path relative to the location of this file. Default: `.` (the current working directory).

5.15.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the ninja example
    clean: Clean the build
    run: Run the files that were built

patterns:                       # Define the patterns that can be used
    TARGET:                     # Define the EXAMPLE_PATTERN.
        default-values:         # Only define the default value
            - hello
            - world

example:
    - clean
    - ninja                      # Use the ninja plugin when running the 'example'␣
↪command
    - run

clean:                          # Use the ninja plugin when running the 'clean'␣
↪command
    - ninja

run:
    - command-line-command

ninja:
    environment:                # Define additional environment variables
        WORLD: "world!"
    build-dir: .                # Set the build dir
    example:                    # Specific settings for the 'example' command
        patterns:               # The patterns that are used by the ninja plugins
            - TARGET
        command-line:           # Define additional command line arguments
            - -k                # An example argument passed to ninja
            - 2
            - "{TARGET}"    # Define the ninja target to execute
    clean:
        command-line:
            - clean

command-line-command:
    patterns:
        - TARGET
    command-line:
        - build/ninja/{TARGET}
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

ninja.build:

```
CXX = g++
CXXFLAGS = -Wall
LDFLAGS =
BUILD_DIR = build/ninja

rule cc
    command = $CXX $CXXFLAGS $LDFLAGS -o $out $in

rule rmdir
    command = rm -rf $dir

build $BUILD_DIR/hello: cc src/hello.cpp
build hello: phony $BUILD_DIR/hello

build $BUILD_DIR/world: cc src/world.cpp
build world: phony $BUILD_DIR/world

build clean: rmdir
    dir = $BUILD_DIR

build all: phony hello world

default all
```

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.15.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Custom plugins* (5) for the available plugins and their configuration options.

See *Configuration* (5) for information about the configuration file format.

5.16 Pmd plugin

5.16.1 Description

The pmd plugin is used for executing the pmd static code analyzer tool suite.

5.16.2 Mandatory settings

There are no mandatory settings for this plugin.

5.16.3 Optional settings

The configuration of the pmd plugin may contain the following settings:

patterns
A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

exec
The path to the pmd-run executable. The path can either be an absolute path or a relative path from the current working directory. Default: `pmd`.

tool
The pmd tool to use. The currently supported tools are:

- cpd

Default: `cpd`

language
Specify the language PMD is analyzing. Check the `--language` option of the pmd documentation for more information. Default: no explicit language parameter is passed.

Cpd specific settings

minimum-tokens
The minimum token length to be considered a duplicate. Check the `--minimum-tokens` option of the cpd documentation for more information. Default: no explicit minimum tokens parameter is passed.

files
A list of files to check for duplicated code. Check the `--files` option of the cpd documentation for more information. Default: no explicit files parameter is passed.

5.16.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the pmd example

patterns:                       # Define the patterns that can be used
    TARGET:                     # Define targets to check
        default-values:         # Only define the default value
            - hello
            - world

example:
    - pmd                       # Use the cppcheck plugin when running the 'example'␣
↪command

pmd:                            # Cppcheck configuration for the 'example' command
    example:
        patterns:               # Define the patterns to use
            - TARGET
        exec: pmd
        tool: cpd
        language: cpp
        minimum-tokens: 100
        files: src/{TARGET}.cpp
        command-line:           # Set additional arguments
            - --non-recursive
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

Makefile:

```
CXX=g++
CXXFLAGS+=-O0 -g --coverage
LDFLAGS+=
SRC_DIR=src
BUILD_DIR=build

hello:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/hello $(SRC_DIR)/hello.cpp

world:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/world $(SRC_DIR)/world.cpp

clean:
        rm -rf $(BUILD_DIR)

.PHONY: clean
```

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.16.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Custom plugins* (5) for the available plugins and their configuration options.

See *Configuration* (5) for information about the configuration file format.

5.17 Scons plugin

5.17.1 Description

The scons plugin is used for executing scons.

5.17.2 Mandatory settings

There are no mandatory settings for this plugin.

5.17.3 Optional settings

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
: A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

build-dir
The path to the build directory. This is either an absolute path are a path relative to the location of this file. Default: `.` (the current working directory).

