Electronic Communications of the EASST
Volume 83 Year 2025

deRSE24 - Selected Contributions of the 4th Conference for Research Software Engineering in Germany

Edited by: Jan Bernoth, Florian Goth, Anna-Lena Lamprecht and Jan Linxweiler

System Regression Tests for the preCICE Coupling Ecosystem

Gerasimos Chourdakis, Valentin Seitz, Benjamin Uekermann

DOI: 10.14279/eceasst.v83.2614

License: This article is licensed under a CC-BY 4.0 License.

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

System Regression Tests for the preCICE Coupling Ecosystem

Gerasimos Chourdakis[1 2], **Valentin Seitz**[2], **Benjamin Uekermann**[1]

[1]Institute for Parallel and Distributed Systems, University of Stuttgart
[2]School of Computation, Information and Technology, Technical University of Munich
gerasimos.chourdakis@tum.de

Abstract: Simulation codes are often implemented as one single application or library, which often clearly defines the system under testing and an oracle for regression tests. For some simulations, coupling multiple codes is necessary. In the case of the coupling library preCICE, a realistic system of at least two coupling participants (typically each a different simulation code) consists not only of the library, but of a complete software stack of components underneath each coupling participant. Each component is developed in a separate software repository, and the numerical and screen output often vary between executions or dependency versions. These make preCICE an interesting study case for system regression tests, requiring solutions to issues not typically faced by stand-alone research software projects. In this paper, we relate the challenges faced when testing the preCICE ecosystem to the testing approaches of other projects, and we present a new framework for system regression tests for preCICE, based on connected Docker containers with cached layers, version control of and numerical comparisons to reference results, and interaction of GitHub Actions workflows across repositories.

Keywords: software testing, system testing, regression testing, multi-physics

1 Introduction

Over the past decades, the scientific community has developed specialized simulation software for a wide range of phenomena, at a wide range of scales. One of the main challenges since longer is multi-physics and multi-scale simulations, such as the fluid-structure interaction in aerolasticity, ocean-atmosphere dynamics in geophysics, or magnetohydrodynamics [KMW+13]. Instead of developing new, even more specialized software for specific multi-physics and multi-scale simulations, the coupling library preCICE[1] (like other solutions) allows scientists and engineers to compose multi-physics and multi-scale simulations, reusing the codes they already have at hand [CDR+22]. They can do that either by directly calling the API of preCICE, or using some of the ready-to-use *adapters* (part of the *preCICE ecosystem*), as discussed in Section 2.

As a library that enables research, preCICE needs to be continuously and reliably tested, with test suites (collections of tests) covering all the critical features. Different levels of testing [BF14] are particularly interesting for preCICE: static (linting), unit, integration, system, and regression tests. The core library is already extensively tested with static, unit, and integration tests, and the same holds for some of the language bindings and adapters [CDR+22, SCU22]. For testing

[1] preCICE website: https://precice.org/, available under LGPLv3 on GitHub: https://github.com/precice/precice/

the complete system (in particular before each release), developers currently run selected test cases (complete coupled simulations that serve as tests) manually and compare results to their expectations, as it is common across research software [KB14]. While several test cases are already available as user tutorials[2] or as further published validation cases (with examples of user applications discussed in the preCICE reference papers [BLG+16, CDR+22]), automating these and tracking any regressions of simulation results is particularly challenging due to the very nature of a coupling library, as discussed in Section 3. Early implementations of such system tests for preCICE [Hos18, Lin19, CDR+22] demonstrated several such challenges at scale.

In Section 2, we dive deeper into the preCICE ecosystem and its development workflows. Based on these, we identify the unique testing challenges that partitioned simulations pose in Section 3, and we study how other approaches fit into these challenges in Section 4. We then define our expectations for a better solution in Section 5 and describe the recently implemented solution in Section 6, before demonstrating it in action in Section 7. Finally, we conclude and present ideas for future research in Section 8.

2 The preCICE partitioned simulation ecosystem

To set up a multi-physics simulation with preCICE, a user needs to start two or more *coupling participants* (see Figure 1). Each participant is a simulation code that uses the preCICE API and links to the preCICE library. A user then needs to start each participant as a different process, which then find each other and communicate through the local or wider network.

The first release of preCICE [BLG+16] introduced this core library, written in C++, which offers efficient communication methods (over TCP sockets or MPI ports), several data mapping algorithms (including RBF-based interpolation methods), and advanced numerical coupling algorithms (such as Anderson acceleration). The second major release of preCICE [CDR+22] included several additional components (such as ready-to-use adapters for several open-source simulation codes) that made it possible to construct such simulations often without writing any additional code. These adapters [UBC+17] are either modified versions of the respective simulation code, wrapper classes, or stand-alone plug-ins that implement calls to the preCICE API.

Today, codes coupled to preCICE can be written in C++, C, Fortran, Python, Julia, Matlab, or Rust. Ready-to-use adapters are available for codes such as OpenFOAM, deal.II, FEniCS, DUNE, DuMux, SU2, CalculiX, and more (Figure 2). Tools such as the Micro-Manager or the FMI Runner allow multi-scale simulations and coupling to system codes. A collection of currently 31 tutorial cases serves as concrete simulation test cases for these components, with many of the test cases being available for multiple combinations of simulation codes. All of these components are developed openly on GitHub[3], in more than 40 repositories. The documentation is sourced from several repositories and rendered in one place: the preCICE website[4].

