OpenFOAM® Journal

Volume 5 [Technical Notes], Pages 115–130
ISSN: 2753-8168

Testing Strategies for OpenFOAM Projects

J. W. Gärtner[1,*], G. Olenik[2], M.E. Fadeli[3], L. Petermann[2], A. Kronenburg[1], H. Marschall[3], and H. Anzt[4,5]

[1]Institute for Reactive Flows, University of Stuttgart, Pfaffenwaldring 31, 70569 Stuttgart
Email address: jan-wilhelm.gaertner@irst.uni-stuttgart.de

[2]Scientific Computing Center (SCC), Karlsruhe Institute of Technology (KIT)

[3]Mathematical Modeling and Analysis, Technical University of Darmstadt

[4]Chair of Computational Mathematics, TUM School of Computation, Information and Technology, Technical University of Munich

[5]Innovative Computing Laboratory (ICL), University of Tennessee (UTK)

DOI: https://doi.org/10.51560/ofj.v5.134
Results with version(s): OpenFOAM® v2012-v2306
Repository: https://github.com/FoamScience/of-unit-testing-paper

Abstract. While testing is increasingly recognized as essential in scientific software development, it is not yet standard practice within the OpenFOAM community for developing new solvers and features. This gap stems partly from the challenges of integrating testing into typical OpenFOAM workflows and limited guidance on implementing effective tests. Writing tests for complex software like OpenFOAM based projects presents unique obstacles, including difficulty in configuring tests for various cases. This paper addresses these issues by discussing established test types in the context of OpenFOAM, identifying common challenges in developing tests for this platform, and suggesting best practices to enhance the testability of code based on OpenFOAM. Detailed guidance is provided for integrating the Catch2 unit test framework, alongside two new tools: the `foamUT` framework and the OpenFOAM Benchmark Runner (OBR), which facilitate unit and integration testing. To illustrate these tools in practice, we present three case studies. The first demonstrates the direct integration of Catch2 in the WENOExt project, showcasing test case creation and its synergy with OpenFOAM projects. The second introduces the foamUT framework, which simplifies Catch2 integration for OpenFOAM projects. Finally, the OBR framework is used for benchmarking and integration tests in the OpenFOAM Ginkgo layer. Additionally, we discuss the current state of testing in OpenFOAM and emphasize the need for more comprehensive testing practices within this community, particularly with libraries such as Catch2. Overall, this paper serves as a practical introduction to unit and integration testing for OpenFOAM developers and introduces new tools that lower the barrier to entry, improve test suite robustness, and simplify unit test integration.

1. Motivation

In the realm of writing code and software development, perfection remains elusive, and errors, commonly known as bugs, are an inevitable part of the process. While often undetected or minor, some bugs can lead to disastrous results [1], such as the Ariane 5 failure in 1996, where a buffer overflow error caused the total loss of the rocket and payload. Bugs can exert a substantial impact not only on commercial or industrial software but also on research software, potentially leading to increased costs, project delays, and even leading to the retraction of papers, hindering the careers of aspiring young scientists [2]. Hence, it is paramount to proactively detect and eliminate bugs during the development phase rather than during the application or even after the publication of scientific work. Further, with expanding the FAIR (Findability, Accessibility, Interoperability, and Reuse) principles to research software with FAIR4RS [3], tests for research software may become a required part of future publications or grant proposals. Outside of the scientific community, test-driven development has become a standard software development tool to reduce bugs and prevent catastrophic failures. In certain cases, tests are even mandated by regulations, such as for medical software (ISO/IEC 62304:2006). In addition, well-tested software reduces the risk of

* Corresponding author

Received: 18 January 2024, Accepted: 9 March 2025, Published: 26 April 2025

major bugs and improves the development process, speeds up merging and refactoring, and allows for quickly locating the sources of errors in complex codes. Despite the recognition of the significance of these testing strategies for research software, the de facto standard for tests, in particular in the OpenFOAM community, remains the single validation against experimental or other numerical work [4–6]. This lack of other testing strategies in research software for OpenFOAM is caused by a multitude of reasons, starting from the large differences in the computer science background of researchers to missing scientific recognition for writing clean, tested software, lack of resources, and a general difficulty in designing tests for problems where the answer is part of the research itself [6]. In addition, writing tests for complex software products, e.g., OpenFOAM, is particularly challenging because of the strong coupling between classes and potentially large problem sizes required for testing. Therefore, this paper targets the typical OpenFOAM engineer engaged in code development, offering a concise introduction to software testing strategies and addressing the specific challenges encountered developing projects with OpenFOAM. Additionally, we present proven methodologies for developing, implementing, and executing various test types, alongside introducing novel tools designed to simplify and standardize the testing process for every researcher developing or maintaining OpenFOAM-based software. The paper is divided into three main sections. First, common testing strategies and their application to OpenFOAM are presented. The second section describes the Catch2 unit testing tool and its integration into OpenFOAM, which is not without challenges. Further, the novel `foamUT` framework is presented for simplified integration of unit testing in OpenFOAM projects and the OpenFOAM Benchmark Runner tool for automatic setup, tear-down, and post-processing of test cases. Lastly, three example test cases from real open-source OpenFOAM projects are presented, showcasing the various testing strategies in OpenFOAM projects.

2. Testing Strategies

Among the various types of software testing, four key approaches stand out: static or linting, unit, integration, and acceptance testing, covering the smallest testable unit in a code, e.g., a function, to the correct operation of the complete software product [7].

2.1. Definitions.

However, it is important to note that there is no standardized, universally accepted definition of what constitutes, e.g., a unit or integration test. Consequently, for the purposes of research code, and specifically when dealing with OpenFOAM, we will define the test types for this context.

Static Tests: Static tests aim at improving code quality and detecting and fixing problems before code translation and execution. This can include checking for syntax errors, such as spelling mistakes of functions, enforcing formatting guidelines, or checking for adequate documentation of the code. Hence, static tests can be performed frequently without requiring any compilation or execution of the project code. Further, these tests can be easily integrated into a git development workflow, for example, with the pre-commit hook provided by OpenFOAM in the `${WM_PROJECT_DIR}/bin/tools/pre-commit-hook`.

Unit Tests: Unit tests target the smallest testable entity of a code, a function. Within the context of OpenFOAM, we define a unit test as the test of a class's public member or a free function for its correct behavior.

Integration Tests: Where unit tests check the correct behavior of an isolated function, integration tests verify the correct interaction of different functions and classes with each other. Often, integration test methods are difficult to implement for OpenFOAM applications due to the often inherent strong coupling between different modules. For example, a compressible solver requires the solution of the mass, momentum, and energy equation, which all depend on each other. Hence, isolating a single transport equation for testing requires artificial inputs mimicking the behavior of the other missing parts, which, however, is often not possible to predict. Therefore, in the context of this paper, we define an integration test for OpenFOAM as the execution of an application like a solver, resembling a so-called big-bang integration testing [8]. Another subtype of the integration test is the regression test. Regression tests are the continuous validation of a solver's solution against a previously generated result that has been verified.

Acceptance Tests: In the software industry, acceptance tests are performed by the customer to check if the business requirements have been fulfilled. In the context of research software, this translates to the correct behavior of the complete program to solve the intended problem, e.g., that the developed solver in OpenFOAM delivers correct results. These types of tests often consist of large test case suits, which are then compared against published results or experimental data sets to validate the software's accuracy.

To effectively employ these testing strategies, knowing when to use each test and how they can support the software development process is essential.