5.17.4 Example

Configuration

```
commands:                          # Define the commands that can be run
    example: Run the scons example
    clean: Clean all built files
    run: Run the built binaries

patterns:                          # Define the patterns that can be used
    SCONS_TARGET:                # Define the EXAMPLE_PATTERN.
        default-values:            # Only define the default value
            - hello
            - world

example:
    - clean
    - scons                        # Use the command-line-command plugin when running␣
↪the 'example' command
    - run

clean:
    - command-line-command

run:
    - command-line-command

scons:
    patterns:                      # The patterns that are used by the make plugins
        - SCONS_TARGET
    example:                       # Specific settings for the 'example' command
        command-line:              # Define additional command line arguments
            - --keep-going         # Pass additional options to scons
            - "{SCONS_TARGET}"     # Define the make target to execute

command-line-command:
    clean:
        command-line: [rm, -rf, build]
    run:
        patterns:
            - SCONS_TARGET
        command-line: ["build/{SCONS_TARGET}"]
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

SConstruct:

```
env = Environment()
Export('env')

SConscript('src/SConscript', variant_dir='build', duplicate=0)

Default(None)
```

SConscript:

```
Import('env')

hello = env.Program('hello.cpp')
env.Alias('hello', hello)

world = env.Program('world.cpp')
env.Alias('world', world)
```

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.17.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Custom plugins* (5) for the available plugins and their configuration options.

See *Configuration* (5) for information about the configuration file format.

5.18 Selector plugin

Description The selector plugin is used for selecting a configuration path based on the value(s) of a target, typically one from a pattern value.

5.18.1 Mandatory settings

The configuration of the command-line-command must contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

targets
: The targets to select on. Note that if patterns are used in this list, they must be listed using the *patterns* configuration, as is the case for every plugin.

  The runtime value(s) associated with the pattern key must resolve either to an existing (configured) plugin or a configured command.

5.18.2 Optional settings

There are no optional settings for the selector plugin.

5.18.3 Example

Configuration

```
# Usage:
#   'exec-helper --settings-file <this file> example' will execute both the example1␣
↪and example2 target.
# Adding the --example <example-value> will only execute the given <example-value>. E.
↪g.:
#   'exec-helper --settings-file <this file> example --example example1' will execute␣
↪the example1 target only.

commands:
    example: An example for using the selector plugin

patterns:
    SELECTOR:                       # Define the pattern to select on.
        default-values:
            - example1
            - example2
        short-option: e
        long-option: --example

example:                            # Use the selector for the example command
    - selector

selector:
    patterns:
        - SELECTOR              # Tell the selector plugin to use the SELECTOR␣
↪pattern for deciding which paths to trigger
```

(continues on next page)

(continued from previous page)

```
    targets: [ "{SELECTOR}" ]   # Execute the target when the selector is activated.␣
↪The target is a permutation of the values in the registered patterns

example1:                           # Define the 'example1' path
    - command-line-command

example2:                           # Define the 'example2' path
    - command-line-command

command-line-command:
    example1:
        command-line: [ echo, example1]
    example2:
        command-line: [ echo, example2]
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.18.4 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Custom plugins* (5) for the available plugins and their configuration options.

See *Configuration* (5) for information about the configuration file format.

5.19 Sh plugin

5.19.1 Description

The sh plugin is used for executing commands in the *sh* shell, rather than executing the command right away. This is very useful for executing command lines that need special shell characters like `&&`, `|`, `;`, `>`.

5.19.2 Mandatory settings

Mandatory settings for all modes

command
: Command to execute in the shell, as a string. See the `-c` option of `sh` for more information.