The design decision of using individual repositories instead of a monorepo was primarily dictated by the fact that each adapter is mainly an extension of each simulation code, and not of the preCICE library. This decision allows each repository to use a different license, build system, CI

[2] preCICE tutorials: https://precice.org/tutorials.html

[3] preCICE organization on GitHub: https://github.com/precice/

[4] preCICE documentation: https://precice.org/docs.html

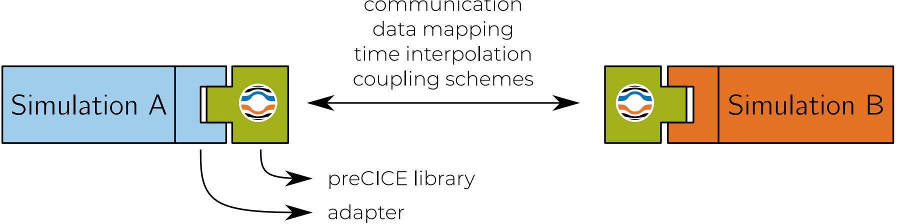

Figure 1: Overview of preCICE: Two or more single-physics simulation codes start normally, each in each own process, and they link to the preCICE library. The API calls, together with any additional glue code, are called an "adapter", which can be part of the simulation code, an additional class, or a stand-alone package. The preCICE library handles the communication between the simulation codes, space and time interpolation of data, and provides coupling algorithms.

workflow, versioning scheme, and release schedule. It has also made it easier for students and external researchers to contribute to individual components, minimizing the need for synchronization and the impact of decisions and actions to the rest of the codebase, and making academic attribution fine-grained. However, this separation of concerns requires additional metadata to define a complete and compatible software stack. This problem is currently addressed by the preCICE distribution[5] and is the subject of a running research project[6].

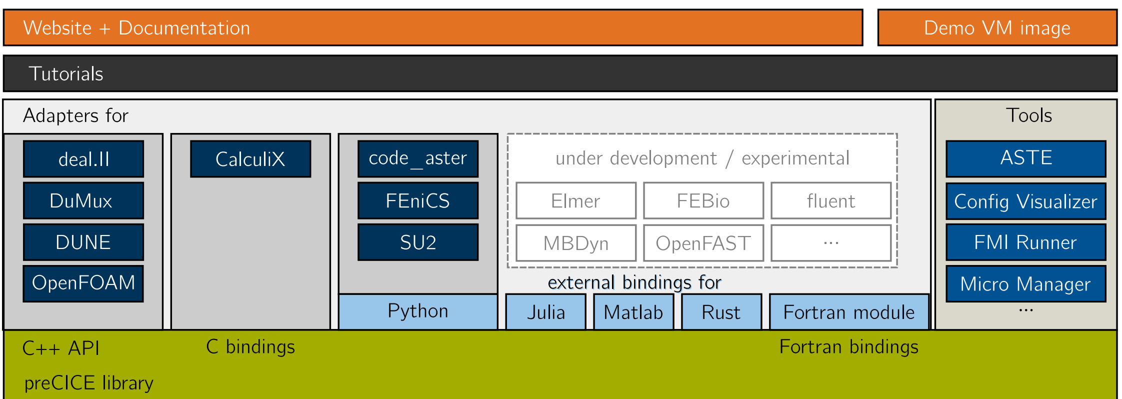

Figure 2: Overview of the preCICE ecosystem as a software stack. Every box represents a separate component, hosted in a separate software repository. The simulation codes (e.g., OpenFOAM) are not part of the preCICE ecosystem, but the adapters for these codes are. All released components, the simulation codes, and additional tools, are included in the Demo Virtual Machine image (an Ubuntu-based Vagrant box).

[5] preCICE distribution: https://precice.org/installation-distribution.html. Latest version: v2404 [CCD+24].
[6] DFG project preECO: https://gepris.dfg.de/gepris/projekt/528693298?language=en

3 Challenges

Research software is often designed as one stand-alone application, developed in one software repository. In such cases, a system test is the complete execution of this one code. A regression test compares the output of this run to the output of previous runs, with the results format being controlled by the application developers. Challenges we have faced in several projects include reproducible and/or parallel runs, scaling up the number of tests, and tracking the history of regressions, while several other common challenges are listed in literature [BF14, KB14]. In performance regression tests, consistency and running on HPC systems become challenging as well. In the case of preCICE, the aspects of the ecosystem discussed in Section 2 bring additional challenges, which we identify in the following.

3.1 Simulation codes call preCICE as a library

Since preCICE is a library, other codes call it, often via additional layer (adapters and language bindings). A system test needs to start the third-party simulation codes, which will then call any intermediate layers, before calling preCICE. In research software applications, it is often enough to start the application itself, which would automatically call any dependencies.

3.2 Testing requires at least two simulation participants

Due to the very nature of partitioned simulations, each test representing a complete system requires starting at least two coupling participants. In the case of preCICE, this means starting two different processes, even if we only need to test the interaction between these participants. We can simplify these systems using mock objects [BF14], but these do not reveal all the numerical, communication, or other technical issues that arise in an actual simulation.

3.3 Each participant uses multiple components

As we develop not only one library, but a complete collection of components that depend on each other and on third-party codes, every coupling participant will involve multiple components of the ecosystem (core library, language bindings, one adapter, specific tutorial setup). We need to explicitly specify compatible versions of each component in a test, which becomes particularly challenging in case of a breaking release in the core library (which brings non-backwards compatible changes in the API and the configuration). Dependency management becomes an issue as well, as the multiple layers might depend on the same shared libraries (such as MPI).

3.4 Multiple repositories, multiple perspectives

Since the various components are developed in different repositories by different people, the question often arises “what is the component under test?”. As discussed in the preCICE v2 reference paper [CDR+22], stakeholders include the release manager, the library developers, the adapter developers, the tutorial contributors, and the developers of the testing infrastructure themselves. The release manager is interested in knowing if all development branches of all components work together and are ready to be released as a new preCICE distribution. For an

adapter developer, important is if a contribution to that adapter works with compatible branches of every other component . Tests need to be triggered across repositories and the results need to be visible at the repository that requested the tests.

3.5 Long building and running time

Every test requires multiple components to form a simulation stack and rebuilding one component of the stack often means rebuilding the components that depend on it as well. For this reason, caching of dependencies becomes essential. As is common in numerical simulations, execution itself also takes significant time. As an example, one test fluid-structure interaction simulation involving OpenFOAM and deal.II requires installing preCICE, OpenFOAM, and deal.II, building the OpenFOAM and the deal.II adapters, getting the tutorials repository, and executing a simulation for at least several minutes.