2.2. **Testing Strategies applied to OpenFOAM.** Incorporating static tests into the software development cycle is especially valuable for larger projects, where they can be seamlessly integrated with the version control system to ensure that every code change committed complies with the project coding guidelines. In such scenarios, static tests are executed before every code commit, forcing contributing researchers to adhere to the project guidelines. Unit tests offer particular advantages when implementing new classes or functions within the OpenFOAM framework. In this context, developing the test cases in parallel with the class or function under construction is advisable. This parallel development serves a dual purpose. Firstly, it encourages the consideration of class design from a user's perspective, prompting contemplation of required input variables, necessary classes, and methods for their incorporation into the new class or function. Secondly, tests serve as documentation or a manual on how to use the code. Lastly, unit tests, designed to exclusively evaluate the class or function, execute swiftly and can be run frequently, ideally after each code modification. Consequently, debugging and code refinement can progress concurrently, substantially accelerating the development. While good testing practices do require additional time and resources, the costs are often amortized by lower debugging and maintenance costs. However, unit tests alone do not guarantee the correct behavior of a complete program or solver in the case of OpenFOAM. Integration tests are therefore employed to check that the individual classes and libraries are correctly linked and executed. The nature of such tests often does not allow one singular boolean value to be checked to validate the correct behavior. Typically, smaller test cases must be run, resulting in multidimensional fields, e.g., a velocity field. For some instances, the method of manufactured solution (MMS) can be used to allow a direct comparison to an analytically derived solution and verify the results. Yet, for complex problems, it may be easier to compare to another numerically generated data set [9]. This can be either a static data set delivered with the code base or, under the assumption that the previous code version is correct, the output of a prior version of the same software. This test type is called a regression test. These tests have the advantage that the code can be continuously checked against the same case, which can be easily integrated into existing continuous integration functions of the commonly used code hosting services. An example requiring regression testing can be a refactoring of a function or module, e.g., a turbulence model, and validating that the refactoring did not change the physical meaning. However, this also shows the limitation of the regression test, as new functionalities or previously not modeled aspects cannot be compared to an old solution. In the case of new models or functionalities, their correct behavior must be confirmed by comparison to independent numerical or experimental data sets and their interpretation by humans. In most cases, this test type is already known and commonly used in the CFD community to prove the correct behavior of a new model and corresponds to the acceptance test type. Although these tests are essential for scientific publications, they do not replace the other test types. First, acceptance tests are typically larger test cases or test suites that can use up to hundreds, if not millions, of compute hours. Therefore, they are unsuitable for development and debugging with frequent execution of tests. Second, running a full-scale simulation does not guarantee that all edge cases are encountered. For example, the simulation may require a limiter to avoid numerical instabilities, e.g., a limiter for the sub-time-stepping to avoid extremely small time steps, which, however, is not explicitly guaranteed to be encountered and executed. In conclusion, all test types complement each other and should be used if relevant in a research software project.

2.3. **Tests in a development workflow.** The previous section has outlined which test types to use and why tests are an important and, at times, a necessary part of the software development cycle. A key feature and advantage of unit tests is the capability to execute tests fast and frequently. Hence, they can easily be integrated into the code development cycle, especially in combination with a version control tool like Git [10, 11]. In the case of OpenFOAM projects, we want to highlight two aspects. The first is test-driven development (TDD), where developers write smaller tests alongside or prior to developing new methods; the second is automatic test execution triggered by each Git pull request or commit. The benefits of TDD have already been discussed and are widely covered in the literature [10, 11]. Automated test execution serves as a safeguard, catching errors that might seem trivial but could otherwise undermine the reliability of subsequent results, requiring extensive re-validation. Many Git hosting platforms, including GitHub, GitLab, and Bitbucket, offer integrated testing services, allowing tests to run automatically as soon as new code is committed or a pull request is initiated. For these reasons, incorporating automated testing into the development workflow is strongly recommended. Detailed guidance on integrating tests with these platforms is available on their websites; as these services are continuously updated, we avoid providing specific instructions here to ensure the information remains current.

2.4. **Current state of testing in OpenFOAM & OpenFOAM projects.** Automated testing is not entirely new to OpenFOAM, but the testing support in the current release remains limited in scope and functionality. Under the `applications/test` directory, OpenFOAM provides basic tests for various classes and functions. However, these tests lack a standardized pass/fail status; for instance, in the vector tests, results are printed directly to the terminal, requiring the user to manually inspect the output. This approach is prone to errors and lacks flexibility for integration into automated workflows, as it requires users to write custom scripts for parsing and comparing text outputs – an approach that is error-prone and difficult to adapt or extend to new tests. We do not expect extensive unit testing to be added to OpenFOAM due to the amount of work required to retrofit tests. But the lack of a clear testing strategy in OpenFOAM also hinders testing in OpenFOAM based community projects.

Many projects based on OpenFOAM already employ some form of tests, which typically involve running small tutorial cases or verifying that a solver completes without crashing. While this can help catch major failures, it often provides limited insight into specific functional issues, as the output is either not machine-readable or fails to pinpoint which module or function is malfunctioning. Some projects have incorporated text parsing to automate the comparison of terminal outputs, but this approach remains limited, as it can lead to false negatives due to slight variations in numerical results or formatting. These limitations highlight the need for more robust testing practices in OpenFOAM, using dedicated testing libraries that support automated result validation and provide human- and machine-readable reports. Such tools can streamline test integration, improve accuracy, and facilitate the identification of specific issues within complex workflows, thereby enhancing the reliability and maintainability of OpenFOAM-based projects.

Notable OpenFOAM projects such as the load balancing library DLBFoam[1], chemistry load balancing library for OpenFOAM v2306[2], wall-modeled LES[3], or directChillFoam[4] employ these testing strategies. However, while these projects demonstrate the integration of testing strategies, the majority of OpenFOAM-based projects still lack systematic testing approaches. In the following sections the challenges of testing in OpenFOAM (Sec. 2.5) and proven solutions to streamline and simplify testing processes are presented (Sec. 3).

2.5. **Challenges of testing in OpenFOAM.** Writing tests for a large and complex code basis a posteriori is a demanding task. Ideally, the software components are already designed to be testable, e.g., allowing black box testing[5], avoiding the introduction of dependencies, etc. [13]. In the case of OpenFOAM, one significant challenge for testing is the complex hierachy of OpenFOAM's classes and various depencies among these classes, partially caused by the complex physical problems solved, which require a significant amount of supporting code to test even small and simple functions. Further, the test case state, e.g., the case files are often decoupled from the source code. To give an example, the test of the calculation of the drag force on a spherical particle requires the construction of the appropriate cloud class and corresponding particles, which in return requires the construction of an `fvMesh` object that has to read information from the hard disk and necessitates a specific case directory structure, see Fig. 1. Hence, for testing a simple drag function, several additional classes must be constructed, which introduces further dependencies and raises the risk of introducing bugs in the test itself. Further, the execution of the test requires a specific case directory following the OpenFOAM structure, e.g., for the construction of the mesh or reading the model settings. While some classes of OpenFOAM provide alternative constructors that can use input streams or that can be constructed by components without reading a file, e.g., for the `fvMesh` class, the construction of these inputs is challenging by itself[6]. Further, some classes use constructors for their private members which require reading files from the hard drive, for example the `KinematicCloud` class which has to read a cloud properties file during construction. Consequently, formulating a minimal OpenFOAM case encompassing a mesh and a `system` folder emerges as an essential and particular facet of unit testing for OpenFOAM. However, reading additional files from disk during a unit test can introduce further potential complication and should be avoided if possible.

