Archive of Numerical Software
vol. 5, no. 1, 2017, pages 151–168
DOI: 10.11588/ans.2017.1.27447

System testing in scientific numerical software frameworks using the example of Dune

Dominic Kempf[1] and Timo Koch[2]

[1]Interdisciplinary Center for Scientific Computing, Heidelberg University
[2]Institute for Modelling Hydraulic and Environmental Systems, University of Stuttgart

Received: January 31st, 2016; **final revision:** July 16th, 2016; **published:** March 6th, 2017.

Abstract: We present `dune-testtools`, a collection of tools for system testing in scientific software using the example of the Distributed Unified Numerics Environment (Dune). Testing is acknowledged as indispensible support for scientific software development and assurance of software quality to produce trustworthy simulation results. Most of the time, testing in software frameworks developed at research facilities is restricted to either unit testing or simple benchmark programs. However, in a modern numerical software framework, the number of possible feature combinations constituting a program is vast. System testing, meaning testing within a possible end user environment also emulating variability, is necessary to assess software quality and reproducibility of numerical results. We provide an easy-to-use interface taking workload off developers and administrators in open-source scientific numerical software framework projects. In our approach, the large number of possible combinations is reduced using the scientific expert knowledge of developers to identify the practically relevant combinations. Our approach to system testing is designed to be integrated in the workflow of a research software developing scientist.

1 Introduction

Numerical software is nowadays an integral part of research at universities and other research facilities particularly considering natural sciences, engineering, and mathematics. Therefore, the quality of such software directly affects the effectiveness of research and the quality of research findings.

Often however, scientists develop and write research software themselves without advanced knowledge of or little experience in programming. The research may require to modify, develop, and understand the code in depth, ruling out using well-tested commercial software that may already exist. Also, time available for thoroughly testing software quality is limited. The situation may improve for researchers when joining their efforts to develop so-called numerical software frameworks. Frameworks provide features, i.e. software components, commonly used in the research area of interest. Thus, scientists profit from code reuse. What was already written by others does not have to be written again; what is already written can be further improved. Among the larger numerical software frameworks for solving partial differential equations, we mention

DUNE [Bastian et al., 2008, 2011], FEniCS [Logg et al., 2012], and deal.II [Bangerth et al., 2016, 2007] as examples.

As many scientists rely on the same code, software quality becomes increasingly important in such software frameworks. Testing software quality is a vital process when developing and distributing software. Much theory and also practical guidance has been elaborated by the software engineering community. We mention [Myers et al., 2011] and [Burnstein, 2006] as well-written text books on software testing. Understandably, the focus lies on commercial software development models where testing is already an established process and software testers are suggested to be as many as developers themselves [Burnstein, 2006, p. 34]. In contrast, in numerical software framework projects, developing, testing, and research are usually conducted by the same persons. The developer is often focused on producing results rather than making the software reliable, maintainable, and usable for others. This means little time is available for testing. This paper focuses on quality measures and testing tools tailored for the numerical software community in universities and other research facilities.

In Section 2, the authors introduce quality measures important for numerical software frameworks. The challenges distinct to assuring quality in such frameworks are summarized. Section 3 introduces `dune-testtools` as the proposed approach to tackle the difficulties testing numerical software frameworks more thoroughly. The example of DUNE as such a framework is introduced revealing the importance, difficulties, and the current lack of system testing. Section 4 and 5 explain the implementation and user interface of `dune-testtools` in detail. Read Section 6 to know how to get to our code. Finally, Section 7 gives an outlook to the authors' future concerns regarding the continuous integration of testing into the software framework development process at research facilities.

2 Quality and testing quality of numerical software frameworks

Numerical software frameworks differ vastly from software products considered in the software engineering community. One main difference, for example, is the small number of developers in numerical software frameworks that are also tester and scientist in personal union. Frameworks have by purpose a very large variability and combinability of features that makes them hard to test as a whole. We will therefore elaborate on our definition of software quality for numerical frameworks before proceeding with our approach.

2.1 Defining quality for numerical software frameworks

The IEEE Standard Glossary of Software Engineering Terminology [IEEE, 1990] provides a compact definition of software quality as *"[t]he degree to which a system, component, or process meets specified requirements . . . [and] user needs or expectations"*. First, we therefore attempt to define typical quality characteristics expected from numerical software frameworks.

- Correctly implemented code: the code is expected to be implemented correctly with respect to syntax and functionality of features. It includes e.g. that the source code compiles with all compilers satisfying the communicated minimum requirements. Implementation correctness also includes runtime criteria like the absence of memory leaks and race conditions and proper error handling (e.g. through the use of exceptions).
- Numerically correct algorithms: the implemented algorithms should function as intended, namely reproduce results expected from theory. Often, there are well-known mathematical properties of an implemented algorithm that have to be satisfied.
- Trustworthy results: models and simulations built with the numerical software framework should produce trustworthy results, in the sense of correctly modeling what experiments or experts predict. Note that this is not solely a task for the framework but also for the scientist to use it as intended.

- Good documentation, maintainability, testabilty: a high coverage of code documentation facilitates code reusage and also testing.
- Flexibility: in order to have many possibilities of combining features, the implementation and interfaces need to be designed with a certain level of abstraction. This is a key competence of numerical software frameworks.
- Sufficient performance and scalability.

2.2 Testing quality for numerical software frameworks

Testing is the process of determining how well these quality characteristics apply to a numerical software framework. In the scientific context, this process is often formalized in terms of verification and validation, e.g. [Oberkampf et al., 2002, Kelly and Sanders, 2008]. Hook and Kelly [2009] also consider what they call "code scrutinization", i.e. finding faults related to wrongly implemented code, among the testing processes. They further argue that in the scientific context it is usually not possible to determine whether a numerical software is correct because exhaustive testing is impossible due to the large number of possible program realizations. It is rather the goal to determine whether the software is trustworthy.