4 Examples from the literature

Various other projects incorporate system testing in their continuous integration pipelines. We present here examples from projects that partially share the same challenges. In Subsection 4.1, we look into the library deal.II, as an example of established good practices in research software. In Subsection 4.2, we look into software with similar goals as preCICE: the multiscale universal interface (MUI) and the model order reduction library pyMOR. In Subsection 4.3, we look into the ecosystems of scientific software installations EESSI and E4S, as well as the xSDK Examples, as they also combine multiple components in their tests. Finally, in Subsection 4.4, we discuss previous approaches to creating system tests for preCICE.

4.1 Example 1: deal.II

deal.II [ABD+21] is an open-source C++ library that allows users to create their own simulation codes, discretizing their models using the finite elements method. It is also developed in academia, and it also provides multiple tutorial cases that need significant time to build and run. deal.II is developed primarily in a single repository, with its complete test suite defined in CMake, and the test state visualized in a dashboard[7]. The example setups are available in the same repository and reference screen output is stored next to each test case configuration[8]. For regression testing, the screen output is compared to the reference as text, a comparison susceptible to floating point errors. In addition to regression tests, deal.II also includes automatic performance tests[9]. CI workflows are executed both on GitHub Actions and on a Jenkins server. Automated tests are also executed for the deal.II code gallery, a repository including more complex, community-contributed examples[10].

This is a good example of current testing practices in numerical research software. Compared to deal.II, the core library of preCICE applies many similar practices, and the regression tests of

[7] deal.II CDash: https://cdash.dealii.org/
[8] The deal.II Testsuite: https://dealii.org/developer/developers/testsuite.html
[9] deal.II performance tests: https://dealii.org/performance_tests/reports/render.html?#!current.md
[10] deal.II code gallery tests: https://github.com/dealii/code-gallery/blob/master/.github/workflows/linux.yml

deal.II correspond to the integration tests of the preCICE library [CDR+22]. However, deal.II does not face the challenge of requiring multiple participants, each using multiple components, and developed in different repositories. Both libraries still rely on multiple other libraries as well, each potentially introducing regressions with each new version, but this is a common issue, outside the scope of this paper.

4.2 Example 2: MUI and pyMOR

MUI [TKB+15][11] is a project with similar goals as preCICE, which does not currently provide ready-to-use adapters for simulation codes. The tests included in the MUI test suite only test the core library, similarly to the integration tests of the preCICE core library[12].

Another example is the Python-based code pyMOR [MRS16][13], which also maintains integrations to simulation codes (FEniCS, deal.II, NGSolve). The deal.II integration is hosted in a separate repository, while the integrations to FEniCS and NGSolve (both offering a Python API) are examples in the main repository. pyMOR itself uses pytest for its test suite, while the deal.II integration is tested directly in its repository, using a tailored Docker image[14].

4.3 Example 3: EESSI, E4S, and the xSDK Examples

The European Environment for Scientific Software Installations [DHH+23] maintains a shared repository of scientific software installations that can be used on various systems, focusing mainly on HPC sites[15]. As a collection of software packages, it serves as another ecosystem to compare to, breaking the boundaries of testing a single application. It includes a test suite[16] based on the ReFrame testing framework [KMR+20]. While ReFrame is a general framework, it mainly facilitates interacting with HPC systems, providing tools to write portable test cases. The challenges it solves are different than the challenges discussed in Section 3, and the two systems could be used together in the future.

Despite the complexity of this project, the challenges are again different: multiple components are indeed combined in a stack, but with lower component coupling as in the case of the preCICE ecosystem. The components (a filesystem, a package manager, and more) run as services or stand-alone packages, and they are developed independently to each other.

Similar testing practices are applied by the Extreme-scale Software Stack (E4S)[17], based on in-house tools to test and deploy the software stack on various HPC systems.

A similar example is the xSDK Examples, providing test cases for various components of the extreme-scale scientific software development toolkit[18]. Every example uses CMake as a build system and defines tests using CTest. A bundle of all components is available via Spack, a

[11] MUI website: http://mxui.github.io/
[12] MUI test suite: https://github.com/MxUI/MUI-Testing
[13] pyMOR: https://pymor.org/
[14] pyMOR-deal.II CI workflow: https://github.com/pymor/pymor-deal.II/blob/main/.ci/job-template.yml
[15] EESSI project: https://www.eessi-hpc.org/
[16] EESSI test suite: https://github.com/EESSI/test-suite, based on ReFrame: https://reframe-hpc.readthedocs.io/
[17] E4S: https://e4s-project.github.io/, testing dashboard: https://dashboard.e4s.io/
[18] xSDK: https://xsdk.info/, SDK Examples: https://github.com/xsdk-project/xsdk-examples

requirement enforced by xSDK. Both of these aspects (uniform build system and package versioning/distribution) are not possible for the preCICE ecosystem, since the preCICE developers have no control over the build system and distribution of each simulation code.

4.4 Previous preCICE system tests

Previously [Hos18, Lin19, CDR+22], preCICE had an earlier prototype of system tests. That was based on Python scripts building and publishing Docker images, one for each test. The scripts and test specifications were hosted in a dedicated repository[19], which was connected to the Travis CI platform. For caching, Docker images were published on Docker Hub[20]. For regression checking, the scripts were performing hash comparisons on the result files and text comparisons on the logs. A separate repository was used to archive reports of the test runs[21].

This solution proved to be difficult to maintain and to scale with the increasing number of components and test cases, even after applying some of the ideas presented in Section 6 [CDR+22], and it was ultimately abandoned after changes in the Travis CI pricing model in 2020. The system tests aimed to test a large cross-product of combinations of operating system, compiler and build options for preCICE, installation methods of preCICE, and simulation code combinations, leading to combinatorial explosion. Most of these tasks were eventually offloaded to the CI of the core library. They also aimed to check every kind of output, which meant maintaining several in-house checking scripts, often needing to update the reference files or temporarily mark individual tests as allowed to fail.