Another particularly challenging subject of testing in OpenFOAM is the parallel execution with MPI. In an attempt to make writing OpenFOAM solvers easier, the MPI communication is capsuled and hidden away in its own OpenFOAM `Pstream` library. This, however, limits the testing to the MPI interfaces provided by OpenFOAM, e.g., no explicit MPI_BARRIER function is provided. Getting access to the low-level MPI functions is possible but challenging. Further, detecting if an error originates from MPI

[1] `https://github.com/Aalto-CFD/DLBFoam`
[2] `https://github.com/ITV-Stuttgart/loadBalancedChemistryModel`
[3] `https://github.com/timofeymukha/libWallModelledLES`
[4] `https://github.com/blebon/directChillFoam`
[5] Examples for black box testing strategies are given in [12]
[6] See following project: `https://github.com/JanGaertner/unitMesh`

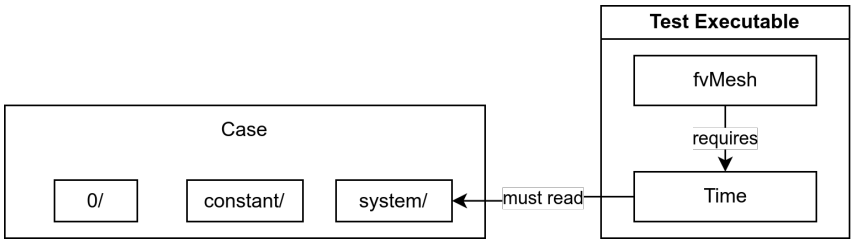

Figure 1. Sketch of the OpenFOAM fvMesh dependency to read the system/controlDict file.

calls or the locally executed code is not always easy or possible. Yet, parallel execution is a vital part of many OpenFOAM functionalities and should be tested.

2.6. **Best Practices for Unit Testing OpenFOAM Code.** Following the discussion on the current state of testing in OpenFOAM code, this section outlines best practices for enhancing testability and robustness in OpenFOAM applications. By addressing the unique challenges associated with file-based operations and focusing on creating testable code, developers can ensure comprehensive test coverage and early detection of potential issues. The practices discussed here highlight the importance of proper class design and case setup.

2.6.1. *Design OpenFOAM Classes for Testing.* In addition to the general principles for designing testable classes in C++ [13], it is advisable to design class constructors that accept either an `Istream` or dictionary as an input to set all required properties. Hence, it avoids the necessity of reading a file from a disk. The required input streams or dictionaries can then be constructed in the test and passed directly to the class constructor without having to write or read files from the disk. Further, it is advised that the class provides an interface to the `objectRegistry`. This can be achieved, e.g., by including a constant reference to the mesh, which allows the lookup of required fields from the object registry. This has the benefit that the class APIs are more stable, as additionally, required fields can be looked up directly in the class and do not need to be explicitly passed.

In case the developer does not have control over the tested code, constructing a dictionary from a `string`, writing it to disk, and immediately constructing their desired object is the best approach to avoid accidental interference from other unit tests. This still holds even if the setup and tear-down phases of the unit tests allow sharing OpenFOAM cases between tests, which is often the case. An example of this approach is demonstrated in Lst. 11 in the case study section, Sec. 4.2, configuring the `dynamicFvMesh` class through the use of a `Foam::IStringStream` object, which can be constructed from a string and can be used to generate the configuration dictionary, which `dynamicFvMesh` required for this class. Note that this approach also makes the required content of such configuration dictionaries very clear and keeps them up to date.

2.6.2. *Isolate functionalities.* An important technique for improving code modularity is to decouple implementation from complex data structures by imlementing kernel functions. For OpenFOAM this could mean to implement free kernel functions that operate on pointer to scalar values instead of opering directly on OpenFOAM-specific types, e.g., `fvScalarFields`. For instance, a free function might work with a raw pointer to a contiguous memory block, such as an array or `std::vector`, or follow a standard C++ pattern using begin and end iterators. This approach is demonstrated in Lst. 1, where the function `transform()` applied to the OpenFOAM data structure `volScalarField`, uses underneath a free function, here called `transformKernel()`, which requires only the begin and end iterator to the data structure and a raw pointer to a new scalar field to return the result [7].

[7]See `https://github.com/hpsim/OGL/blob/v0.5.4/unitTests/test_HostMatrix.C` for an actual implementation of this approach.

```
template<class iter>
void transformKernel(iter begin, iter end, scalar* data) {/*some code*/};

Foam::volScalarField transform(const scalarField& values) {
    volScalarField out(...);  // Construct new and empty volScalarField
    transformKernel(values.begin(),values.end(),out.data());
    return out;
}
```

Listing 1. Pseudo code implementing a test friendly transformKernel function

One advantage of this approach is that the `transformKernel()` can now be tested without instantiating a `volScalarField` which would require a mesh object and object registry. Instead, `std::vector`, or any other container type with a begin and end iterator can be used as shown in Lst. 2.

```
SOME_TEST() {
    std::vector<scalar> in { 1.0, 2.0, 3.0};
    std::vector<scalar> out ();
    out.reserve(in.size());
    transformKernel( in.begin(), in.end(), out.data());
    SOME_ASSERT(out);
}
```

Listing 2. Implementation of a unit test for the transformKernel function

3. Testing Tools for OpenFOAM

A vast ecosystem of unit testing libraries has emerged in the realm of computer science to assist developers in this process. These libraries offer a standardized and well-organized framework for creating and executing tests, simplifying the maintenance and scalability of test suites as the codebase expands. Moreover, they provide a wide range of assertion macros and utilities, streamlining the process of verifying code behavior and reducing the likelihood of overlooking edge cases. Given that OpenFOAM is primarily written in C++, this discussion will focus on the Catch2 framework. Catch2 is by far not the only commonly used unit testing framework for C++. However, the design of Catch2 allows a comparably simple integration into OpenFOAM. Integrating Catch2 into an OpenFOAM code development process can bring significant benefits regarding code reliability and maintainability. The different build systems of OpenFOAM and Catch2 make integrating the two frameworks not straightforward. Also, executing tests for functions or models executed in parallel with MPI is usually not trivial. At last the different test cases, their setup and tear-down have to be managed as well. These issues are addressed in the following sections, which first present the direct integration of Catch2 in an existing OpenFOAM project. In the following subsection the `foamUT` utility is presented, providing a user-friendly and simple to use framework, which handles these details in the background, allowing the user to concentrate on only writing test code. The last section describes the OpenFOAM Benchmark Runner project, which allows automatic execution of different test cases, taking care of the test case setup and execution considering different hardware platforms and job submission tools, required, e.g., in an HPC context.

3.1. Catch2 and OpenFOAM.

Catch2 is a comparably lightweight and easy-to-install unit testing framework. However, the integration of Catch2 , or any unit testing framework, into OpenFOAM poses several challenges, as mentioned in the introduction, see Sec. 2.5. These issues are addressed in the following, and an example workflow is presented on how to couple OpenFOAM and Catch2.

The general setup of Catch2 tests consists of a main function and files containing the tests enclosed in a `TEST_CASE()` macro. It is generally advised to keep the testing code (C++ compilation units defining the test cases inside `tests`) separate from the target library code (which usually resides in `src`) [14]. An example structure of an OpenFOAM project using Catch2 is visualized in Fig. 2. Here, the additional tests folder is added alongside the `src/` folder of the project, containing a `src/` and `cases/` directory. The source code of the individual tests resides in the `tests/src/` directory, and the OpenFOAM cases required for execution are stored in the separate `tests/cases/` directory. To reiterate, the OpenFOAM cases are necessary as OpenFOAM tests often have to read information from the hard disk to construct fundamental OpenFOAM classes, such as the `fvMesh` or the object registry. A minimal test case contains at least a `system/` folder with a `controlDict`. However, it may also include additional files required for execution in the `constant/` or an initial time state in a `0/` directory. At last, the Catch2 library has to

be coupled with the test source code in `tests/src/`, which is depicted by the arrow from Catch2 to `src/` in Fig. 2.