The process of code scrutinization directly corresponds to assuring correctly implemented code. Code scrutinization is a process early in the development phase and can be supported by complementing features with simple test programs. Verification of numerical code and its algorithms is the process of testing the algorithms for cases where the outcome is well-known. For example, for a simplified PDE and specific boundary and initial conditions there often exists an analytical solution that the numerical solution should converge to when refining the space and time discretization; it is also well-known that a Newton-Raphson scheme converges in one iteration for linear problems. In the examples, the numerical solution and the number of iterations are output parameters of a test and can be automatically checked against the reference values. Validation of a program or model is the process of analyzing the program output with expert knowledge or in comparison with experimental data. The scientist might know that in an experimental setup corresponding to the simulation setup the oil pumping rate lies between 4 and 5 kg/s. If the numerical model suits the experimental setup the code can be validated using the experimental parameter range. This can also be done automatically within a testing framework. The result should not change if the code makes use of different but also suitable discretization scheme or a different linear solver. Code scrutinization, verfication, and validation are processes suited for determining if the program produces trustworthy results. All three processes are outcome motivated, in the sense that they only evaluate results of a simulation program.

However, in the context of numerical frameworks, reusability, stability, and combinability play a key role assuring software quality. Testing variability and combinability can be viewed as the repetition of the above introduced processes for many possible realizations of programs and many possbile user configurations of those programs using features of a numerical software framework. Again, exhaustive testing is practically impossible.

2.3 Levels of testing

Burnstein [2006] decribes a need to categorize testing in four levels — unit testing, integration testing, system testing, and acceptance testing. Unit testing tests "*a single component*" [Burnstein, 2006, p. 133]. Unit testing is often available for numerical software frameworks and is good practice to write unit tests alongside every new feature.

Integration testing combines several components into working units. Integration testing can rarely be found in numerical software frameworks. If a certain interface is of great importance in a framework, sometimes interface checks are offered qualifying as integration testing. As the modules in a modular framework are considered mostly independent, intermodular integration testing is often not sensible.

System testing, testing the combinations of features from several modules in a working numerical program, tests a fully working end user scenario in contrast to integration testing. System testing is not commonly employed in numerical software frameworks to the authors' knowledge. An exception are so-called tutorial, demo, or test programs testing specific simple setups with an educational intention. Sometimes framework include a suite of benchmarks which qualify as tests in the sense of verification and validation but generally do not test variability and feature combination. However, the vast combinatoric possibilities of a numerical framework require more comprehensive testing of module interdependencies in order to assure the reliablilty and generality of the framework. We will therefore have a focus on system testing subsequently.

Finally, acceptance testing is "*formal testing conducted to enable a user . . . to determine whether to accept a system or component*", according to the IEEE Standard Glossary of Software Engineering Terminology [IEEE, 1990]. In numerical software frameworks with an open-source development model, acceptance testing is constantly ongoing through the user community and feedback is commonly issued via mailing lists and issue trackers.

2.4 Regression testing

The code basis in open source numerical software frameworks is constantly under development and improvement. Those changes can be tracked by the testing suite by means of so-called regression testing. Burnstein defines that "*regression testing is . . . the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes.*" [Burnstein, 2006, p. 176]. For regression testing, the output of a numerical code is compared to a reference output created when the software was considered correct. Regression tests are possible on all four levels. Particularly, performing regression tests on a system test level has the ability to reveal if code changes might be an improvement in one configuration while being harmful for another.

2.5 System testing

System testing can be considered the most important testing level for a numerical framework, as the framework by design should allow arbitrary combination of modular blocks. In unit and integration testing, developers usually write tests covering their own code. In system testing however, code maintained by multiple developers is combined, revealing the bugs and corner cases the upstream developers have not thought of.

One approach to system testing within the software engineering community is to formalize software variability through Software Product Line Engineering (SPLE) [Pohl et al., 2005]. In SPLE, a *variability model* descibing thoroughly all possible configurations of the software is set up. Configurations usually vary in characteristic properties called *variation points*. Possible values of a variation point are called *variants*. A variability model for a complex software usually features a complex structure of constraints between variants.

Using SPLE in the context of a numerical framework for PDEs (Dune) has already been investigated by Remmel et al. [2011] and Remmel [2014]. She states some of the fundamental inherent problems with SPLE for scientific frameworks: Determining a variability model for all the scientific applications is not possible due to the unlimited variability the mathematical input provides. Instead, SPLE is applied to model variability within the framework for a fixed mathematical model. This limited variability model and its constraints can only be determined with the knowledge of the developer.

3 `dune-testtools` – A practical approach to integration and system testing in numerical software frameworks

This section will describe our approach to system testing of a numerical software framework. As our development is driven by the Dune project as an example framework, we briefly introduce those properties of Dune that influenced the design process.

Dune, the Distributed Unified Numerics Environment, is an open source modular numerical software framework developed at more than 10 universities in Europe [Bastian et al., 2011]. It has a highly modular structure where currently five modules are *core modules* and as such mandatory for most applications. Built upon these core modules exist *discretization modules* like `dune-pdelab` (fast prototyping of new discretizations [Bastian et al., 2010]) and domain-specific application frameworks like DuMu^x (flow and transport processes in porous media [Flemisch et al., 2011]). Users will typically write their codes in form of *application modules* depending on the above. All those modules may further depend on external and third-party libraries. The modular structure of Dune is managed by a CMake-based buildsystem. One of the design principles of Dune is the definition of stable interfaces for common simulation components. The most famous example of this generic approach is the grid interface in `dune-grid` that allows to easily switch the underlying grid implementation of a code. Such a generic programming approach offers great opportunities for easily setting up system tests.