5 Design objectives

Having already looked at similar projects and having learned from previous attempts to design system tests for preCICE, we define here our expectations from a better solution. These aim to prevent issues faced mainly in the previous work discussed in Subsection 4.4 and in the current state of (manual) testing at the project.

Open: In the current practice of the project, developers test release candidates by executing simulations attached to research projects. These are often not publicly available at the time of testing, and the exact testing procedure depends on the individual developer. This involves trust and is prone to human error. To prevent that, the testing infrastructure should be developed in the open and test reports should be public. The configuration also serves as user documentation and the public reports increase transparency and provide continuous evidence for the maintenance status of the project. Wherever possible, test cases should also be maintained as tutorials, in the tutorials repository.

Extensive coverage: The tutorials that users run as examples also serve as test cases. However, in the current state of the project, only a small subset of the tutorials is tested, as these are manually executed. This often leads to uncaught issues either in specific features, or in

[19] Previous preCICE system tests: https://github.com/precice/systemtests
[20] preCICE Docker Hub: preCICE Docker Hub account: https://hub.docker.com/u/precice
[21] Reports archive of the previous preCICE system tests: https://github.com/precice/precice_st_output

specific tutorials. In a better solution, every code and tutorial that the user interacts with should be part of the test suite, minimizing the blind spots for the developers.

Easy test case generation: Adding tests is often significant additional effort. At the same time, preCICE users often share their simulation configurations in various ways, and these could serve as test cases. If a tutorial or other published case setup is already available, integrating that to the system tests should require minimal additional work and no duplication.

Ability to run non-public tests and codes, on different systems: In simulation software, code or data is sometimes only available under a restrictive license. In some cases, the simulations need extensive resources, special hardware, or libraries only available on specific systems. Therefore, the tests should be able to run on different systems. This does not conflict with the first objective (openness), as everything is by default open, but the infrastructure should also handle non-open cases, whenever restrictions apply.

Avoid vendor lock-in: The previous attempt to create system tests for preCICE was ultimately terminated by a policy change of the CI provider. Therefore, the tests should be easy to execute on or port to different CI providers, to avoid a vendor lock-in. Similarly, the solution should not rely on external resource providers (such as run servers or container registries) that might impose usage limits beyond our control[22].

Report results where expected: In the previous work, connecting the test reports with the state of the code tested was particularly complicated. In a better solution, reports should be easily discoverable (e.g., in the same pull request in an adapter repository, or in a subdirectory in a local run). Looking at the history, it should be clear which tests failed when, and where the full details are.

Avoid unnecessary rebuilds: Running a test for one component often means rebuilding many other components of the stack as well. In the previous work, this meant rebuilding the complete test stack for every test, even if only a small change in one component was introduced. The testing infrastructure should implement caching mechanisms to avoid unnecessary rebuilds of components already available.

Maintain as little code as possible: As we learned from the previous preCICE system tests, it is difficult to maintain test-specific run scripts or in-house output checking scripts for file formats that we do not control. A better solution should rely on established dependencies and stable file formats, follow modularity, and avoid duplication.

[22] It should be noted here that this is not a costs-saving policy, but rather a way to avoid the common organizational overhead of paying for external services in academia.

6 Proposed solution

We propose a framework for testing the preCICE ecosystem that addresses the challenges discussed in Section 3 and fulfills the design objectives listed in Section 5. We begin with an overview of the overall architecture, and we dive deeper into specific ideas and technical details in the following subsections. We demonstrate a concrete example with the respective specification files in Section 7.

6.1 Overview

The framework comprises YAML-based specification files, Python scripts, and Dockerfiles, stored directly in the preCICE tutorials repository[23]. This makes the tests easy to keep in synch when adding or updating tutorial cases, and facilitates using tutorials as test cases, as well as publishing test cases as tutorials. It also makes the tests and reference results directly available to every user and developer from the same repository, allowing to easily run and modify them. Specification files are stored in the directory of each tutorial, providing a machine-readable description of which coupling participants are available and which components are required[24]. The specification files define components (e.g., adapters), tests that depend on these components, and test suites that group tests together. The scripts filter the tests, prepare and run Docker containers, compare the results, and archive reports. Figure 3 shows an overview of the architecture. The documentation of the system tests is publicly available[25].

6.2 Containerization and caching

To execute tests on a controlled and portable environment, we use Docker. Instead of building one Docker image with every component required, as is common when testing single applications, we build one Docker container per coupling participant, and connect them using Docker Compose[26] (in which case, each container is a different service). This means that, to build one component, we do not need to build other components that are not dependencies for it, saving building time. An additional service provides the requested version of the tutorials repository. After the simulation is done, a last service handles the comparisons to the reference results.

To avoid rebuilding upstream components that were not changed since previous tests, we use a single, multi-stage Dockerfile[27], describing the dependency graph of the ecosystem. A multi-stage Dockerfile specifies named stages, which work similarly to intermediate images: They can be defined as base images for other images, and we can copy files from them. This delegates the caching to Docker itself, provided that the consecutive tests are running on the same system. To take advantage of this feature, and to reduce the CI resource usage, the GitHub Actions workflow executes the tests always on the same, self-hosted virtual machine (instead of directly using the provided GitHub runners), which caches the Docker layers.