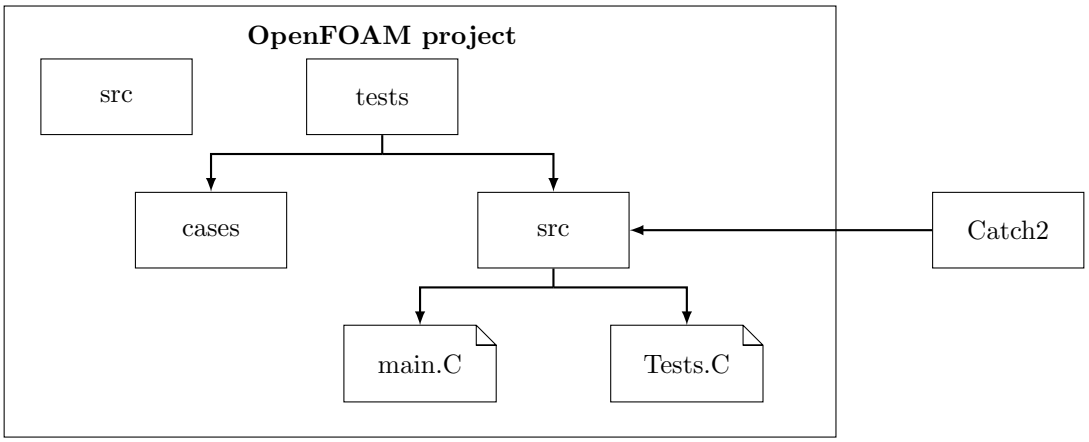

Figure 2. General setup of Catch2 with an OpenFOAM project.

3.1.1. *Compiling Catch2 and OpenFOAM tests.* The library Catch2 is compiled and installed with the build-tool CMake, which cannot be easily integrated into the standard build system of OpenFOAM, called wmake. Therefore, the unit testing framework must be installed prior to compiling the OpenFOAM test cases using the CMake instructions given on the Catch2 GitHub page. As Catch2 uses standard CMake functionalities and provides detailed installation instructions, a compiled and installed Catch2 library in the path `Catch2_Install_Path` is assumed in the following. Here, the variable `Catch2_Install_Path` denotes the path where Catch2 is installed, which can be specified by the user with CMake command line arguments [8].

Once Catch2 is installed, it must be linked to the test codes of the OpenFOAM project. The test code includes a custom main function, see Sec. 3.1.2, and the source files defining individual tests using Catch2's `TEST_CASE()` macro. To compile the custom main function with the OpenFOAM tests and link them to the Catch2 library, the `Make/options` have to be adjusted. The `Make/options` file of an executable is divided into two sections, first `EXE_INC` with all include statements and possible command line arguments, and `EXE_LIBS` listing the required libraries. To include the Catch2 library in the linking, the name of the libraries as well as their path has to be provided in the `EXE_LIBS` list, see Lst. 3. With these additions, the test code can be compiled using the `wmake` function.

```
1  EXE_INC = \
2  -std=c++14 \
3  - ADD INCLUDES
4  ...
5  EXE_LIBS = \
6  -L<Catch2_Install_Path>/lib \
7  -lCatch2 \
8  -lCatch2Main
```

Listing 3. Example of Make/options file with Catch2. Note: As Catch2 requires at least C++ standard 14, while OpenFOAM, up to version v2312, still uses C++ 11 with the GCC compiler, the statement `-std=c++14` has to be added to the `EXE_INC` list.

3.1.2. *Command line arguments of Catch2 and OpenFOAM.* For many tests of OpenFOAM, the default include statements of `setRootCase.H`, `createTime.H`, and `createMesh.H` are required. This standard setup of the OpenFOAM environments requires reading the command line arguments given as the number of command line inputs and the character array, `char* argv[]`. However, Catch2 also needs to parse the command line arguments, which is in conflict with OpenFOAM. To achieve seamless interoperability between Catch2 and OpenFOAM, an unambiguous delimiter (e.g., three hyphens) as a convenient way to distinguish between arguments designated for Catch2 and those intended for OpenFOAM can be used, as

[8]For further details, see `https://github.com/catchorg/Catch2`

exemplified in Lst. 4. An implementation of such a delimiter for integration of Catch2 and OpenFOAM command line arguments is shown in the foamUT framework, which is presented in Sec. 3.2.

```
./testBinary [catch_options] --- [openfoam_options]
```

Listing 4. Example of command line filtering for OpenFOAM and Catch2

While Catch2 includes a default main function (if linking against `libCatch2Main.a`), a custom main function for Catch2 is required to separate the command line arguments for Catch2 and OpenFOAM. A typical approach is to parse the command line and to create a new character array, which is then used as an input for the functions in `setRootCase.H` of OpenFOAM. Examples of how this can be realized can be found in the projects WENOExt[9] [15] and `foamUT`[10] [16] on GitHub.

3.1.3. *Writing Catch2 test cases.* Catch2 offers a variety of macros to write unit tests (for example, `REQUIRE` to check boolean values or `Approx` for floating point comparison). A minimal example unit test case for validation of the initialization of three components in `vector::zero` is illustrated in Lst. 5.

```
TEST_CASE("vector::zero initializes three components", "[tags]") {
    REQUIRE(vector::zero.size() == 3);
}
```

Listing 5. Basic form of a Catch2 test case

In brief, Catch2 test cases are named and tagged (the first and second arguments to the `TEST_CASE` macro in Lst. 5) for filtering at test execution, provided it can effectively process Catch2 arguments. This functionality proves valuable for segregating various test execution modes, such as testing functions in serial and parallel settings, as well as specifying the OpenFOAM cases for test execution if necessary. One use case could be tagging all test cases using the standard OpenFOAM cavity tutorial as a basis with the keyword `[cavity]`. Execution of the test library with the tag `[cavity]` would then execute all tests that are compatible with the cavity mesh and case setup.

Another valuable test construct is `TEMPLATE_TEST_CASE`, which generates test cases for function or class templates with specified template arguments. For example Lst. 6 presents a unit test case, which expressively tests the constructor of `List` class templates for specific types, i.e. `scalar` and `vector`.

```
TEMPLATE_TEST_CASE("List empty constructor", "[tags]", scalar, vector) {
    REQUIRE(List<TestType>().size() == 0);
}
```

Listing 6. Basic form of a Catch2 test case for templates

3.1.4. *OpenFOAM's error handling.* The default error-handling mode in OpenFOAM is to terminate with fatal error codes upon critical failures. While this approach is suitable for production runs of CFD solvers, it is suboptimal for unit-testing scenarios, where proceeding to the next test case after a fatal failure is generally desired. Additionally, Catch2 provides support for exception matching, allowing developers to ensure that a function throws errors in specific situations. To avoid aborting when an OpenFOAM Fatal Error is encountered, it is recommended to enable exception handling before initiating the Catch2 session, with `FatalError::throwExceptions`, as demonstrated in Lst. 7. This will allow the subsequent unit tests to execute even if the current one throws an exception. OpenFOAM's default behavior will result in halting all unit tests because of the call to abort on Fatal Errors.

```
int main(int argc, char *argv[]) {
    // Use exceptions instead of aborting on FATAL ERRORs
    FatalError::throwExceptions();
    Catch::Session session; // Instantiate and start tests session
    session.run();
}
```

Listing 7. Switching exception throwing in OpenFOAM

Unfortunately, the error handling of parallel test cases is not straightforward due to the handling of exceptions, deadlocks, and time-outs in MPI. Therefore, any solution that does not lead to the termination of the tests when an error occurs in parallel execution will be MPI-implementation specific [17].

[9] `https://github.com/WENO-OF/WENOEXT/blob/master/tests/src/main.C`
[10] `https://github.com/FoamScience/foamUT/blob/master/tests/testDriver.C`

3.2. **Unit testing OpenFOAM code with the foamUT framework.** The `foamUT` toolkit represents a general-purpose unit testing solution tailored specifically to OpenFOAM projects [16]. Its primary goal is to address the unique challenges developers face when testing OpenFOAM-based software, especially in the context of the complex dependencies and build environments across different OpenFOAM forks and versions.