Dune's open source workflow includes a group of core developers that have direct write access to the core repositories. Non-core developers contribute through merge requests. All users report issues through issue trackers and mailing lists. Most participants including the core developers develop Dune next to their scientific research activity. As a consequence, the resources available for maintenance of the software are limited. Extensive testing has proven to suffer badly from this lack of resources in the past. To the authors' knowledge, the majority of Dune developers work on Unix-like systems in a commandline-centered fashion and there is little to no usage of IDEs in the Dune community.

From the above description of the Dune project, the authors conclude the following requirements for a sustainable testing environment for modular software frameworks.

- Any solution for testing needs to integrate well into the existing modular structure of Dune.
- To lower the entry barrier to a minimum, new workflows should integrate well with existing ones (no additional tools, IDEs etc.).
- Writing a new system test should be an easy and quick procedure.

As pointed out by Remmel et al. [2011] in the context of SPLE, the variability of framework-based scientific applications is twofold. On the one hand, there is the variability in the mathematical model serving as input data to the framework. On the other hand, given the output of the framework for a fixed mathematical model (an executable), there is variability in the configuration of the simulation run. See Figure 1 for an illustration.

The variability model for all scientific applications is too large to determine due to the infinite nature of the mathematical variation points. Only considering the variability model for a fixed mathematical input model also bears problems, as the variability model will still become very large. This results from the fact that the number of variation points for scientific software such as Dune is high and the variability model is the combinatoric product of these variants. Therefore, a strategy to reduce the amount of test cases for system testing while assuring sufficiently high coverage is needed.

We base our approach on the idea, that the entire Dune user base needs to be involved in the quality assurance process. This is important for the following reasons.

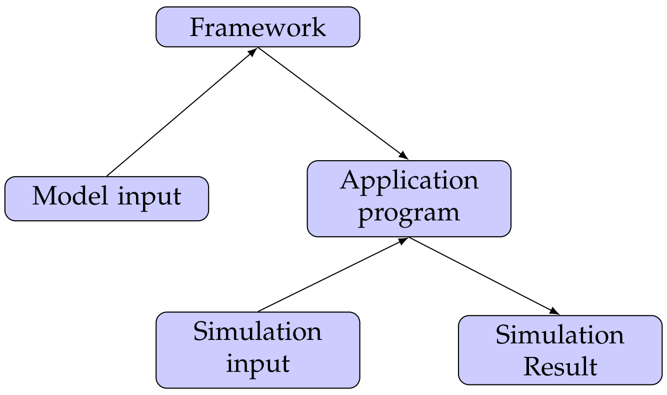

Figure 1: A simulation is set up in two stages. Firstly, a mathematical model is selected and the necessary features for implementing this model are chosen from the features offered by the numerical framework. Secondly, the application program is configured by the simulation input being compile time or runtime parameters. Both stages are objected to a large variability.

- The entirety of system tests should cover all the application areas of DUNE. Those are however defined by the users putting efforts into applying DUNE to new areas. In other words, the only sensible sampling of the variability model that includes the mathematical models is to use those samples that DUNE users are actively working on.
- A system test is best written by the expert in the field, who is the user working on the exact problem. Also, defining a reasonably large sample of the variability model for a given model is only possible with good scientific knowledge of the problem. For instance, the choice of linear solver and/or preconditioner for a PDE problem cannot be done a priori.
- Reusing user code for system tests can result in a mutual benefit for user and framework. On the one hand, the user contributes samples to the framework's test suite that increase the coverage of the variability model. On the other hand, the user gets an automated testing workflow for his or her code, increasing the quality and reproducibility of the code and the scientific work and results.

To sum it up: instead of providing a full variability model, we carefully choose a set of system tests that try to cover as many use cases of the DUNE project as possible. For a given system test, we again do not provide the full variability of the framework, but instead go for samples carefully selected by the user. These samples should cover all of the configurations relevant to the problem. The system tests in their entirety should aim for a full coverage of relevant variant combinations of the framework.

Considering tools for test evaluation, we heavily rely on regression testing. In a steadily changing code base, it is of particular importance to track regressions in a system testing context. We store reference files, created when the code was considered correct, i.e. usually around the time of a new version release, under version control. We mention the recently introduced large file storage concept of Git [Git-lfs webpage] as solution to the problem of storing large files under version control. Keeping reference files over a longer period of time eliminates the chance for slow regression and reduces the testing effort in comparison to using ad-hoc before and after comparisons whenever the code base is changed.

We provide an implementation of the above presented concepts in the form of the DUNE module `dune-testtools`. Having the implementation as a DUNE module integrates it easily into the modular structure of the DUNE project. `dune-testtools` only provides means to *define* system tests independently of the system the test will be run on. `dune-testtools` contains a Python package `dune.testtools` which does the heavy lifting of the implementation. Furthermore, it features CMake code for the buildsystem integration and some C++ helper classes. The following Sections 4 and 5 elaborate on how `dune-testtools` implements feature modeling and test evaluation, respectively.

4 Feature modeling in `dune-testtools`

As mentioned previously, our approach relies heavily on the application developers to define the variability model for their use cases. To reduce entry barriers to writing system tests to a minimum, we try to integrate our approach as deeply as possible into the existing workflow of Dune developers. As a consequence, we completely avoid external tools, but instead work with what we identified as the integral parts of a simulation code:

- A C++ source file that runs the simulation,
- an ini file (i.e. a parameter file) that allows to configure the simulation run, and
- little CMake code to define executable and tests.