[23] preCICE system tests scripts: https://github.com/precice/tutorials/tree/v202404.0/tools/tests
[24] An example of this metadata is later shown in Listing 3.
[25] preCICE system tests documentation: https://precice.org/dev-docs-system-tests.html
[26] Docker Compose: https://docs.docker.com/compose/
[27] Multi-stage Docker builds: https://docs.docker.com/build/building/multi-stage/

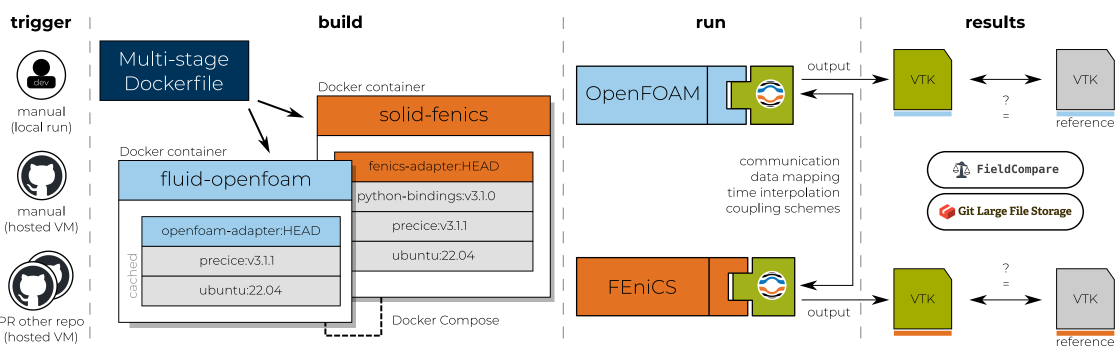

Figure 3: Overview of the system tests. A user triggers the system tests either locally or on GitHub, or via a pull request on another, authorized repository on GitHub. A local run means directly running the script `systemtests.py`. A run on GitHub Actions runs the same script, and it always runs it on the same, self-hosted VM, for easier caching. Depending on the components needed for a test (for this test, direct dependencies are the OpenFOAM and FEniCS adapters), the system tests build Docker containers and connect them as Docker Compose services, reusing any available cached layers. The participants of the coupled simulation get coupled via preCICE. For testing purposes, preCICE outputs VTK files with the data values on the interface, which are then compared against reference results (stored in Git LFS) using fieldcompare. The results comparison step can easily be extended to also compare other files in standard formats.

Since all preCICE components use Git as a version control system, it is important for caching that we checkout specific versions (commits or tags) of each component. Checking out the latest development branch will be detected as a (false) cache hit, since a Git branch does not correspond to a fixed version. This issue can be resolved by automatically determining the latest commit of each branch, before triggering the tests.

6.3 Local and CI runs

Tests can be executed locally or on the CI (manually, or triggered by other repositories). The entry point of the system tests is the script `systemtests.py`, which takes as arguments the names of the test suites to execute and a string of build arguments (mainly including which versions of each component to use). Internally, this script calls `docker build` and `docker compose`, redirects the screen output to a different log file per service, requests the comparison of the results to the references, prepares a report, and archives artifacts.

The same script is executed in the CI of the tutorials repository via GitHub Actions. This is triggered either manually (via the GitHub web interface or the GitHub CLI), or by other authorized repositories (by adding a `trigger-system-tests` label to a pull request). As the CI is only triggering a script (i.e., only the minimally required steps are included in the CI specification), we minimize the vendor lock-in, and we make the tests easier to develop and debug. An alternative approach would have been to use act[28] to run the GitHub Actions locally, but such an

[28] act: https://github.com/nektos/act

approach would again increase the dependence on a specific CI resource provider.

6.4 Numerical comparisons

We learned already that trying to compare every kind of output (and especially as text) is a moving target and requires additional comparison scripts to be developed, for output that might be changing values and format between runs or over different versions of dependencies. Instead, we have reduced the oracles to just the values of the exchanged data on the coupling interfaces. As the tests we are executing are transient PDE simulations, even minor numerical errors in the middle of the domain will cause disturbances to the coupling interfaces over time. These values can be written by preCICE in a stable and standard VTK format[29]. We then numerically compare these results to their references using fieldcompare[30] [GKP+23], allowing for floating-point errors. fieldcompare displays statistics about the files failing the comparisons, and it can also export differences between the two meshes, to facilitate locating the source of errors.

In the future, we also want to archive and compare additional results about each simulation, such as the number of coupling iterations and the respective residuals. These are also written by preCICE as CSV files, and the framework can easily be extended to compare more standard files, such as iteration counts and residuals from a CSV file.

6.5 Storing and versioning reference results

Reference results are generated by the testing framework on the test system, based on requested versions of each component. In order to track changes, the results are stored as part of each tutorial, in the same repository, using the Git Large File Storage system (LFS)[31], which uploads the files to a Git LFS server and stores identifiers instead of the files themselves in the source repository. This allows the reference results to be updated in the same pull request that updates a tutorial, without having to implement any additional versioning, and without polluting the source repository with automatically generated artifacts. To work around resource limitations, and to minimize vendor lock-in, we host a Git LFS server ourselves, using soft-serve[32].

As users use tutorials as starting points to build their own simulations or couple their own codes, a side effect of storing the reference results in the same directory is that users can easily compare to them. This is particularly useful in the context of the replay mode of the artificial solver testing environment, which can feed pre-recorded data to a coupling participant[33].

6.6 Reporting and archiving

Each test job is executed in a time-stamped directory and redirects all screen output to log files. At the end of a run tiggered by the CI, the complete run folder is archived and uploaded as a build artifact, listed in the GitHub Actions summary. This includes a report summarizing the versions of each component used, system information, and more metadata important for reproducibility,

[29] preCICE exports: https://precice.org/configuration-export.html
[30] fieldcompare: https://gitlab.com/dglaeser/fieldcompare
[31] Git LFS: https://git-lfs.com/
[32] soft-serve: https://github.com/charmbracelet/soft-serve
[33] ASTE: https://precice.org/tooling-aste.html

including a snapshot of the testing scripts themselves. Archives are directly related to the CI job that executed the tests, and easily discoverable from the respective pull request, together with the respective success status.