A key feature of `foamUT` is its flexibility and ease of integration. Developers are responsible only for writing their test code and specifying build options. The toolkit abstracts away common complexities such as handling serial and parallel setups, providing generic OpenFOAM cases, and managing test drivers. At the same time, it allows developers the flexibility to customize as needed to fit specific project requirements. `foamUT` also promotes healthy testing practices, such as randomizing test inputs, which helps to avoid bias in tests and to detect defects caused by unexpected inputs. Furthermore, it is optimized for continuous integration (CI) environments by separating test program execution (`testDriver`) and its dependencies from the production code in the target project (as shown in Fig. 3) as applications and solvers destined to production use tend to demand a lot more resources and computation time.

An important aspect of `foamUT` is its compatibility across multiple OpenFOAM versions and forks. This broad compatibility reduces the effort developers need to spend on adjusting their tests for different OpenFOAM environments, enabling a unified approach to testing that works consistently across projects if support for multiple forks is desired. `foamUT` is a framework that provides a standardized way to manage Catch2 and write tests for OpenFOAM functions and classes. The general workflow for unit testing with `foamUT` is shown in Fig. 3 and comprises several steps[11].:

(1) Develop the unit test code, tagging each test unit with either `[serial]`, `[parallel]` or both, as well as the target OpenFOAM cases to run on. The test code can reside in the target project tree's `tests/targetLibTests` folder. `foamUT` will then run tests in two passes, one in serial, and the second pass in parallel with four (4) MPI ranks by default. A minimal unit test is given in Lst. 8, which would be implemented, for example, in the `myTests.C` file within the project tree, as shown in Fig. 3.
   - The standard `foamUT` test drivers initialize some global variables in a consistent way with their instantiation in OpenFOAM solvers, including the Time and command-line `argList` objects, as demonstrated in Lst. 8.
   - Lst. 8 illustrates a unit test to ensure the time index starts at 0. This test is set up as a dummy one to ensure the correct functioning of `foamUT` with the used OpenFOAM version. In particular, this unit test showcases some of the most important `foamUT` conventions, mainly using test case tags to specify the OpenFOAM cases to run the test on (e.g. the `cavity` case, which is included in the testing toolkit), and parallel execution tags, so the unit test will run both in serial and parallel modes.

```
1 extern Time* timePtr;
2 extern argList* argsPtr; // not used in this snippet
3 TEST_CASE("Check time index", "[cavity][serial][parallel]") {
4     Time& runTime = *timePtr;
5     REQUIRE(runTime.timeIndex() == 0);
6 }
```

Listing 8. Example timeTests.C unit test file using foamUT conventions

(2) Developers are also required to provide `Make/{files,options}` to link test units against target libraries such as `libOpenFOAM.so`.
(3) `foamUT` needs to be cloned into a separate directory outside of the project tree, usually in a temporary location. A simple `git clone` is sufficient to achieve this (The important bits from `foamUT` repository structure are illustrated in Fig. 3).
(4) The developer must copy, or symlink, library test code to `foamUT/tests`. This approach enables the toolkit to execute both the project-specific tests and the default tests, which make sure `foamUT` is compatible with the OpenFOAM version the user is using. However, the latter can be easily removed if they are not required. Figure 3 also shows how `tests/targetLibTests` folder, which contains the actual test code (`myTests.C` in this example) and the corresponding `Make` folder, is sym-linked inside `tests` folder from the `foamUT` repository tree.
(5) `foamUT` provides a default shell-case with a simple "lid-driven cavity" mesh, which can be used to run the tests on without further developer efforts. If custom OpenFOAM cases are required

[11]Further detailed information can be found in the GitHub Wiki: `https://github.com/FoamScience/foamUT/wiki`

to run the tests, the developer then needs to copy them into the `cases` folder in the `foamUT` tree before attempting to run the tests (Right hand side of Fig. 3).

(6) Executing `Alltest` script (from the `foamUT` repository tree in Fig. 3) will iterate over all provided OpenFOAM cases and run corresponding unit tests both in serial and parallel modes.

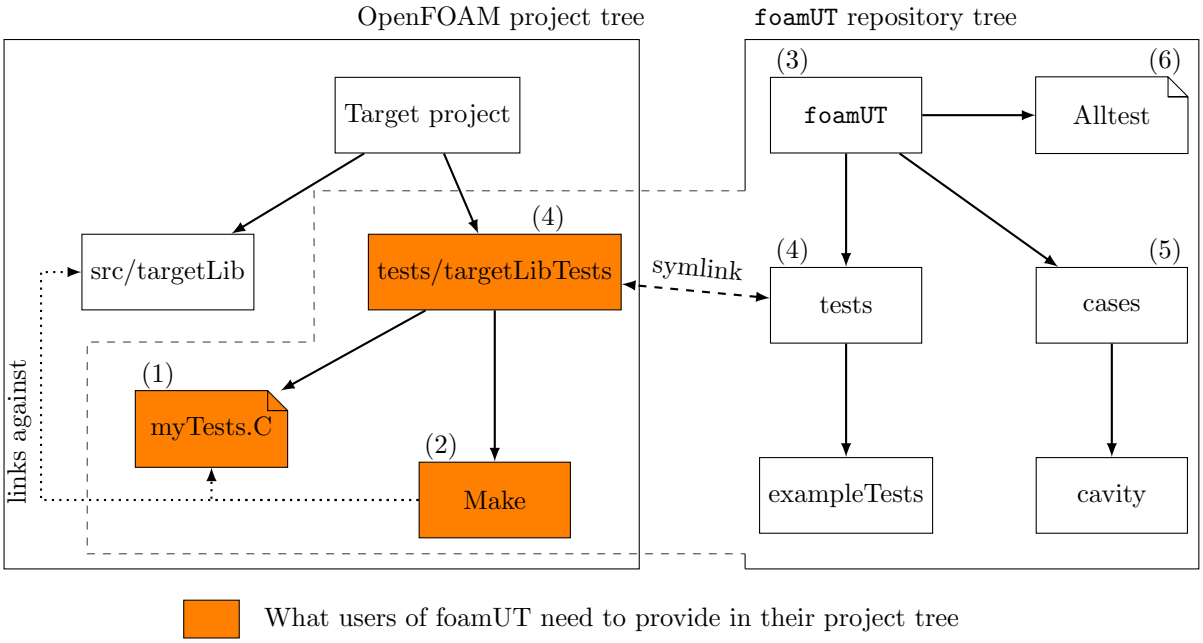

Figure 3. Default unit-testing workflow with `foamUT`. Numbers represent the different workflow steps, files and folders are represented by boxes with and without earmarks respectively, and solid arrows indicate the contents of a folder.

Adhering to the guidelines depicted in Fig. 3 ensures that a fresh testing environment is deployed each time the `Alltest` script is executed. This practice helps maintain a clean project tree, as the project only needs to manage the actual test code without the need to include `foamUT`, any "test driver", or `Catch2` libraries as dependencies. Furthermore, this workflow is well-suited for Continuous Integration (CI) tasks since the only prerequisite is to create a container with OpenFOAM installed that can compile the target libraries.

3.3. **Setup and Tear-Down of Test Cases with OBR.** In addition to the integration of Catch2, or other unit testing software, in the OpenFOAM development workflow, setup, tear-down and management of the test cases pose an additional challenge, see Sec. 2.5. Often, several test cases have to be managed and set up properly for each unit or integration test, e.g., by modifying the dictionary files for the respective tests. Further, benchmarks may be conducted on different hardware platforms to ensure functionality and good performance for different hardware vendors. This is particulary challenging in an HPC context which requires the use of job submission systems like slurm. While the aforementioned points can, in principle, be handled by ordinary shell scripts, in practice, these shell scripts are often hard to develop, maintain, and extend. Thus, to simplify setting up different integration test cases and handle the benchmark workflows' complexity while ensuring the benchmark results' reproducibility, a software utility named OpenFOAM Benchmark Runner (OBR) [18] has been developed.