5.19.3 Optional settings

The configuration of the sh plugin may contain the following additional settings:

command-line
Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

patterns
A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

5.19.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: run the sh example

patterns:                       # Define the patterns that can be used
    EXAMPLE_PATTERN:            # Define the EXAMPLE_PATTERN.
        default-values:         # Define the default value
            - world!

example:
    - sh                        # Use the sh plugin when running the 'example' command

sh:                             # Sh plugin configuration settings
    example:                    # Settings specific to the 'example' command
        environment:            # Define the environment
            EXAMPLE_ENVIRONMENT: hello
        patterns:               # Define the patterns that are used
            - EXAMPLE_PATTERN
        command: 'echo ${EXAMPLE_ENVIRONMENT} && echo {EXAMPLE_PATTERN} && echo
↪"working directory is $(pwd)"'      # Define the shell command
        command-line: [ -ex]     # Pass additional command line arguments
        working-dir: /tmp       # Set the working directory to a predefined value
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.19.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.20 Valgrind plugin

5.20.1 Description

The valgrind plugin is used for executing code coverage analysis using valgrind.

5.20.2 Mandatory settings

The configuration of the valgrind plugin must contain the following settings:

run-command
: The exec-helper command or plugin to use for running the binaries which need to be analyzed.

5.20.3 Optional settings

The configuration of the valgrind plugin may contain the following settings:

patterns
: A list of patterns to apply on the command line. See *Patterns* (5).

command-line
: Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

tool
: The valgrind tool to use. Default: the `tool` is omitted.

5.20.4 Example

Configuration

```
commands:                       # Define the commands that can be run
    example: Run the lcov example
    build: Build the files
    clean: Clean the build
    run: Run the built binaries

patterns:                       # Define the patterns that can be used
    MAKE_TARGET:                # Define make targets for building
        default-values:         # Only define the default value
            - hello
            - world

example:
    - build
    - valgrind                  # Use the valgrind plugin when running the 'example'␣
↪command
```

(continues on next page)

(continued from previous page)

```
build:
    - make

clean:
    - make
    - command-line-command

run:
    command-line-command

valgrind:                          # Configure the valgrind plugin
    run-command: run               # Execute the 'build' command for building, running␣
↪and analyzing the project
    tool: memcheck                 # Set the tool
    command-line:                  # Set additional arguments for valgrind
        - --error-exitcode=255

make:
    build:
        patterns:
            - MAKE_TARGET
        command-line:
            - "{MAKE_TARGET}"
    clean:
        command-line:
            - clean

command-line-command:
    patterns:
        - MAKE_TARGET
    run:
        command-line: ["build/{MAKE_TARGET}"]
    clean:
        command-line:
            remote-gcda-file: [ rm, -rf, "{MAKE_TARGET}.gcda"]
            remote-gcno-file: [ rm, -rf, "{MAKE_TARGET}.gcno"]
```

Additional files

In order for the above example to work, the following file hierarchy needs to be created in the directory:

Makefile:

```
CXX=g++
CXXFLAGS+=-O0 -g --coverage
LDFLAGS+=
SRC_DIR=src
BUILD_DIR=build

hello:
        mkdir -p $(BUILD_DIR)
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/hello $(SRC_DIR)/hello.cpp

world:
        mkdir -p $(BUILD_DIR)
```

(continues on next page)

(continued from previous page)

```
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(BUILD_DIR)/world $(SRC_DIR)/world.cpp

clean:
        rm -rf $(BUILD_DIR)

.PHONY: clean
```

hello.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "Hello" << std::endl;
    return EXIT_SUCCESS;
}
```

world.cpp:

```
#include <cstdlib>
#include <iostream>

auto main() -> int {
    std::cout << "World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.20.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Custom plugins* (5) for the available plugins and their configuration options.

See *Configuration* (5) for information about the configuration file format.

5.21 Zsh plugin

5.21.1 Description

The zsh plugin is used for executing commands in the *zsh* shell, rather than executing the command right away. This is very useful for executing command lines that need special shell characters like `&&`, `|`, `;`, `>`.

5.21.2 Mandatory settings

Mandatory settings for all modes

command
Command to execute in the shell, as a string. See the `-c` option of `zsh` for more information.

5.21.3 Optional settings

The configuration of the zsh plugin may contain the following additional settings:

command-line
Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

patterns
A list of patterns to apply on the command line. See *Patterns* (5).

enviroment
A list of environment variables that should be set before the commands are executed. See *Environment* (5).

command-line
Additional command line parameters to pass as a list of separate arguments. By default no additional arguments are added.

working-dir
The working directory of the command. Can be an absolute path are a relative one w.r.t. the path to the considered configuration file. Commands that should be executed relative to the current working dir can use the `{EH_WORKING_DIR}` pattern.