7 Demonstrating example

We present here an example of a system test, demonstrating the modularity and flexibility of the testing specification. We first describe the structure of a test case. We then describe a component, we define the test, and add it to a test suite. We trigger this test suite locally, and we finally trigger this test suite from a GitHub pull request. As an example, we will use the already integrated flow over a heated plate tutorial[34]. An overview of the files discussed in this section is shown in Listing 1. More details are provided in the documentation[35].

```
1  flow-over-heated-plate/                              # Subsection 7.1, Listing 2
2    - <simulation configuration files>
3    - metadata.yaml                                    # Subsection 7.1, Listing 3
4    - reference-results/                               # Subsection 7.1
5      - fluid-openfoam_solid-fenics.tar.gz
6      - <further reference results>
7
8  tools/tests/
9    - components.yaml                                  # Subsection 7.2, Listing 4
10   - component-templates/
11     - openfoam-adapter.yaml                          # Subsection 7.2, Listing 4
12     - <further component templates>
13   - dockerfiles/ubuntu_2204/
14     - Dockerfile                                     # Subsection 7.2, Listing 5
15   - tests.yaml                                       # Subsection 7.3, Listing 6
16   - systemtests.py                                   # Subsection 7.4
17   - <further system tests scripts>
18
19 .github/workflows/                                   # Subsection 7.5, Listing 7
20   - run_testsuite_workflow.yml
21   - run_testsuite_manual.yml
22   - generate_reference_results_workflow.yml
23   - generate_reference_results_manual.yml
```

Listing 1: Overview of files discussed in the demonstrating example. All files are available on https://github.com/precice/tutorials/tree/v202404.0.

[34] Flow over a heated plate tutorial: https://precice.org/tutorials-flow-over-heated-plate.html

[35] System tests documentation: https://precice.org/dev-docs-system-tests.html

7.1 Structure of a preCICE tutorial

Every preCICE tutorial follows a structure described in the contributing guidelines[36], with an example shown in Listing 2. Each tutorial has two or more participants (one for each subdomain, in this case a `Fluid` and a `Solid`), and one or more simulation code options for each participant (in this case, `Fluid` can be OpenFOAM or SU2, while `Solid` can be OpenFOAM, FEniCS, Nutils, or Dune-Fem). Each tutorial directory contains a `README.md` file, a preCICE configuration file, visualization and cleaning scripts, images, and one configuration directory for each `participant-code` test case. In addition to these, each tutorial contains a `metadata.yaml` file and a `reference-results/` directory. This `metadata.yaml` provides a machine-readable description of the tutorial, as shown in Listing 3. Reference results are named after the specific combination of codes (e.g., fluid-openfoam and solid-fenics) that produced them and include an automatically-generated description file regarding the setup that generated them.

```
1  flow-over-heated-plate/
2    - README.md              # Description of the case
3    - precice.config.xml     # A works-with-all preCICE configuration file
4    - fluid-openfoam/        # A concrete case for the Fluid participant
5      - run.sh               # A short script to run the solver1 case
6      - <OpenFOAM config>
7    - fluid-su2/
8    - solid-dunefem/
9    - solid-fenics/
10   - solid-nutils/
11   - solid-openfoam/
12   - metadata.yaml          # Metadata of the tutorial
13   - reference-results/     # Results for various combinations
```

Listing 2: Excerpt of the file tree of the flow-over-a-heated-plate tutorial

```
1  name: Flow over heated plate      # Human-readable name
2  path: flow-over-heated-plate      # Where this test case is located
3
4  participants: ['Fluid', 'Solid'] # List of coupling participants
5
6  cases:                            # List of simulation cases (options)
7    fluid-openfoam:                 # Arbitrary test case name
8      participant: Fluid            # Which participant this corresponds to
9      directory: ./fluid-openfoam  # Directory providing the configuration
10     run: ./run.sh                 # Commands to start the participant
11     component: openfoam-adapter   # Direct component dependency of this case
```

Listing 3: Excerpt of a `metadata.yaml` of a tutorial (test case)

[36] preCICE contributing guidelines: https://precice.org/community-contribute-to-precice.html#contributing-tutorials

7.2 Components

Components are defined in a `components.yaml` file. Each component relates to a Docker Compose service, and defines a Jinja-based template[37] that the system tests use to generate a `docker-compose.yaml` configuration. See Listing 4 for an excerpt of these files. Docker Compose uses the multi-stage Dockerfile (Listing 5), and each component requests a specific stage of that multi-stage build, which depends on different previous stages.

```
# Excerpt of the tutorials/tools/tests/components.yaml:
openfoam-adapter:
  repository: https://github.com/precice/openfoam-adapter
  template: component-templates/openfoam-adapter.yaml
  build_arguments:         # Arguments needed to build the service
    PRECICE_REF:           # ... Version of preCICE to use (a commit/tag)
    PLATFORM:              # ... Dockerfile platform used (e.g., ubuntu_2204)
    TUTORIALS_REF:         # ... Tutorial git reference to use
    OPENFOAM_ADAPTER_REF: # ... OpenFOAM adapter git reference
    OPENFOAM_EXECUTABLE:  # e.g., openfoam2312

# tutorials/tools/tests/component-templates/openfoam-adapter.yaml:
build:
  context: {{ dockerfile_context }} # Dockerfile to use
  args:     # Parse arguments from the components.yaml
  target: openfoam_adapter # Dockerfile stage to build
volumes:   # Working directory (owned by the current user)
  - {{ run_directory }}:/runs
command: > # Commands to execute inside the container for this service
  /bin/bash -c "id && cd '/runs/{{ tutorial_folder }}/{{ case_folder }}' &&
  openfoam {{ run }} | tee system-tests_{{ case_folder }}.log 2>&1"
```