3.3.1. *OBR workflow.* The basic workflow of creating, running, and post-processing parameter studies with OBR is shown in Fig. 4. All OBR workflow steps, e.g., modifying the `blockMeshDict` file or setting up different linear solver in the `fvSolution` file based on a base case, are defined via a YAML workflow file (see left-most block in Fig. 4). This separates the desired outcome, for example, setting the `endTime` of a simulation to a specific value, as shown in Lst. 9, from the required steps or code to achieve this goal. While the user is responsible for defining the required state, OBR is responsible for implementing an approach to reach the required state, for example, by implementing functionality to modify OpenFOAM dictionaries on disk.

```
- controlDict:
    endTime: 1.0
```

Listing 9. Required YAML workflow file entry to set the endTime to 1.0

To enable machine-specific workflows, the YAML workflow file can contain references to environment variables using the `${{env.VARIABLE_NAME}}` notation. This allows to consider machine-specific properties, like the number of cores per node for the domain decomposition. The YAML workflow files are copied to the target machine or kept in a dedicated test folder in case of integration tests with GitHub actions. Based on the encoded workflow, on the execution of `obr init`, OBR creates a directed acyclic graph (DAG) of jobs. Each job is represented by a subdirectory with a unique identifier in a workspace folder, the desired target state, and contains the required simulation case files. When `obr run -o generate` is executed, OBR is responsible for setting up the OpenFOAM cases to the target state by executing certain operations such as fetching the base case, running `blockMesh` to generate a computational grid, manipulating dictionary file entries to set up simulation properties such as solvers. After the setup of the individual cases `obr run -o runParallelSolver` and `obr submit -o runParallelSolver` execute, the solver runs either locally or via an automatically generated submission script, e.g., for slurm. `obr status` checks the status of individual cases and stores properties such as latest time step, execution time, etc. in a cache file to make it retrievable via `obr query` and OBRs python API. Finally, results can be post-processed by applying post-processing scripts to all eligible cases, including parsing of the log files (see right-most block in Fig. 4). After the post-processing step, general simulation properties, the number

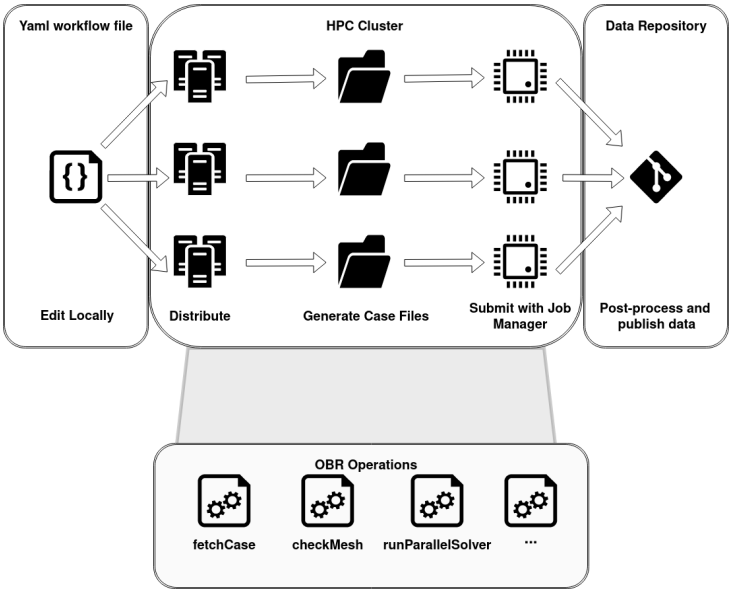

Figure 4. Case generation, execution, and post-processing workflow with OBR, showing different stages of a typical workflow on multiple HPC cluster and typical operations.

of iterations spent in a specific linear solver, or the timestep continuity errors can be accessed via obr queries. For the integration test, the `--verify-against=name.json` option of `obr query` compares the output of the query command to a given `name.json` file.

4. Case Studies

The previous sections elaborated on the theoretical background, defining test types, workflows, and use of the test framework Catch2, `foamUT` and `OBR`. The following section presents three cases to provide insight into how this can be applied to real-world problems. The first part presents the WENOExt library [15, 19], which uses Catch2 for unit and integration testing. However, the necessity to write a custom main function for the tests and the rather uncommon use of CMake for OpenFOAM projects to compile the library, motivates the `foamUT` project explored in Section 3.2, here applied to the second case study a load-balanced adaptive mesh refinement library (blastAMR). The last part presents a different testing approach for the OpenFOAM Ginkgo Layer (OGL) project, combining the `OBR` software library with the GitHub actions workflow to conduct integration tests considering various case setups and library features.

4.1. **Catch2 case study: The WENOExt library.** The open-source library WENOExt, which introduces Weighted Essentially Non-Oscillating schemes to OpenFOAM [15,19], uses Catch2 to run unit and integration tests [12]. Following the proposed directory structure of Fig. 2, a `tests` directory is present at the root of the project. Within this directory resides the `src` folder containing the custom main function, as well as the individual tests. In this project, the Catch2 library is included as a git sub-module and is installed within the project structure. Further, the `tests` folder contains a script to run the tests and a `report` folder in which the results of the tests can be visualized as a report in HTML format.

The test library of WENOExt includes unit tests for fundamental functions, e.g., a new `List3D` class, special math functions required for the WENO library, etc. Additionally, integration tests to test the integration of WENOExt into OpenFOAM's divergence operator, e.g., with the advection of a slotted disk, are included. These test types validate the accuracy of the scheme by calculating the numerical diffusion of a scalar field in a pure advection case and constitute an example of method of manufactured solution (MMS) by comparison to an analytical function where the derivative is computed precisely. The integration and accuracy test cases require different case setups, e.g., a 2D and 3D case for the advection with different boundary conditions. Hence, not only one OpenFOAM case but several are included in the `Cases` directory. As all tests are compiled into one single executable, the execution of the different test types is managed using Catch2 tags. This motivates the introduction of a shell script to run the tests, which handles the execution of the tests with their Catch2 tags in the matching OpenFOAM case in the `Cases` directory.

The aforementioned challenges of writing a custom main function, as discussed previously in Sec. 3.1.2, are addressed by using a global variable of type `Foam::argList*` to mimic the behavior of `setRootCase.H` and make it available to all test cases. An example of how a `TEST_CASE()` can look like is shown in Lst. 10.

```
// include statements
#include "globalFoamArgs.H"
...

TEST_CASE("WENOUpwindFit Test","[upwindFitTest]") {
    // Setup OpenFOAM environment
    // Replace setRootCase.H for Catch2
    Foam::argList& args = getFoamArgs();
    #include "createTime.H"        // create the time object
    #include "createMesh.H"        // create the mesh object

    ...
}
```

Listing 10. Example test case of WENOExt to create the OpenFOAM environment with a fvMesh and Time object. See the WENOUpwindFit-Test.C source code of WENOExt.

The option to execute parallel runs is included with an additional Catch2 command line argument, using the `cli` parser of Catch2. However, parsing additional OpenFOAM arguments is not supported by this main function. The rather complicated handling of command line arguments and different test cases within one executable motivates the use of the `foamUT` tool as an alternative simple-to-use tool, which handles these details in the background, allowing the user to concentrate on the writing of tests.

4.2. `foamUT` **case study: A load-balanced adaptive mesh refinement library.** In this example, the `blastAMR` library, a port of adaptive mesh refinement (AMR) functionalities from `blastFOAM` [20], is tested using foamUT. These unit tests ensure that functionality remains consistent with the original library. A primary focus is verifying that `mesh.update()` correctly handles both refinement and unrefinement, which are tested on different OpenFOAM mesh types composed of combinations of polyhedral, hexahedral, 2D, or 3D elements.