The C++ source file is only to a limited extent suitable for modelling variability. Code will get duplicated too often. It will look cluttered and as a consequence it will not be maintainable. However, we consider maintainability one of the crucial factors for the success of the quality assurance process. We therefore decided to use the ini file for modelling variability by extending its syntax to what we call *meta ini files*. Instead of just one configuration, meta ini files describe a set of combinations and therefore serve as a domain specific language for modelling the variability model for our system tests. Meta ini files are expanded into a set of normal ini files in a preprocessing step.

4.1 Extending ini files

Normal Dune-style ini files contain key/value pairs, which are separated by an equal sign (=). Keys can be grouped by sections which may either be defined explicitly by bracing its name with square brackets, or implicitly by using dots in key names. Sections may be nested to arbitrary depth.

In meta ini files, we introduce commands which can be applied to key/value pairs with a Unix-style pipe (|). The most important command is the `expand` command, as it provides the mechanism to describe sets of configurations. The values of keys that have the `expand` command applied are expected to be comma-separated lists. Those lists are split and the list of configurations is updated to hold the product of all possible values. The following example yields a total of 6 configurations describing all the combinations of values for `theta` and `timestep`.

Meta ini Code

```
theta = 0.5, 1.0 | expand
timestep = 0.1, 1.0, 10.0 | 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. The expansion process when having `expand` commands with the same argument but a differing number of comma separated values is semantically not well-defined and therefore produces an error. The next example yields 2 configurations and does a refinement in space and time simultaneously.

Meta ini Code

```
gridlevel = 1, 2 | expand refinement
timestep = 1.0, 0.5 | expand refinement
```

Expansion with and without argument may be combined arbitrarily. The following example combines the two above examples for a total of 4 configurations.

Meta ini Code

```
theta = 0.5, 1.0 | expand
gridlevel = 1, 2 | expand refinement
timestep = 1.0, 0.5 | expand refinement
```

Sometimes, you will want to define a key's value depending on the value of another key, where that other key has the `expand` command applied. You may do so by writing the keyname in curly brackets into the values. See the following example, where an output naming scheme is controlled through this mechanism.

Meta ini Code

```
dimension = 2, 3 | expand
boundarycondition = dirichlet, neumann | expand
outputfile = {boundarycondition}_{dimension}d.output
```

The curly bracket syntax may even be used in a nested way and provides a powerful tool for the meta ini language.

With the `expand` command as shown above, a set of configurations needs to have a product structure in order to be described by a meta ini file. To overcome this limitation, the `exclude` command has been introduced. It allows to define a boolean condition using curly brackets. If that condition evaluates to `True` in Python, the entire configuration is removed from the set of configurations. Because the `exclude` command does not operate on a key/value pair by definition, it is instead applied to the condition directly in a separate line of the ini file. In the following example, the grid levels that are not suitable for three-dimensional simulations are discarded. It yields a total of 8 configurations, 5 of which are for the two-dimensional and 3 of which are for the three-dimensional case.

Meta ini Code

```
dimension = 2, 3 | expand
gridlevel = 1, 2, 3, 4, 5 | expand
{dimension} == 3 and {gridlevel} > 3 | exclude
```

Once the set of generated configurations starts to grow, it is helpful to control the naming scheme of the generated ini files. We use the special key __name for that. You may define it according to your needs using the curly bracket syntax. If your definition does not result in unique names, consecutive numbering will be appended to your naming scheme. The __name key is optional and not defining it will result in consecutively numbered ini files.

So far, we have only considered the `expand` and the `exclude` command. There are some more commands implemented which we only mention briefly here and refer to the documentation of `dune-testtools` for further reading.

- `unique` makes the values to the given key unique throughout the set of configurations, useful for unique names like output filenames. The __name mechanism uses this command internally.
- `eval` allows to perform simple arithmetic operations on keys
- `tolower` and `toupper` transform strings to their lower-, uppercase representations, respectively.

Note that the meta ini syntax uses some characters for defining its own syntax. These are

- `{` and `}` in values for key-dependent resolution,

- `|` in values for piping commands,
- `,` in comma separated value lists when using the `expand` command.

If you want to use those characters as part of a value, you need to escape either by a preceding backslash or by embracing quotes.

4.2 Static and dynamic variations

So far, we have introduced meta ini files as a way of expressing the variability model for simulation codes. Each variation point is represented by a key which has the `expand` command applied. However, one of the biggest challenges with the variability model of simulation environments such as Dune is the fact that many common variation points deal with compile-time variants. Unfortunately, not all variation points describe static variations and treating dynamic variations as static ones will result in a waste of resources. We will therefore describe in this section, how to extend the concept of meta ini files by the ability to describe static and dynamic variations in a mixed fashion.

We introduce a special group in the meta ini file called `__static`. Any key/value pairs in the `__static` section are to be passed as C preprocessor variables to the simulation executable. To that end, the expansion process and the buildsystem need to be interleaved. We will not cover the details of this interface from CMake to Python here. Instead, we describe the CMake API from `dune-testtools` in detail in Section 4.3. Here, we restrict ourselves to describe how to add static variations to meta ini files. See the following minimal example that defines the grid type as a variation point.

Meta ini Code

```
[__static]
GRIDTYPE = Dune::YaspGrid<2>, Dune::UGGrid<2> | expand
```

When used with the macros described in Section 4.3, CMake will parse this meta ini file at configure time and generate executables from the static section. In analogy to the `__name` mechanism, you can control the naming scheme of the executables by defining the key `__exec_suffix` with the curly bracket syntax. The suffix will be made unique and be appended to the target basename given to CMake.

To correctly set up executables and tests for a system test with both static and dynamic variations, CMake needs to determine the correct number of static variants and the correct number of dynamic variations per executable. While the first one is easily done by filtering the `__static` section from the set of configurations, the latter bears some problems. We implicitly implied until now that any non-`__static` key defined a dynamic variation. However, when writing more sophisticated meta ini files, you might want to define some auxiliary keys not defining dynamic variations. Take the following advanced example where auxiliary keys are used to define a polynomial degree depending on the grid.

Meta ini Code