5.21.4 Example

Configuration

```
commands:                          # Define the commands that can be run
    example: run the zsh example

patterns:                          # Define the patterns that can be used
    EXAMPLE_PATTERN:               # Define the EXAMPLE_PATTERN.
        default-values:            # Define the default value
            - world!

example:
    - zsh                           # Use the zsh plugin when running the 'example'␣
↪command

zsh:                                # Sh plugin configuration settings
    example:                       # Settings specific to the 'example' command
        environment:               # Define the environment
            EXAMPLE_ENVIRONMENT: hello
        patterns:                  # Define the patterns that are used
            - EXAMPLE_PATTERN
        command: 'echo ${EXAMPLE_ENVIRONMENT} && echo {EXAMPLE_PATTERN} && echo
↪"working directory is $(pwd)"'      # Define the shell command
```

(continues on next page)

(continued from previous page)

```
        command-line: [ -ex]      # Pass additional command line arguments
        working-dir: /tmp        # Set the working directory to a predefined value
```

Usage

Save the example to an **exec-helper** configuration file and execute in the same directory:

```
eh example
```

5.21.5 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Configuration* (5) for information about the configuration file format.

See *Plugins* (5) for information about the configuration file format.

5.22 Description

This document describes the list of **plugins** that can be used in the associated **exec-helper** binaries.

5.23 General plugins

command-line-command
: The command-line-command plugin is used for executing arbitrary command line commands. See *Command-line-command plugin* (5).

sh
: The sh plugin is used for executing arbitrary commands in the sh shell. This is very useful for executing command lines that need special shell characters like `&&`, `|`, `;`, >. See *Sh plugin* (5).

bash
: The bash plugin is used for executing arbitrary commands in the bash shell. This is very useful for executing command lines that need special shell characters like `&&`, `|`, `;`, >. See *Bash plugin* (5).

fish
: The fish plugin is used for executing arbitrary commands in the fish shell. This is very useful for executing command lines that need special shell characters like `&&`, `|`, `;`, >. See *Fish plugin* (5).

zsh
: The zsh plugin is used for executing arbitrary commands in the zsh shell. This is very useful for executing command lines that need special shell characters like `&&`, `|`, `;`, >. See *Zsh plugin* (5).

selector
: The selector plugin is used for selecting certain configuration paths based on the value of a pattern. See *Selector plugin* (5).

docker
: The docker plugin is used for running commands inside a Docker container. See *Docker plugin* (5).

5.24 Build plugins

bootstrap
The bootstrap plugin is used for calling bootstrap scripts, typically used as a step in a build chain. See *Bootstrap plugin* (5).

make
The make plugin is used for running the make build system. See *Make plugin* (5).

scons
The scons plugin is used for running the scons build system. See *Scons plugin* (5).

cmake
The cmake plugin is used for running the CMake build system. See *CMake plugin* (5).

meson
The meson plugin is used for running the CMake build system. See *Meson plugin* (5).

5.25 Analysis plugins

clang-static-analyzer
The clang-static-analyzer plugin is used for applying the clang static analyzer tool on source code files. See *Clang-static-analyzer plugin* (5).

clang-tidy
The clang-tidy plugin is used for applying the clang tidy tool on source code files. See *Clang-tidy plugin* (5).

cppcheck
The cppcheck plugin is used for applying cppcheck on source code files. See *Cppcheck plugin* (5).

lcov
The lcov plugin is used for applying the lcov code coverage analysis tool. See *Lcov plugin* (5).

pmd
The pmd plugin is used for applying pmd analysis on source code files. See *Pmd plugin* (5).

valgrind
The valgrind plugin is used for applying valgrind analysis. See *Valgrind plugin* (5).

5.26 Custom plugins

You can write your own plugins and integrate them with **exec-helper**. These plugins are first-class citizens: you can write plugins that overwrite the system plugins themselves. See *Custom plugins* (5) for more information on writing your own plugins.

5.27 See also

See *exec-helper* (1) for information about the usage of **exec-helper**.