The tests begin by refining cells in a stationary box and checking that `mesh.update()` produces the expected cell count. After moving the box, the code tests whether the new location refines appropriately and the old mesh location coarsens. Tests cover key parameters, including:

- Maximum refinement level, since different refinement levels might be treated differently by the library code
- Refiner engine type. Only the polyhedral refiner is tested extensively.

[12] `https://github.com/WENO-OF/WENOEXT`

- Load balancing switch (Checks with and without load balancing)

```
// Info : TEST_CASE, GENERATE, and REQUIRE are all Catch2 macros
TEST_CASE ( "Check refinement/unrefinement polyhedral AMR/LB functionality",
"[hex2D][hex3D][poly2D][poly3D][serial][parallel]" ) {
    // STAGE 01: Setup required state
    word refiner = GENERATE("polyRefiner");
    word balance = Pstream::parRun() ? GENERATE("no", "yes") : GENERATE("no");
    label nBufferLayers = GENERATE(1, 2);
    label maxRefL = GENERATE(1, 3);
    // Supported constant/dynamicMeshDict entries
    IStringStream is(
       "dynamicFvMesh   adaptiveFvMesh;"
       "balance         "+balance+";    refiner         "+refiner+";"
       "lowerRefineLevel 0.01;    unrefineLevel 0.01;"
       "nBufferLayers   "+Foam::name(nBufferLayers)+";"
       "maxRefinement   "+Foam::name(maxRefL)+";"
    );
    // Create mesh object by reading the input string stream
    IOdictionary dynamicMeshDict(
    IOobject(
       "dynamicMeshDict", runTime.constant(), runTime,
       IOobject::NO_READ, IOobject::NO_WRITE
       ), is
    );
    dynamicMeshDict.regIOobject::write();
    #include "createDynamicMesh.H";
    // Record initial number of cells, boxCells is a cellSet
    label origRefBoxNCells = returnReduce(boxCells.size(), sumOp<label>());
    // STAGE 02: Run testing code
    setFieldValues();
    for(label i = 0; i < maxRefL; i++) {runTime++; mesh.update();}
    // Check the refined box again with a cellSet
    label newRefBoxNCells = returnReduce(boxCells.size(), sumOp<label>());
    // STAGE 03: Check for test conditions
    REQUIRE(newRefBoxNCells >= 4 * origRefBoxNCells);
}
```

Listing 11. Example unit test case for checking correct load-balanced AMR funtionality depending typical use cases. The unit test is set to run both in serial and in parallel, following the `[serial][parallel]` tags , on custom OpenFOAM cases which provide combinations of hexahedral and polyhedral meshes.

An example unit test is shown in Lst. 11, in addition to testing the functionality, it also clarifies what settings are supported by the load-balanced adaptive mesh refinement mechanism and their well-tested value range, hence the unit test acts as an up-to-date documentation for the target class. The test parameters, if any, are incorporated at a test-case level, and generators are usually used to produce a matrix of test cases with a minimal amount of code, as illustrated by using the `GENERATE` macro in Lst. 11. With Catch2, each generated branch of the test case executes the tested functions in an isolated sandbox, where other branches' environments and results do not interfere with each other, with the selected parameters and evaluates passing conditions.

4.3. **Integration testing and benchmarking of the OpenFOAM Ginkgo Layer (OGL).** The following section discusses the automation of integration testing and benchmarking of the OpenFOAM Ginkgo Layer[13] (OGL) [21] with the software library OpenFOAM Benchmark Runner (OBR)[14] [18]. An issue arising with OpenFOAM testing is to ensure that the test cases are in a proper state prior to test execution. This may include mesh generation with `blockMesh` and decomposition of the mesh or the clean up of generated files and modified dictionary entries. This is of particular importance for benchmarks where the mesh size or time steps are varied for the same test case, hence requiring modification of the files stored on disk within the test case. This and related problems are tackled and solved by the OBR software presented in the following section using OGL as a case study.

[13] `https://github.com/hpsim/ogl`
[14] `https://github.com/exasim-project/obr`

OGL is an external plugin for OpenFOAM that allows offloading of the linear solver to GPUs facilitated by the Ginkgo library. Hence, ensuring correct behavior is crucial for valid simulation results. To fulfill this requirement and guarantee code quality standards throughout the development process, OGL uses GitHub actions functionality[15] as the principal test environment, responsible for executing the test runs. Here, several test types, like static tests to ensure correct code formatting and reduce typos, unit, and integration tests are executed. The different test workflow runs are specified by YAML files stored in a designated `.github/workflows` folder and are typically executed whenever a new pull request is opened or a set of commits is pushed to the repository on the GitHub server. The discussion in this case study, however, will focus on the setup and execution of the integration tests of OGLs CI/CD pipeline. The test pipeline is defined within the `.github/workflows/build-foam.yml` and `.github/workflows/integration-tests.yml` files, and is comprised of first building OGL against different versions of OpenFOAM, and only after a successful build the integration tests are executed. Fig. 5 shows a screenshot of the build matrix and the dependencies between different jobs. It can be seen that, at first, the build matrix is created by reading path and version information from a file and emitting a JSON dictionary, which is used within all subsequent jobs. After the creation of the build matrix the build jobs are executed, which compile OGL against. In the screenshot, two build jobs are shown, build-v2212 and build-10, corresponding to ESI v2212 and OpenFOAM Foundation version 10 of OpenFOAM. Only if the build test succeeds the integration tests are executed, which consist of two sub jobs for each build, namely the setup of the test cases and the execution and verification of the simulation results.

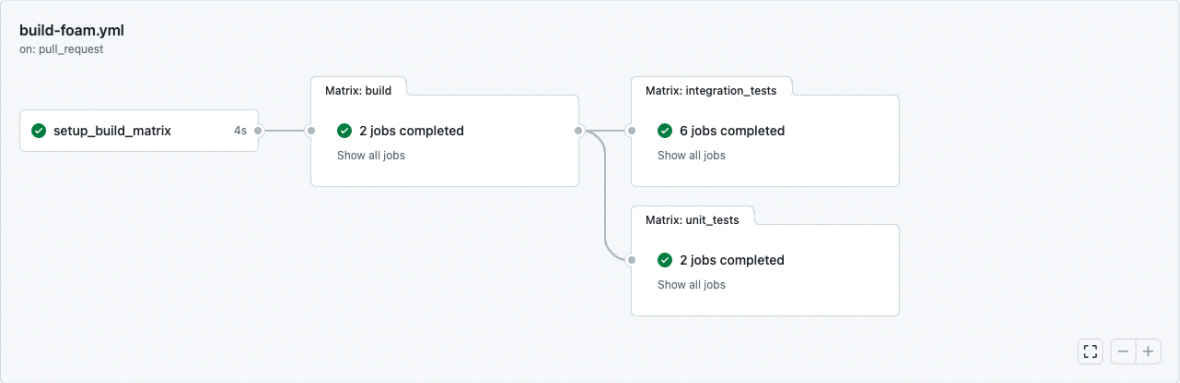

Figure 5. Screenshot of the build and integration test workflow for OGL using GitHub actions, showing the setup, build, and unit and integration test stages.

For the integration tests of OGL, the following setup is implemented. The required YAML workflow file is stored as `cavity.yaml` in the `test` sub-folder and an excerpt of this file is given in Lst. 12. The `case` defines the base from which the parameter variation is built, here the cavity case in the `incompressible/icoFoam` tutorial folder. After copying the base case into the workspace folder, several operations are executed defined by the `post_build` section. This includes adding `libOGL.so` to the list of loaded shared objects, executing `blockMesh`, and decomposing the case. After the base case is fully set up, all variations from the `variation` section are applied. For brevity in Lst. 12 only one variation is shown, changing the solver for the pressure execution to `GKOCG`.