```
grid = yasp, ug | expand grid
deg_yasp = 2
deg_ug = 2, 3 | expand
[__static]
GRIDTYPE = Dune::YaspGrid<2>, Dune::UGGrid<2> | expand grid
DEGREE = {deg_{grid}}
```

The example would produce a total of four test from two executables. Two of these tests would effectively be equivalent as they are using `Dune::YaspGrid<2>` as their grid type and only differ in the unused value `deg_ug`. To overcome this limitation, we have to mark the keys `deg_yasp` and `deg_ug` as auxiliary keys by putting them into the `__local` group. The following corrected example produces the desired amount of three tests.

Meta ini Code

```
grid = yasp, ug | expand grid
[__local]
deg_yasp = 2
deg_ug = 2, 3 | expand
[__static]
GRIDTYPE = Dune::YaspGrid<2>, Dune::UGGrid<2> | expand grid
DEGREE = {__local.deg_{grid}}
```

The keys in the `__local` group are expanded and they are available for curly bracket syntax resolution. However, at the end of the expansion process, the `__local` section is removed from the generated configurations and the set of configurations is shrunk by eliminating duplicate configurations.

Using the meta ini mechanism to have CMake automatically add targets bears another problem if you are used to guard the addition of executables in CMake with `if`-clauses that depend on configure checks for external packages. Consider the above example in a situation, where the external dependency UG was not found on the system. In that case, the executable should not even be added in the buildsystem. Such guards need to be defined in the meta ini file through the command `cmake_guard`, which similar to the `exclude` command does not operate on a key/value pair but on a single value. This value is evaluated in CMake at configure time and the executable is not added, if it evaluated to `FALSE`. The above example can be fixed as follows. `TRUE` is used as a non-operational guard variable here, as `Dune::YaspGrid` is always available.

Meta ini Code

```
[__static]
GRID = Dune::YaspGrid<2>, Dune::UGGrid<2> | expand grid
TRUE, HAVE_UG | expand grid | cmake_guard
```

CMake provides a very useful mechanism to attach metadata to tests in the form of so called labels. Tests can have an arbitrary amount of labels attached and labels do not have any semantics by themselves. The testing driver `ctest` can be used with regular expressions on test labels. Being able to define labels in meta ini files allows us to define different priority groups. This can be done through the command `label` which, as `exclude`, operates on a Python Boolean expression. It takes the label to be applied as the first argument. Similar to the `expand` command, you may provide an arbitrary identifier as the second parameter. Labels with the same identifier will override previously defined labels with the same identifier. Consider the following example where all tests have the `WEEKLY` label applied, but one labeled `NIGHTLY`. Note how `True` is used to set the default label `WEEKLY` for all configurations.

Meta ini Code

```
model = 1, 2, 3, 4, 5 | expand
True | label WEEKLY PRIORITY
{model} == 3 | label NIGHTLY PRIORITY
```

Writing system tests with `dune-testtools`, you might experience that a generic coding style within the C++ source file is very beneficial. You will often define types for simulation components in the meta ini file. If all possible variants for a given static variation point fulfill the same interface or concept, you can do that with a single codepath and thus achieve maintainability. We have identified this need and try to provide tools for the most common tasks either through `dune-testtools` or through contribution to the DUNE core modules. This currently includes

- the `IniGridFactory` from `dune-testtools`, which allows to create grids from a section in an ini file. It provides specializations for the most important DUNE grids.
- The ongoing project [dune-istl GitLab] of providing fully dynamic, ini file-based solver and preconditioner setup in `dune-istl`.

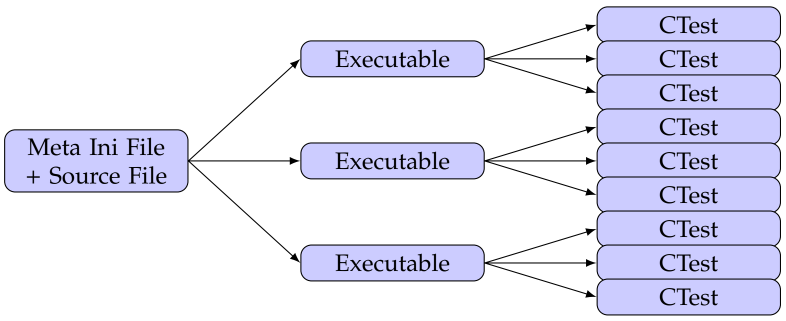

Figure 2: Expansion procedure for the macro `dune_add_system_test`

4.3 Incorporation into the build system

One of the core principles of `dune-testtools` is to provide a powerful buildsystem API to the user, not requiring any specific CMake knowledge on the user side. The interface to add a system test with both static and dynamic variations is the following CMake macro.

CMake Code