See *Custom plugins* (5) for the available plugins and their configuration options.

See *Configuration* (5) for information about the configuration file format.

CHAPTER 6

Feature documentation

6.1 Command line arguments

```
@cmd_args @no_args
Feature: Calling exec-helper without command-line options
    Scenarios for calling exec-helper without command-line options

    Background:
        Given a controlled environment

    @successful
    Scenario: The application is called with no command line arguments and no valid␣
↪configuration file
        When we call the application
        Then the call should fail with return code 1
        And stderr should contain 'Could not find an exec-helper settings file'

    @successful
    Scenario: The application is called with no command line arguments and a valid␣
↪configuration file
        Given a valid configuration
        When we call the application
        Then the call should fail with return code 1
        And stderr should contain 'must define at least one command'
```

```
@cmd_args @invalid_args
Feature: Call the application with invalid arguments
    Scenarios for when the application is called with invalid command-line arguments

    Examples:
    | command_line              |
    | -b                        |
    | --blaat                   |
```

(continues on next page)

(continued from previous page)

```
    | -b blaat                  |
    | --blaat blaat             |
    | --blaat blaat --foo bar   |

    Background:
        Given a controlled environment

    @error
    Scenario: The version option is defined on a valid command line
        Given a valid configuration
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should fail with return code 1
        And stderr should contain 'unrecognised option'
        And stdout should contain 'Usage'
        And stdout should contain '--help'

    @error
    Scenario: The version option is defined on a valid command line with no␣
↪configuration file
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should fail with return code 1
        And stderr should contain 'Could not find an exec-helper settings file'
        And stderr should not contain 'unrecognised option'
        And stdout should contain 'Usage'
        And stdout should contain '--help'
```

```
@cmd_args @help_option
Feature: Use the help command-line option
    Scenarios for when the help option is given on the command line

    Examples:
    | command_line                    |
    | -h                              |
    | --help                          |
    | --help --version --debug debug  |
    | --debug debug --help --version  |
    | --version --debug debug --help  |

    Background:
        Given a controlled environment

    @successful
    Scenario: The help option is defined on a valid command line
        Given a valid configuration
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'Usage'
        And stdout should contain 'Optional arguments:'
        And stdout should not contain 'Configured commands:'

    @successful
    Scenario: The help option is defined on a valid command line with no␣
↪configuration file
```

(continues on next page)

(continued from previous page)

```
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'Usage: exec-helper [Optional arguments] COMMANDS...
↪'
        And stdout should contain 'Optional arguments:'
        And stdout should not contain 'Configured commands:'

    @successful
    Scenario: The help option is defined for a configuration with a command
        Given a valid configuration
        And the <command> command
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'Usage: exec-helper [Optional arguments] COMMANDS...
↪'
        And stdout should contain 'Optional arguments:'
        And stdout should contain 'Configured commands:'
        And stdout should contain <command>

        Examples:
        | command  |
        | Command1 |

    @successful
    Scenario: The help option is defined for a configuration with a pattern
        Given a valid configuration
        And the <pattern> pattern
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'Usage: exec-helper [Optional arguments] COMMANDS...
↪'
        And stdout should contain 'Optional arguments:'
        And stdout should not contain 'Configured commands:'
        And stdout should contain 'Values for pattern'

        Examples:
        | pattern
↪ |
        | { "key": "PATTERN", "long_options": ["blaat"], "default_values": ["blaat"] }
↪ |

    @successful
    Scenario: The help option is defined for a configuration with a pattern and a␣
↪command
        Given a valid configuration
        And the <command> command
        And the <pattern> pattern
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'Usage: exec-helper [Optional arguments] COMMANDS...
↪'
        And stdout should contain 'Optional arguments:'
        And stdout should contain 'Configured commands:'
```

(continues on next page)