The required steps for executing the GitHub action workflow for OGLs integration tests are summarized in Lst. 13. However, Lst. 13 is merely a simplification of `.github/workflows/integration-test.yml` contents since the original workflow is split up over several steps and includes caching of files and folders to avoid unnecessary regeneration of files and folders across different workflow runs.

The last command in Lst. 13 shows a filtered query, which checks the global state of the simulation, the continuity errors, and the Courant number of all cases that satisfy the filter predicates and compares it against a set of criteria defined in `${{matrix.Case}}_validation.json` file. Here, filters are used to select only a subset of the integration tests at once in order to validate from simple cases, i.e., without preconditioner and default matrix format, to more complex cases, i.e., with preconditioner, other solver, and different matrix formats. The validation file can employ the `json-schema`[16] format, which to specify a rich set of criteria, e.g., checking if a numerical value is present and within a given range, to

[15] `https://github.com/features/actions`
[16] See `https://json-schema.org`

validate against. In summary, a workflow run is only considered to be successful if the compilation of OGL against different OpenFOAM versions succeeds, the converted and exported matrices pass a basic plausibility test, and the simulations executed with different solver properties run to completion with appropriate CFL number and continuity errors.

```yaml
case:
  type: OpenFOAMTutorialCase
  solver: icoFoam
  domain: incompressible
  case: cavity/cavity
    post_build:
    - controlDict:
      libs: [libOGL.so]
    - blockMesh
    - decomposePar:
      method: simple
      numberOfSubdomains: 2
      variation:
    - operation: fvSolution
      schema: "linear_solver/{solver}"
      values:
      - set: solvers/p
        preconditioner: none
        solver: GKOCG
        executor: reference
  ...
```

Listing 12. Example OBR case YAML file encoding a parameter study with different linear solver

```
source /root/OpenFOAM/${{inputs.path}}/etc/bashrc
mkdir ${{matrix.Case}} && cd ${{matrix.Case}}
obr init --config ${{matrix.Case}}.yaml
obr run -o generate
obr run -o runParallelSolver
obr status
obr query \
-q global -q continuityErrors -q CourantNumber \
--filter preconditioner==none \
--filter matrixFormat==Coo \
--filter global==completed \
--validate_against=${{matrix.Case}}_validation.json
```

Listing 13. Example OBR workflow including validation of results

5. Conclusion

This work emphasizes the critical role of systematic testing in developing robust research software within the OpenFOAM community. By examining and categorizing common testing types from the software engineering field, this paper provides a structured framework that supports testing in OpenFOAM projects. Key challenges specific to OpenFOAM, including complex dependencies, file-based setup, and limited automation capabilities, were identified. Addressing these challenges, this work introduced two tailored tools: the foamUT framework, which simplifies unit test integration, and the OpenFOAM Benchmark Runner (OBR), which facilitates setup and execution for comprehensive integration and performance testing.

The utility of these tools is illustrated through three case studies: the WENOExt library's direct integration of Catch2 for unit testing, the application of foamUT in a load-balanced adaptive mesh refinement tool, and OBR's application in benchmarking for the OpenFOAM Ginkgo Layer. Each case study demonstrates the practical value of these tools, showing how they address common barriers to testing and enable consistent, automated validation across OpenFOAM projects.

In conclusion, this work aims to elevate testing practices in OpenFOAM by lowering barriers to entry and providing tools that make testing both accessible and effective for developers. By supporting modular and automated testing workflows, this approach contributes to enhancing the reliability, maintainability, and scalability of OpenFOAM-based software. Future work could extend these tools to accommodate more advanced HPC environments, further automate testing workflows, and explore additional integration with existing HPC infrastructure.

Acknowledgements

Funding by DFG under grant number 390544712 is acknowledged with thanks.

Author Contributions: Conceptualisation, J.W.G., G.O. and M.E.F.; methodology, J.W.G., G.O. and M.E.F.; software, J.W.G., G.O. and M.E.F.; validation, J.W.G., G.O. and M.E.F.; formal analysis, J.W.G., G.O. and M.E.F.; investigation, J.W.G., G.O. and M.E.F.; resources, A.K., L.P., H.A., and H.M.; data curation, J.W.G., G.O. and M.E.F.; writing—original draft preparation, J.W.G., G.O. and M.E.F.; writing—review and editing, A.K., L.P., and H.M.; visualisation, J.W.G., G.O. and M.E.F.;; supervision, A.K., L.P., H.A., and H.M.; project administration, J.W.G.; funding acquisition, A.K., L.P. and H.M. All authors have read and agreed to the published version of the manuscript.

References

[1] T. Huckle and T. Neckel, *Bits and Bugs: A Scientific and Historical Review on Software Failures in Computational Science*. Philadelphia, PA: Society for Industrial and Applied Mathematics, Mar. 2019.
[2] G. Miller, "A Scientist's Nightmare: Software Problem Leads to Five Retractions," *Science*, vol. 314, no. 5807, pp. 1856–1857, Dec. 2006.
[3] M. Barker, N. P. Chue Hong, D. S. Katz, A.-L. Lamprecht, C. Martinez-Ortiz, F. Psomopoulos, J. Harrow, L. J. Castro, M. Gruenpeter, P. A. Martinez, and T. Honeyman, "Introducing the FAIR Principles for research software," *Scientific Data*, vol. 9, no. 1, p. 622, Oct. 2022.
[4] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and Software Technology*, vol. 56, no. 10, pp. 1219–1232, Oct. 2014.
[5] N. U. Eisty and J. C. Carver, "Developers perception of peer code review in research software development," *Empirical Software Engineering*, vol. 27, no. 1, p. 13, Oct. 2021.
[6] ——, "Testing research software: A survey," *Empirical Software Engineering*, vol. 27, no. 6, p. 138, Jul. 2022.
[7] H. Washizaki, Ed., *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0*. IEEE Computer Society, 2024.
[8] K. Naik and P. Tripathy, *Software Testing and Quality Assurance*. Hoboken, NJ: Wiley, 2008.
[9] L. Shunn and F. Ham, "Method of manufactured solutions applied to variable-density flow solvers," *Ann Res Briefs - Center Turbul Res*, 01 2007.
[10] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, 2005.
[11] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*, ser. The Addison-Wesley Signature Series. Addison-Wesley, 2010.
[12] P. M. Jacob and M. Prasanna, "A comparative analysis on black box testing strategies," in *2016 International Conference on Information Science (ICIS)*, 2016, pp. 1–6.
[13] K. Iglberger, *C++ Software Design: Design Principles and Patterns for High-Quality Software*. O'Reilly, 2022.
[14] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1100–1125, 2014.
[15] J. W. Gärtner, A. Kronenburg, T. Martin, and I. Shevchuk, "Weighted Essentially Non-Oscillating Scheme - WENO," 2022.
[16] M. E. Fadeli, "foamUT - a unit/integration testing framework for openfoam code." [Online]. Available: https://github.com/FoamScience/foamUT
[17] "MPI: A Message-Passing Interface Standard – Version 4.1," University of Tennessee, Tech. Rep., 2023.
[18] "OBR - openfoam benchmark runner." [Online]. Available: https://github.com/exasim-project/OBR
[19] J. W. Gärtner, A. Kronenburg, and T. Martin, "Efficient WENO library for OpenFOAM," *SoftwareX*, vol. 12, p. 100611, 2020.
[20] "blastFoam: A solver for compressible multi-fluid flow with application to high-explosive detonation." [Online]. Available: https://github.com/synthetik-technologies/blastfoam
[21] G. Olenik, M. Koch, Z. Boutanios, and H. Anzt, "Towards a platform-portable linear algebra backend for OpenFOAM," *Meccanica*, 2024.