Technical Documentation of **dune-testtools**

Timo Koch* Dominic Kempf†

June 02, 2015

Contents

$$
\begin{aligned}
< ini > &::= \{< pair > \mid < group >\}^* \\
< group > &::= [\_ < str > \_] \\
< pair > &::= < str > \equiv < str >
\end{aligned}
$$

Figure 1: EBNF describing normal DUNE-style ini files.

1 Introduction

The dune-testtools project is part of a project on quality assurance and reproducibility in numerical software frameworks. It is joint work between the dune-pdelab[1] and the DuMu_x[2] development teams.

2 The Meta Ini Format

The *meta ini* format is used in dune-testtools as a domain specific language for feature modelling. It is an extension to the ini format as used in DUNE. To reiterate the syntax of such ini file, see the EBNF in figure 1 and example 1. Note that, you can define groups of keys either by using the `[..]` syntax, by putting dots into keys, or by using a combination of both.

Listing 1: A normal DUNE-style ini file

```
key = value
somegroup.x = 1
[somegroup]
y = 2
[somegroup.subgroup]
z = 3
```

The meta ini format is an extension to the normal ini file, which describes a set of ini files within one file. You find the EBNF of the extended syntax in 2. The rest of this section is about describing the semantics of the extensions.

2.1 The command syntax

Commands can be applied to key/value pairs by using a pipe and then stating the command name and potential arguments. As you'd expect from

[1] See `https://www.dune-project.org/pdelab/`

[2] See `http://dumux.org/`

$$
\begin{aligned}
< ini > &::= \{< pair > | < group > | < include >\}^* \\
< group > &::= \underline{[} < str > \underline{]} \\
< pair > &::= < str > \underline{=} < value > \{\underline{|} < command >\}^* \\
< value > &::= < str > \{\underline{\{} < value > \underline{\}}\}^* < str > \\
< command > &::= < cmdname > \{< cmdargs >\}^* \\
< include > &::= \underline{include} < str > |\underline{import} < str >
\end{aligned}
$$

Figure 2: EBNF describing the expanded meta ini syntax.

a pipe, you can use multiple commands on single key/value pair. If so, the order of resolution is the following:

- Commands with a command type of higher priority are executed first. The available command types in order of priority are: POST_PARSE, PRE_EXPANSION, POST_EXPANSION, PRE_RESOLUTION, POST_RESOLUTION, PRE_FILTERING, POST_FILTERING, AT_EXPANSION.
- Given multiple commands with the same type, commands are executed from left to right.

2.2 The expand command

The `expand` command is the most important command, as it defines the mechanism to provide sets of ini files. The values of keys that have the expand command are expected to be comma-separated lists. That list is split and the set of configurations is updated to hold the product of all possibile values. Listing 2 shows a simple example which yields 6 ini files.

Listing 2: A simple example of expanded keys

```
key = foo, bar | expand
someother = 1, 2, 3 | expand
```

Sometimes, you may not want to generate the product of possible values, but instead couple multiple key expansions. You can do that by providing an argument to the expand command. All expand commands with the same argument, will be expanded together. Having expand commands with the same argument but a differing number of camma separated values is not well-defined. Listing 3 shows again a minimal example, which yields 2 configurations.

Listing 3: A simple example of expanded keys with argument

```
key = 1, 2 | expand foo
someother = 4, 5 | expand foo
```

The above mechanism can be combined at will. Listing 4 shows an example, which yields 6 ini files.

Listing 4: A simple combining multiple expansions

```
key = foo, bar | expand 1
someother = 1, 2, 3 | expand
bla = 1, 2 | expand 1
```

2.3 Key-dependent values

Whenever values that contain unescaped curly brackets, the string within those curly brackets will be interpreted as a key and will be replaced by the associated value (after expansion). This feature can be used as many times as you wish, even in a nested fashion, as long as no circular dependencies arise. See listing 5 for a complex example of the syntax. In that example one configuration with `y=1` and one with `y=2` would be generated.