```cmake
dune_add_system_test(SOURCE src
                     BASENAME base
                     INIFILE ini
                     TARGETS output
                     [SCRIPT script])
```

Writing such a block into a `CMakeLists.txt` file will trigger the following procedure, which is illustrated by Figure 2.

- The given `INIFILE` will be expanded into the current build directory.
- CMake will add executables that are built from the given `SOURCE` with a naming scheme based on the given `BASENAME` with appended `__exec_suffix`. A list of the generated executables is exported to the variable given to `TARGETS`. You may use this to alter targets by adding flags etc. We recommend using the `dune_enable_all_packages` feature from `dune-common` though, to minimize problems arising from the necessity to add specific flags to the generated executables.
- Tests will be added to the `ctest` testing suite for all dynamic variants. The executables are expected to take an ini file as their first and only command line argument.

The `SCRIPT` parameter will be explained in detail later on, as it allows you to attach testing tools to your system tests.

`dune-testtools` also provides CMake macros to use meta ini files outside of the context of testing. These can be useful for setting up reproducible numerical studies. We do not describe these here, but instead refer to the buildsystem documentation of `dune-testtools`.

5 Verification and validation with `dune-testtools`

A test of a numerical code will usually run a small sample problem and return some data. The test suite then needs to determine whether the test passed or failed. The criterium for success of a test is directly related to quality measures the test is supposed to verify. To avoid rewriting the testing code in all tests' executables, `dune-testtools` implements testing methods as wrapper scripts. Wrapper scripts in `dune-testtools` are Python scripts controlling pre-processing, execution of the program, and post-processing. The wrapper scripts decide whether a test passed or not. It supplies useful information to the developer in case it did not. The wrapper scripts are meant to be customized writing few lines of Python code, although `dune-testtools` already provides many useful wrappers. The wrappers are configured by the meta ini file in the section `wrapper.<wrapperscriptname>`.

The wrapper script for a system test is defined through the above introduced CMake macro `dune_add_system_test` by setting the SCRIPT parameter. As of the writing of this article, `dune-testtools` contained the wrapper scripts `dune_execute.py`, `dune_execute_parallel.py`, `dune_outputtreecompare.py`, `dune_vtkcompare.py`, `dune_convergencetest.py`. These wrappers are described in more detail in this section.

5.1 Simple execution wrappers

The default wrapper `dune_execute.py` is used whenever the SCRIPT parameter is omitted in CMake. It runs the executable and forwards the return code.

The wrapper `dune_execute_parallel.py` runs the executable in parallel and forwards the return code. The wrapper would be defined in the CMake macro as follows.

CMake Code

```
dune_add_system_test(...
                     SCRIPT dune_execute_parallel.py)
```

The number of processors for the parallel run can then be specified in the meta ini file.

Meta ini Code

```
[wrapper.execute_parallel]
numprocessors = 4
```

The MPI implementation is automatically detected by the build system and information on it is passed to the wrapper script.

5.2 Regression tests

Assume we have a reference program output produced when the state of the program was considered correct. Regression testing compares the output of the software's current state to this reference output, as introduced in Section 2.4. A very common output of numerical simulation is a computational result in a VTU format [VTK] file, e.g. a discrete solution on an unstructured grid. `dune-testtools` provides the wrapper script `dune_vtkcompare.py` for regression testing VTU output.

Technical difficulties comparing such unstructured grid files include that the data is commonly given in single precision floating point numbers which need to be compared with a certain tolerance value (*fuzzy comparison*). A further problem inherent to floating point numbers is the comparison of numbers that are physically zero but suffer from numerical noise. Finally, different grid managers might use different ordering of vertices or cells.

The wrapper script `dune_vtkcompare.py` offers solutions to tackle these problems. To this end, before comparing the data, both files are sorted by ascending vertex coordinates. Then, different sets of parameters are sorted by their name attribute, because the parameter order in VTU files is mutable. Floating point numbers are first compared by means of an absolute comparison, i.e. two single precision floating point numbers a and b are considered equal if the absolute value of their difference is smaller than the machine epsilon scaled with the magnitude of the parameter, i.e. if $|a-b| <= \epsilon \cdot \max(\mathcal{P})$. Herein, $\mathcal{P}$ denotes the set of absolute parameter values given in the VTU file. The machine epsilon for IEEE 754 single precision floating point numbers is $\epsilon \approx 1.19 \cdot 10^{-7}$, according to the definition of Goldberg [1991].

If a and b are not equal, the script relaxes the criterium and compares relatively. The two floating point numbers are considered being close enough if $|a - b| <= \delta \cdot \max(|a|, |b|)$, where δ is the threshold for the largest acceptable deviation. The relative criterium is necessary as a numerical code introduces rounding errors for every floating point computation. The errors can easily sum up to relatively large errors.

If a data set is physically zero, e.g. the z-component of the velocity in a simulation was fixed to zero but the actual number is subjected to computations with rounding errors, the result can be an array of numbers very close to zero considered as numerical noise. In a case like that, two numbers $v_{z,1} = 1.5 \cdot 10^{-21}$ and $v_{z,1} = 1.2 \cdot 10^{-18}$ are considered zero but will fail both the absolute and relative criterium. For these cases the scripts provides a zero threshold per parameter under which a parameter will be considered zero and thus be exempted from comparison.

Following the above mentioned concepts, the wrapper script `dune_vtkcompare.py` can be configured via the meta ini file as follows.

Meta ini Code

```
[wrapper.vtkcompare]
name = my_vtkfile
reference = path_to_reference_file
extension = vtu
relative = 1e-2
absolute = 1.2e-7 # set to machine epsilon of used floating point format
zeroThreshold.velocity = 1e-18
```

`name` and `reference` are mandatory parameters denoting the file to be compared and the reference file, respectively. The path `name` is relative to the build directory of the test and the path `reference` is relative to the source directory. `extension` specifies the file extensions of the files to be compared and defaults to `vtu`. The parameters `relative` and `absolute` define the thresholds for relative and absolute floating point comparison. Note that the absolute threshold will be scaled to the magnitude of the data to compare. They default to $\delta = 10^{-2}$ and $\epsilon = 1.2 \cdot 10^{-7}$, respectively. The zero threshold can be set for every parameter in the VTU file by name (`velocity` in the above example).

The wrapper script `dune_outputtreecompare.py` does regression testing with arbitrary user data, where the ini file format is used for the output. `dune-testtools` provides a C++ class `Dune::OutputTree` that facilitates writing data to ini files. It inherits from `dune-common`'s `Dune::ParameterTree` and adds some convenience methods for writing out data. The following code shows how to instantiate a `Dune::OutputTree` and fill it with data.

C++ code