(continued from previous page)

```
        And stdout should contain 'Values for pattern'
        And stdout should contain <command>

        Examples:
        | command  | pattern
↪            |
        | Command1 | { "key": "PATTERN", "long_options": ["blaat"], "default_values":␣
↪["blaat"] }  |
```

```
@cmd_args @version_option
Feature: Use the version command-line option
    Scenarios for when the version option is given on the command line

    Examples:
    | command_line                      |
    | --version                         |
    | --version --debug debug --dry-run |
    | --debug debug --version --dry-run |
    | --dry-run --debug debug --version |

    Background:
        Given a controlled environment

    @successful
    Scenario: The version option is defined on a valid command line
        Given a valid configuration
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'exec-helper'
        And stdout should contain 'COPYRIGHT'

    @successful
    Scenario: The version option is defined on a valid command line with no␣
↪configuration file
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'exec-helper'
        And stdout should contain 'COPYRIGHT'
```

```
@cmd_args @dry_run_option
Feature: Use the dry run command-line option
    Scenarios for when the dry run option is given on the command line

    Examples:
    | command_line                      |
    | -n                                |
    | --dry-run                         |
    | --dry-run --debug debug --verbose |
    | --debug debug --dry-run --verbose |
    | --verbose --debug debug --dry-run |

    Background:
        Given a controlled environment
```

(continues on next page)

(continued from previous page)

```
    @successful
    Scenario: The keep-going option is defined on a valid command line
        Given a valid configuration
        When we add the <command> command
        And we add the <command_line> as command line arguments
        And we add the <command> to the command line options
        When we call the application
        Then the call should succeed
        And the <command> command should be called 0 times

        Examples:
        | command  |
        | describe |
```

```
@cmd_args @keep_going_option
Feature: Use the keep-going command-line option
    Scenarios for when the keep-going option is given on the command line

    Examples:
    | command_line                          |
    | -k                                    |
    | --keep-going                          |
    | --keep-going --debug debug --verbose  |
    | --debug debug --keep-going --verbose  |
    | --verbose --debug debug --keep-going  |

    Background:
        Given a controlled environment

    @successful
    Scenario: The keep-going option is defined on a valid command line
        Given a valid configuration
        When we add the <command> that returns <return_code>
        And we add the <command_line> as command line arguments
        And we add the <command> <nb_of_times> to the command line options
        When we call the application
        Then the call should fail with return code <return_code>
        And the <command> command should be called <nb_of_times> times

        Examples:
        | command | return_code   | nb_of_times |
        | fail    | 0             | 1           |
        | fail    | 0             | 3           |
        | fail    | 1             | 1           |
        | fail    | 1             | 4           |
```

```
@cmd_args @list_plugins_option
Feature: Use the 'list plugins' command-line option
    Scenarios for when the 'list plugins' option is given on the command line

    Examples:
    | command_line                           |
    | --list-plugins                         |
    | --list-plugins --debug debug --dry-run |
    | --dry-run --list-plugins --debug debug |
    | --debug debug --dry-run --list-plugins |
```

(continues on next page)

(continued from previous page)

```
    Background:
        Given a controlled environment

    @successful
    Scenario: The 'list plugins' option is defined on a valid command line
        Given a valid configuration
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'command-line-command'

    @successful
    Scenario: The 'list plugins' option is defined on a valid command line with no␣
↪configuration file
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain 'command-line-command'
```

6.2 Configuration

Usage information on the configuration can be found on the *Configuration* page.

```
@config @environment
Feature: Test settings the environment for the configured commands
    Scenarios for setting the environment for configured command(s)

    Background:
        Given a controlled environment
        And a valid configuration

    @successful
    Scenario: Set the environment to a fixed value
        Given the <command> command
        And the <environment> is configured for <command> command in the configuration
        When we run the <command> command
        Then the call should succeed
        And the runtime environment for <command> should contain the given
↪<environment>

        Examples:
        | command   | environment                          |
        | Command1  | KEY1:VALUE1                          |
        | Command2  | KEY1:VALUE1;KEY2:VALUE2;KEY3:VALUE3  |

    @successful
    Scenario: Replace patterns in the configured environment
        Given the <command> command
        And the <pattern> pattern
        And the <pattern> is configured for <command> command in the configuration
        And the <environment> is configured for <command> command in the configuration
        When we run the <command> command
        Then the call should succeed
```

(continues on next page)