Listing 5: A complex example of key-dependent value syntax

```
k = a, ubb | expand
y = {bl{k}}
bla = 1
blubb = 2
```

2.4 Other commands

The following subsections describes all other general purpose commands, that exist in dune-testtools. This does not cover commands that are specific to certain testtools. Those are described in section 4.

2.4.1 The unique command

A key marked with the command `unique` will be made unique throughout the set of generated ini files. This is done by appending a consecutive numbering scheme to those (and only those) values, that appear multiple times in the set. Some special keys like `__name` (see section 3) have the unique command applied automatically.

Using the curly bracket syntax to depend on keys which have the `unique` command applied is not well-defined.

include.ini

```
x = new
include other.ini
y = new
```

other.ini

```
x = old
y = old
```

Result

```
x = old
y = new
```

Figure 3: A minimum example to illustrate the `include` statement

2.4.2 Simple value-altering commands: tolower, toupper, eval

`tolower` is a command turning the given value to lowercase. `toupper` converts to uppercase respectively.

The `eval` command applies a simple expression parsing to the given value. The following operators are recognized: addition (`+`), subtraction (`-`), multiplication (`*`), floating point division (`/`), a power function(`^`) and a unary minus (`-`). Operands may be any literals, `pi` is expanded to its value. See listing 6 for an example.

Listing 6: An example of the eval command

```
radius = 1, 2, 3 | expand
circumference = 2 * {r} * pi | eval
```

Note that the `eval` command is currently within the POST_FILTERING priority group. That means you cannot have other values depend on the result with the curly bracket syntax.

2.5 The include statement

The `include` statement can be used to paste the contents of another inifile into the current ini file. The positioning of the statement within the ini file defines the priority order of keys that appear on both files. All keys prior to the include statements are potentially overriden if they appear in the include. Likewise, all keys after the include will override those from the include file with the same name. See figure 3 for a minimal example.

This command is not formulated as a command, because it does, by definition not operate on a key/value pair. For convenience, `include` and `import` are synonymous w.r.t. to this feature.

2.6 Escaping in meta ini files

Meta ini files contain some special characters. Those are:

- `[ ]` in group declarations
- `=` in key/value pairs
- `{ }` in values for key-dependent resolution
- `|` in values for piping commands
- `,` in comma separated value lists when using the `expand` command

All those character can be escaped with a preceding backslash. It is currently not possible to escape a backslash itself. It is neither possible to use quotes as a mean of escaping instead. Escaping is only necessary when the character would have special meaning (You could in theory have for example commata in keys). Escaping a dot in a groupname is currently not supported, but it would be bad style anyway.

3 System tests with CMake and the Meta Ini Format

CMake[3] is the build system supported by dune-testtools. The CTest system is used to generate test. A CMake interface handles the interplay between meta ini files and the generation of (system) tests. The meta ini syntax can be used without CMake in useful applications as well but shows its power and convenience in combination with the build system. In particular the CMake interface is designed to understand particular meta ini syntax that enables the meta ini file to control the build and test generation process.

Dune-testtool offers tools particularly for system testing. This means that the designed test has to cover a wide range of the software's functionality in contrast to checking a specific feature, i.e. unit testing. There are two different concepts to generate that range in dune-testtools

- Dynamic variations of parameters that are known at runtime
- Static variations of parameters / types / classes that are known at compile time

[3] `http://www.cmake.org/`

Both variations can be specified in the meta ini file. The subsequent sections present how to use the CMake interface in combination with meta ini files to generate system tests.