Listing 4: Definition of the openfoam-adapter component and Docker Compose service template

```
# Excerpt of the tutorials/tools/tests/dockerfiles/ubuntu_2204/Dockerfile
FROM ubuntu:22.04 as base_image
# ... (define stages base_image, precice_dependencies, precice)

FROM precice_dependecies as openfoam_adapter
# ... (install OpenFOAM, provide version-specific OPENFOAM_EXECUTABLE)
COPY --from=precice /home/precice/.local/ /home/precice/.local/
WORKDIR /home/precice
# Commands to get and build the specific version of the component
RUN git clone https://github.com/precice/openfoam-adapter.git && \
    cd openfoam-adapter && git checkout ${OPENFOAM_ADAPTER_REF} && \
    /usr/bin/${OPENFOAM_EXECUTABLE} ./Allwmake -j $(nproc)
```

Listing 5: Multi-stage Dockerfile, highlighting the openfoam_adapter stage

[37] Jinja: https://jinja.palletsprojects.com/

7.3 Tests and test suites

With test cases at hand (via tutorials with a `metadata.yaml`), we can define actual tests and group them into test suites in one central `tests.yaml` file (Listing 6). Remember that each tutorial can be executed with a cross product of different participant options. A *test* specifies which particular combinations of codes (test cases) will be coupled (which exactly will the system under testing be) and which reference results will be used for regressions. Each test is then included in one or more *test suites*. A test suite `openfoam_adapter_pr` would only execute tests relevant for every pull request to the OpenFOAM adapter, while a `release` test suite would include more tests and run for longer time.

Having a central file provides a clear overview of which tests will be executed when, and makes pre-processing easier. At the moment, test cases can only be defined in the tutorials repository, but this will be extended to allow sourcing test cases from different repositories (e.g., publication datasets or simulation setups with restrictive licenses).

```
1  # Excerpt of the tutorials/tools/tests/tests.yaml
2  test_suites:                   # A list of all test suites
3    openfoam_adapter_pr:         # Arbitrary name for a test suite
4      tutorials:                 # Test case repository to look into
5        - path: flow-over-heated-plate # Directory inside the repository
6          case_combination:               # Which code to execute for each
7            - fluid-openfoam              # participant
8            - solid-openfoam
9          reference_result: "./flow-over-heated-plate/reference-results/\
10                            fluid-openfoam_solid-openfoam.tar.gz"
11   release_test:                # A longer test suite (...)
```

Listing 6: Definition of tests and test suites

7.4 Running locally

We can run the `release_test` test suite locally on a laptop. This only requires Python and Docker. To ensure that the right versions are tested, we have to specify them as build arguments:

```
python systemtests.py --build_args=PRECICE_REF:v3.1.1[,...] --suites=[...]
```

A concrete example using the versions described in the preCICE Distribution v2404.0 is:

```
python systemtests.py
  --build_args="PRECICE_REF:v3.1.1,PYTHON_BINDINGS_REF:v3.1.0,
                OPENFOAM_ADAPTER_REF:v1.3.0, FENICS_ADAPTER_REF:v2.1.0,
                SU2_VERSION:7.5.1, SU2_ADAPTER_REF:64d4aff,
                TUTORIALS_REF:v202404.0"
  --suites=release_test
```

Having to provide exact versions for all components is definitely non-ergonomic, but it is currently necessary to avoid false cache hits. A potential solution is a preparation step mapping from branches to specific commits, as in the `gather-refs` step in the GitHub Actions workflow discussed next.

7.5 Triggering on GitHub Actions

The GitHub Actions workflow `run_testsuite_workflow.yml` of the tutorials repository collects inputs (such as the versions to test), checks out the tutorials repository for a given Git reference, executes the system tests as if running locally (in a hosted Linux runner), and uploads the logs and results as build artifacts. A separate workflow, `run_testsuite_manual.yml` can trigger the main workflow manually. This separation allows that the main workflow can be triggered by other repositories, which must include an access token in their secrets. An example is shown in Listing 7, where the CI of the OpenFOAM adapter is triggering the system tests in the tutorials repository[38]. To avoid running the rather long system tests on every commit, the tests are only triggered every time the `trigger-system-tests` label is added on the pull request. Reference results are generated on the same system, using the corresponding workflows `generate_reference_results_workflow.yml` and its manual equivalent. These run the tests, commit the results to the repository, and upload them to the Git LFS server.

```
1  # Excerpt from openfoam-adapter/.github/workflows/system-tests.yaml
2  on: pull_request: types: [labeled]
3
4  jobs:
5    gather-refs:
6      # Get the Git reference of the current tutorials develop branch
7      # using the nmbgeek/github-action-get-latest-commit action (...)
8
9    run-system-tests:    # Execute when a label is added to the PR
10     if: ${{ github.event.label.name == 'trigger-system-tests' }}
11     needs: gather-refs # First wait for the gather-refs job to complete
12     uses: precice/tutorials/.github/workflows/run_testsuite_workflow.yml@dev
13     with:                # Arguments for the reusable workflow
14       suites: openfoam_adapter_pr
15       build_args: TUTORIALS_REF:${{needs.gather-refs.outputs.reftutorials}},
16         PRECICE_REF:v3.1.1,OPENFOAM_EXECUTABLE:openfoam2312,
17         OPENFOAM_ADAPTER_REF:${{ github.event.pull_request.head.sha }}
```

Listing 7: Triggering the system tests from another repository. The workflow described in `run_testsuite_workflow.yml` executes the same `systemtests.py` script, with the arguments passed to it.