(continued from previous page)

```
        And the runtime environment for <command> should contain the given
↪<environment>

        Examples:
        | command  | pattern                                                  |␣
↪environment                                 |
        | Command1 | { "key": "PATTERN", "default_values": ["blaat"] }    | KEY:
↪{PATTERN}                              |
        | Command1 | { "key": "PATTERN", "default_values": ["blaat"] }    | {PATTERN}
↪:VALUE                                 |
        | Command1 | { "key": "PATTERN", "default_values": ["blaat"] }    | {PATTERN}:
↪{PATTERN}                              |
        | Command1 | { "key": "PATTERN", "default_values": ["blaat"] }    | this-
↪{PATTERN}-key:this-{PATTERN}-value    |
        | Command1 | { "key": "SPA CE", "default_values": ["bla a at"] } | {SPA CE}:
↪{SPA CE}                               |
```

6.3 Custom modules

```
@cmd_args @custom_plugins @custom_plugins_discovery
Feature: Discover custom plugins
    Scenarios for discovering custom plugins at runtime

    Examples:
    | command_line   |
    | --list-plugins |

    Background:
        Given a controlled environment
        And a valid configuration
        And a random custom plugin directory

    @successful
    Scenario: Discover the system modules
        When we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain <plugin_id>
        And stdout should contain regex <description>

        Examples:
        | plugin_id             | description                                                  ␣
↪       |
        | bootstrap             | Lua plugin for module \S*/plugins/bootstrap.lua              ␣
↪       |
        | clang-static-analyzer | Lua plugin for module \S*/plugins/clang-static-
↪analyzer.lua|
        | clang-tidy            | Lua plugin for module \S*/plugins/clang-tidy.lua             ␣
↪       |
        | cmake                 | Lua plugin for module \S*/plugins/cmake.lua                  ␣
↪       |
        | command-line-command  | Command-line-command \(internal\)                            ␣
↪       |
        | cppcheck              | Lua plugin for module \S*/plugins/cppcheck.lua               ␣
↪       |
```

(continues on next page)

(continued from previous page)

```
        | docker                | Lua plugin for module \S*/plugins/docker.lua       |
        | lcov                  | Lua plugin for module \S*/plugins/lcov.lua         |
        | make                  | Lua plugin for module \S*/plugins/make.lua         |
        | ninja                 | Lua plugin for module \S*/plugins/ninja.lua        |
        | pmd                   | Lua plugin for module \S*/plugins/pmd.lua          |
        | scons                 | Lua plugin for module \S*/plugins/scons.lua        |
        | selector              | Lua plugin for module \S*/plugins/selector.lua     |
        | valgrind              | Lua plugin for module \S*/plugins/valgrind.lua     |

    @error
    Scenario: Fail to find a custom module when the search path is not set properly
        Given a custom module with id <plugin_id>
        When we register the command <command> to use the module <plugin_id>
        And we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should not contain <plugin_id>

        Examples:
        | plugin_id                 | command  |
        | exec-helper-custom-module | Command1 |

    @successful
    Scenario: Discover a custom module by setting the search path in the configuration
        Given a custom module with id <plugin_id>
        When we register the command <command> to use the module <plugin_id>
        And add the search path to the configuration
        And we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain <plugin_id>
        And stdout should contain regex <description>

        Examples:
        | plugin_id                 | description                                                  |
                              | command  |
        | exec-helper-custom-module | Lua plugin for module \S*/custom-plugins/exec-
helper-custom-module.lua  | Command1 |

    @successful
    Scenario: The search custom plugin configuration takes precedence over the system
modules
        Given a custom module with id <plugin_id>
        When we register the command <command> to use the module <plugin_id>
        And add the search path to the configuration
        And we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain <plugin_id>
```

(continues on next page)