3.1 A simple test

A simple *ctest* using an ordinaty DUNE-style ini file can be added using the CMake interface as follows

Listing 7: A CMakeLists.txt adding a simple test with the dune-testtools CMake interface function `dune_add_system_test`

```
dune_add_system_test(SOURCE mytest.cc
                     BASENAME mytest
                     INIFILE mytest.ini)
```

Note that for this simple example the macro is equal to writing

Listing 8: A CMakeLists.txt adding a simple test with the standard CMake macros

```
add_executable(mytest mytest.cc)
add_test(NAME mytest
         COMMAND ./mytest mytest.ini)
```

We will see more sensible usage of the macro subsequently.

3.2 Dynamic variations

For creating tests covering different runtime parameters the CMake macro `dune_add_system_test` can be used exactly as in the simple test example above. The ordinary ini file gets replaced with a meta ini file as in Listing 11.

Listing 9: An example of dynamic variations

```
[TimeManager]
timestep = 0.1, 0.5 | expand
endtime = 5, 10 | expand
```

As explained above the meta ini file expands to four ini files with the respective combinations of the parameters `timestep` and `endtime`. Using that meta ini file in combination with the CMake macro call

Listing 10: A CMakeLists.txt generating dynamic variations

```
dune_add_system_test(SOURCE mytest.cc
                     BASENAME mytest
                     INIFILE dynamic.mini)
```

will create four ctests each running the created executable with one of the four generated ini files. Naturally, the source file `mytest.cc` should make use of the given parameters. The standard naming scheme for the ini files will result in `mytest_0000.ini, mytest_0001.ini, ....` For more control over the naming schemes see Section 3.4. The created tests are named as a combination of executable and ini file, hence `mytest_mytest_0000, mytest_mytest_0001, ....`

3.3 Static variations

Creating tests convering different compile time parameters is also straight forward. The meta ini file can be used to generate precompiler variables. A meta ini file creating static variations is shown in Listing **??**.

Listing 11: An example of dynamic variations

```
[__STATIC.COMPILE_DEFINITIONS]
SOLVER = Dune::Solv1, Dune::Solv2<T> | expand
```

This meta ini is parsed for CMake and will generate two different executables for each precompiler variable using the same call as above

Listing 12: A CMakeLists.txt generating static variations with compile definitions

```
dune_add_system_test(SOURCE mytest.cc
                     BASENAME mytest
                     INIFILE static.mini)
```

Not that the special group `[__STATIC.COMPILE_DEFINITIONS]` is reserved for this purpose. The created executables will be called `mytest_0000, mytest_0001, ...` according to the standard naming scheme. For customization please read the next section.

3.4 Naming schemes for ini files

The meta ini syntax offers special keys controlling the naming scheme of ini files and executables.

Listing 13: Meta ini file using the special keys for setting the naming scheme

```
__name = {naming}
__exec_suffix = {__STATIC.COMPILE_DEFINITIONS.SOLVER}
__ini_extension = input
__ini_option_key = --input
timestep = 0.5, 1.0 | expand t
naming = smalltimestep, bigtimestep | expand t
```

```
[__STATIC.COMPILE_DEFINITIONS]
SOLVER = solver1, solver2 | expand
```

The meta ini file in Listing 13 makes use of those special keys. The key `__name` sets the basename for the generated ini files by expansion. If it is unique within the set of inifiles the basename is the name of the ini file, if not a numbered scheme will be added to provide uniqueness. The key `__execsuffix` sets the suffix that will be added to the name of the executable. Again, if not unique a numbered scheme will be added automatically to the suffix. The key `__ini_extension` sets the extension of the created ini files. The default is `*.ini`. The `__ini_option_key` can be used to tell CMake that the ini file is attached to an executable using an option key. In this example the programme will be called with `./mytest_solver1 --input smalltimestep.input`.

3.5 Understanding the CMake macro `dune_add_system_test`

3.6 Creating tests from existing executables

3.7 Creating executables with static variations without tests

4 Testtools

4.1 Fuzzy compare VTK files

4.2 Comparing custom output / The Dune::OutputTree

4.3 Convergence tests

5 Buildbot configuration

6 Handling test results