```cpp
Dune::OutputTree outputTree("out.ini");
// execute code yielding a variable numIterations
outputTree.set("Iterations", numIterations);
```

Upon destruction, the contents of the `Dune::OutputTree` will be automatically dumped to a file. The wrapper script comparing ini files can be configured as follows via the test's meta ini file.

Meta ini Code

```
[wrapper.outputtreecompare]
name = myoutputfile
reference = path_to_reference_file
extension = out
type = fuzzy
exclude = DebugInfo
relative = 1e-2
absolute = 1.5e-7
zeroThreshold.norm = 1e-18
```

Most options work as explained above for the VTU file comparison. The parameter `type` specifies whether the files are compared `exact` (not recommended when floating point numbers are involved) or a `fuzzy` comparison is preferred. When comparing fuzzy, every value in the ini file that is convertible to a floating point number will be compared as such using the absolute and relative thresholds specified. A key only written for e.g. debugging purposes that is not required to equal the reference file can be excluded by specifying the `exclude` key that takes a space separated list as value. Multiple such output data can be compared in a single test. For configuring the comparison differently for multiple files the following scheme applies.

Meta ini Code

```
[wrapper.outputtreecompare]
name = myoutputfile1 myoutputfile2
reference = path_to_reference_file1 path_to_reference_file2
extension = out out

[wrapper.outputtreecompare.myoutputfile1]
type = exact
exclude = DebugInfo

[wrapper.outputtreecompare.myoutputfile2]
type = fuzzy
zeroThreshold.norm = 1e-18
```

5.3 More involved test wrappers

Several quality measures of numerical software can only be tested with more involved tests often requiring several runs of the same executable. A before mentioned method to verify an algorithm for e.g. a discretization scheme solving a PDE is a convergence study. The grid is refined several times and a rate of convergence can be computed. The rate is inherent to the scheme and should not change when the code base changes. `dune_convergencetest.py` implements a test wrapper for convergence tests. It does require the program to compute an error, e.g. with respect to a known analytical solution. The key in the meta ini file that defines the samples for the convergence test has to marked with the `convergencetest` command, which is a special form of the `expand` command.

Meta ini Code

```
[grid]
refinement = 0, 1, 2, 3, 4 | convergencetest
```

The above example defines a test conducting a grid convergence study. The convergence test wrapper can be configured via the same meta ini file.

Meta ini Code

```
[wrapper.convergencetest]
expectedrate = 2.0
absolutedifference = 0.1
```

The test runs the executable once for every level of refinement. It then computes the convergence rate automatically and compares it to the expected one. In the above example the test would fail if the computed rate differs by more than 0.1 from 2. To compute the rates correctly we make some assumptions on how the data is dumped by the program. The following dummy code snippet taken from `dune-testtools` outlines such a test using Dune.

C++ code

```
#include <dune/testtools/outputtree.hh>
#include <dune/common/parametertreeparser.hh>
#include <dune/common/parametertree.hh>