(continued from previous page)

```
        And stdout should contain regex <description>

        Examples:
        | plugin_id              | description                                                                 |
↪               | command    |
        | make                   | Lua plugin for module \S*/custom-plugins/make.lua                           |
↪               | Command1   |
        | command-line-command   | Lua plugin for module \S*/custom-plugins/command-
↪line-command.lua | Command1   |

    @successful
    Scenario: Discover a custom module by setting the search path on the command line
        Given a custom module with id <plugin_id>
        When we register the command <command> to use the module <plugin_id>
        And add the search path to the command line
        And we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain <plugin_id>
        And stdout should contain regex <description>

        Examples:
        | plugin_id                 | description                                                                 |
↪                         | command  |
        | exec-helper-custom-module | Lua plugin for module \S*/custom-plugins/exec-
↪helper-custom-module.lua  | Command1 |

    @successful
    Scenario: The search custom plugin command line takes precedence over the system
↪modules
        Given a custom module with id <plugin_id>
        When we register the command <command> to use the module <plugin_id>
        And add the search path to the command line
        And we add the <command_line> as command line arguments
        And we call the application
        Then the call should succeed
        And stdout should contain <plugin_id>
        And stdout should contain regex <description>

        Examples:
        | plugin_id              | description                                                                 |
↪               | command    |
        | make                   | Lua plugin for module \S*/custom-plugins/make.lua                           |
↪               | Command1   |
        | command-line-command   | Lua plugin for module \S*/custom-plugins/command-
↪line-command.lua | Command1   |

    @successful
    Scenario: The search custom plugin command line parameter takes precedence over
↪the one(s) in the configuration
        Given a custom module with id <plugin_id>
        And the same custom module <plugin_id> on a different location and add it to
↪the command line search path
        When we register the command <command> to use the module <plugin_id>
        And add the search path to the configuration
        And we add the <command_line> as command line arguments
        And we call the application
```

(continues on next page)

(continued from previous page)

```
        Then the call should succeed
        And stdout should contain <plugin_id>
        And stdout should contain regex <description>

        Examples:
        | plugin_id                  | description
↪                              | command  |
        | exec-helper-custom-module | Lua plugin for module \S*/custom-plugins/other/
↪exec-helper-custom-module.lua  | Command1 |
```

```
@custom_plugins @custom_plugins_usage
Feature: Using custom plugins
    Scenarios for using custom plugins

    Examples:
    | plugin_id                 | command  |
    | exec-helper-custom-module | Command1 |
    | make                      | Command2 |

    Background:
        Given a controlled environment
        And a valid configuration
        And a random custom plugin directory
        And a custom module with id <plugin_id>
        And a registered command <command> that uses the module <plugin_id>
        And the custom plugin search path is registered in the configuration

    @successful
    Scenario: Check that the custom plugin is called
        When run the <command> command <nb_of_times> in the same statement
        Then the call should succeed
        And the <command> command should be called <nb_of_times> times
        And stderr should be empty

        Examples:
        | nb_of_times   |
        |   1           |
        |   10          |
```

6.4 Execution order

```
@execution_order
Feature: Execution order
    The order of execution must be as defined by the exec-helper configuration and
↪specification

    Background:
        Given a controlled environment
        And a valid configuration

    @successful
    Scenario: Run a command with one associated command line a number of times
        When we add the <command> command
```

(continues on next page)

(continued from previous page)

```
        And run the <command> command <nb_of_times> in the same statement
        Then the call should succeed
        And the <command> command should be called <nb_of_times> times
        And stderr should be empty

        Examples:
        | command       | nb_of_times   |
        | some-command  |   1           |
        | other-command |   10          |
```

6.5 Working directory

```
@working_dir @settings_file_location
Feature: All paths in a configuration file are relative to the location of the␣
↪settings file
    Scenarios for checking all paths relative to the settings file

    Examples:
    | command  |
    | Command1 |

    Background:
        Given a controlled environment
        And a valid configuration
        And the <command> command

    @successful
    Scenario: The default working directory is the location of the settings file
        Given a current working directory of <start_working_dir>
        When we run the <command> command
        Then the call should succeed
        And the working directory should be the environment root dir
        And the PWD environment variable should be the environment root dir

        Examples:
        | start_working_dir |
        | /tmp              |
        | .                 |
        | ./blaat           |
        | ./a/b/c/d         |
        | ~                 |
        | /tmp/blaat/       |
```

6.6 Test reports

The Feature test report shows the detailed results of the feature scenario's.

The Unit test coverage report shows the detailed coverage of the unit tests.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

Symbols

E