int main(int argc, char** argv)
```

```
 6 {
 7     // read the given ini file
 8     Dune::ParameterTree params;
 9     Dune::ParameterTreeParser::readINITree(argv[1], params);
10 
11     // get refinement level
12     auto refinement = params.get<int>("grid.refinement");
13     // ... and refine the grid accordingly
14 
15     // construct an output tree with the ini tree
16     Dune::OutputTree outputTree(params);
17 
18     // do simulation and compute an error norm and hmax
19 
20     // output convergence data
21     outputTree.setConvergenceData(norm, hmax);
22     return 0;
23 }
```

Lines 7-9 read the expanded ini file associated with the run into a `Dune::ParameterTree` object. A `Dune::OutputTree` can be constructed from the input ini file to avoid specifying its filename redundantly. Line 21 shows how to use a convenience method of the `Dune::OutputTree` class to easily dump a computed error norm and the global maximum grid element diameter to file.

Other more involved testing methods are e.g. scalability tests where a program is run with different numbers of processors and a speed-up per additional processor can be observed. Performance tests measure the runtime of a certain program. Code changes should usually not increase runtime (this might be a necessary evil of another advantage), whereas a decreased runtime is considered an improvement. The latter tests are architecture dependent and can only be executed in comparison to a reference state obtained on the same machine. A possible scenario would be to run such tests before and after a change to the code base. Such tests can be automated when using appropriate version control systems. Scalability and performance testing will appear in future releases of `dune-testtools`.

5.4 Writing new test wrappers

The existing test wrappers can be customized easily with little knowledge of Python. For example, the parallel wrapper could be combined with the wrapper comparing output data, to test if the output from a parallel program run is equal to that of a sequential execution. If you want to write a wrapper from scratch, check the Python module `dune.testtools.wrapper.argumentparser` for a list of information that CMake forwards to the wrapper scripts. Following `dune-testtools`'s style of configuring test wrappers avoids confusion.

6 Code Availability and Portability

The code presented in this paper is available freely under a BSD license. You find an overview of the quality assurance related projects of the authors under

https://gitlab.dune-project.org/groups/quality

Note, that `dune-testtools` also requires the module `dune-python`, which is also available under the same URL. `dune-python` is a buildsystem extension to `dune-common` that unifies the ways of distribution and installation of DUNE modules and Python packages. The centerpiece of `dune-python` is a Python virtualenv, a tool setting up a system-independent environment for running Python code. It is automatically set up at configure time in the build directory. `dune-python` is designed to be used by any other DUNE module distributing Python code.

As can be expected from a project on quality assurance, we have taken measures to ensure the quality of our code. Extensive unit testing is done for Python and C++ code and as well for

CMake code through configure time unit tests. We do provide comprehensive documentation, which can be built by typing `make doc` in the toplevel build directory. In `dune-testtools`, you find the result in the (build directory) subdirectory `doc/sphinx/html`.

The methods presented in this paper are - though tailored for it - not limited to usage with the Dune project. The Python package `dune.testtools`, which contains the code for both meta ini expansion and testing tools is completely general and does not contain any Dune specifics. The interface between the Python package and the projects buildsystem, however, needs to be rewritten in order to use it with a different numerical code.

7 Outlook – an open-source workflow for automated system testing in numerical software frameworks developed at research facilities

The module `dune-testtools` presented in this paper is only a first step towards an automated system testing workflow. It provides convenient, buildsystem integrated tools to define low dimensional samples of the high dimensional variability model of simulation codes based on Dune. Furthermore, it provides tools to verify and validate results for the numerical solution of partial differential equations specifically. All these tools focus on the definition of tests independent of the environment that the tests are to be run in. The system tests defined with `dune-testtools` may be run on any local machine by running `ctest` in the build directory.

In future work we will target the infrastructure for automating the execution of system tests. This includes the following tasks:

- A buildbot [Buildbot webpage] based build server setup
- Using docker [Docker webpage] containers to model heterogeneous userlands
- Integrating buildbot into a merge-request based development model

8 Acknowledgments

The authors would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/2) at the University of Stuttgart, as well as the Ministry of Science, Research and the Arts, Baden-Württemberg, Germany.

References

W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a General Purpose Object Oriented Finite Element Library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007. doi: 10.1145/1268776.1268779. URL `http://dx.doi.org/10.1145/1268776.1268779`.

W. Bangerth, D. Davydov, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and D. Wells. The `deal.II` library, version 8.4. *Journal of Numerical Mathematics*, 24, 2016. doi: 10.1515/jnma-2016-1045. URL `http://dx.doi.org/10.1515/jnma-2016-1045`.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2–3):103–119, 2008. doi: 10.1007/s00607-008-0003-x. URL `http://dx.doi.org/10.1007/s00607-008-0003-x`.

P. Bastian, F. Heimann, and S. Marnach. Generic implementation of finite element methods in the distributed and unified numerics environment (dune). *Kybernetika*, 46(2):294–315, 2010. URL `http://www.kybernetika.cz/content/2010/2/294/paper.pdf`.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, J. Fahlke, C. Gersbacher, C. Gräser, C. Grüninger, D. Kempf, R. Klöfkorn, S. Müthing, M. Nolte, M. Ohlberger, and O. Sander. Dune Webpage, 2011. http://www.dune-project.org.

Buildbot webpage. http://buildbot.net.

I. Burnstein. *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.

Docker webpage. https://www.docker.com/.

dune-istl GitLab. The project is developed on the branch `feature/istl-ini` in the `dune-istl` repository. See the merge request https://gitlab.dune-project.org/core/dune-istl/merge_requests/5 for an overview what is happening. [Accessed: February 2017].

B. Flemisch, M. Darcis, K. Erbertseder, B. Faigle, A. Lauser, K. Mosthaf, S. Müthing, P. Nuske, A. Tatomir, M. Wolff, et al. DuMu^x: Dune for multi-{phase, component, scale, physics,...} flow and transport in porous media. *Advances in Water Resources*, 34(9):1102–1112, 2011. doi: 10.1016/j.advwatres.2011.03.007. URL http://dx.doi.org/10.1016/j.advwatres.2011.03.007.

Git-lfs webpage. https://git-lfs.github.com/.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991. doi: 10.1145/103162.103163. URL http://dx.doi.org/10.1145/103162.103163.

D. Hook and D. Kelly. Testing for trustworthiness in scientific software. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 59–64, May 2009. doi: 10.1109/SECSE.2009.5069163. URL http://dx.doi.org/10.1109/SECSE.2009.5069163.

IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. doi: 10.1109/IEEESTD.1990.101064. URL http://dx.doi.org/10.1109/IEEESTD.1990.101064.

D. Kelly and R. Sanders. Assessing the quality of scientific software. In *First International Workshop on Software Engineering for Computational Science and Engineering*, 2008. URL http://secse08.cs.ua.edu/Papers/Kelly.pdf.

A. Logg, K.-A. Mardal, and G. Wells. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8. URL http://dx.doi.org/10.1007/978-3-642-23099-8.

G.J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

W. Oberkampf, T. Trucano, and C. Hirsch. Verification, validation and predictive capability in computational engineering and physics. In *Hopkins University*, pages 345–384, 2002. doi: 10.1115/1.1767847. URL http://dx.doi.org/10.1115/1.1767847.

K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005. ISBN 978-3-540-24372-4. doi: 10.1007/3-540-28901-1. URL http://dx.doi.org/10.1007/3-540-28901-1.

H. Remmel. *Supporting the Quality Assurance of a Scientific Framework*. PhD thesis, 2014.

H. Remmel, B. Paech, C. Engwer, and P. Bastian. Supporting the Testing of Scientific Frameworks with Software Product Line Engineering: A Proposed Approach. In *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, SECSE '11, pages 10–18, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0598-3. doi: 10.1145/1985782.1985785. URL http://doi.acm.org/10.1145/1985782.1985785.

VTK. The Visualization Toolkit: File Formats. http://www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf [Accessed: February 2017].