[38] Example PR that has triggered this workflow: https://github.com/precice/openfoam-adapter/pull/333

8 Conclusion and future work

Having compared testing practices of partially similar projects with the needs of preCICE, a coupling ecosystem of components developed in different repositories, we can conclude that testing preCICE poses additional challenges than usual, and serves as an interesting study case for system and regression testing. We derived design objectives from these challenges and the comparison to related work. The presented solution fulfills all the design objectives, which previous approaches only partially fulfilled. The testing infrastructure is well-integrated into the same examples that every user sees. Every tutorial with metadata can be considered as a test by editing only one file. Components can be sourced by arbitrary Git repositories, while the system can be extended to support test cases from external repositories. The tests can run on any Docker-enabled system and use only established free/open-source dependencies, while the code needed to integrate into GitHub Actions or any other CI provider is minimal, avoiding vendor lock-in. Results are reported via the screen output and log files, while the GitHub Actions workflows archive them as part of the respective test job. Caching of Docker layers minimizes the time needed to re-run tests and, because of the multi-stage Docker builds and the separation of one container per coupling participant, the cache can be effectively managed directly by Docker. Finally, due to several design decisions, the code of the proposed solution is significantly simpler and easier to maintain compared to the previous state, and updating the reference results is a matter of triggering a workflow. Even though not designed as a stand-alone framework directly portable to other projects, several of the ideas presented here can be easily applied to other projects (especially with services-oriented components) to simplify their system regression tests.

The system tests have already been helpful in action, already discovering a bug that was not previously visible by the routine release checks in the v3.1.0 release. Still, full-scale deployment is pending, and the triggering mechanisms can be improved to not require prescribing the version of each component, while still avoiding false cache hits. Testing can be more targetted with a parameter matrix. Reporting can be even more transparent with job summaries and dashboards, and local debugging of remote runs can be better facilitated. The test options, together with the metadata, should be standardized with a configuration schema, which is part of a current research project. In particular, an option to restrict the simulated time of a case could help to partially integrate longer cases into the CI. Archiving and checking additional results (such as number of iterations or runtime) could help discover more faults and performance regressions, with the latter paving the way for running performance tests on HPC hardware.

Acknowledgements: Several people have helped with previous work (as cited). In particular, Yakup Hoshaber (under the supervision of Florian Lindner) made a first attempt, while Dmytro Sashko and Konrad Eder (under the supervision of Gerasimos Chourdakis, with further input from Benjamin Rodenberg, Frédéric Simonis, and Benjamin Uekermann) extended the previous infrastructure. Gerasimos Chourdakis would also like to thank Jan Philipp Thiele for his feedback during and after the deRSE24 conference and Pavlos Tzianos for the helpful discussions about Docker. We thankfully acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2075 – 390740016 and under project number 528693298, and the support by the Stuttgart Center for Simulation Science (SimTech).

Bibliography

[ABD+21] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells. The deal.II finite element library: Design, features, and insights. *Computers & Mathematics with Applications* 81:407–422, 2021.
https://doi.org/10.1016/j.camwa.2020.02.022

[BF14] P. Bourque, R. E. Fairley (eds.). *SWEBOK: guide to the software engineering body of knowledge, version 3*. IEEE Computer Society, Los Alamitos, CA, 2014.
http://www.swebok.org/

[BLG+16] H.-J. Bungartz, F. Lindner, B. Gatzhammer, M. Mehl, K. Scheufele, A. Shukaev, B. Uekermann. preCICE – A fully parallel library for multi-physics surface coupling. *Computers and Fluids* 141:250–258, 2016. Advances in Fluid-Structure Interaction.
https://doi.org/10.1016/j.compfluid.2016.04.003

[CCD+24] J. Chen, G. Chourdakis, I. Desai, C. Homs-Pons, B. Rodenberg, D. Schneider, F. Simonis, B. Uekermann, K. Davis, A. Jaust, M. Kelm, N. Kotarsky, H. Kschidock, D. Mishra, M. Mühlhäußer, T. P. Schrader, M. Schulte, V. Seitz, J. Signorelli, G. van Zwieten, N. Vinnitchenko, T. Vladimirova, L. Willeke, E. Zonta. preCICE Distribution Version v2404.0. 2024. In review (28.05.2024).
https://doi.org/10.18419/darus-4167

[CDR+22] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H. Bungartz, L. Cheung Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, O. Koseomur. preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]. *Open Research Europe* 2(51), 2022.
https://doi.org/10.12688/openreseurope.14445.2

[DHH+23] B. Dröge, V. Holanda Rusu, K. Hoste, C. van Leeuwen, A. O'Cais, T. Röblitz. EESSI: A cross-platform ready-to-use optimised scientific software stack. *Software: Practice and Experience* 53(1):176–210, 2023.
https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3075

[GKP+23] D. Gläser, T. Koch, S. Peters, S. Marcus, B. Flemisch. fieldcompare: A Python package for regression testing simulation results. *Journal of Open Source Software* 8(81):4905, 2023.
https://doi.org/10.21105/joss.04905

[Hos18] Y. Hoshaber. Entwurf und Implementierung von Systemtests für eine verteilte Multi-Physik Simulationssoftware. Bachelor's thesis, University of Stuttgart, 2018.
https://www2.informatik.uni-stuttgart.de/bibliothek/ftp/medoc_restrict.ustuttgart_fi/BCLR-2018-21/BCLR-2018-21.pdf

[KB14] U. Kanewala, J. M. Bieman. Testing scientific software: A systematic literature review. *Information and Software Technology* 56(10):1219–1232, 2014. https://doi.org/10.1016/j.infsof.2014.05.006

[KMR+20] V. Karakasis, T. Manitaras, V. H. Rusu, R. Sarmiento-Pérez, C. Bignamini, M. Kraushaar, A. Jocksch, S. Omlin, G. Peretti-Pezzi, J. P. S. C. Augusto, B. Friesen, Y. He, L. Gerhardt, B. Cook, Z.-Q. You, S. Khuvis, K. Tomko. Enabling Continuous Testing of HPC Systems Using ReFrame. In Juckeland and Chandrasekaran (eds.), *Tools and Techniques for High Performance Computing*. Pp. 49–68. Springer International Publishing, Cham, 2020. https://doi.org/10.1007/978-3-030-44728-1_3

[KMW+13] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, X. Jiao, K. Jordan, D. Kaushik, E. Kaxiras, A. Koniges, K. Lee, A. Lott, Q. Lu, J. Magerlein, R. Maxwell, M. McCourt, M. Mehl, R. Pawlowski, A. P. Randles, D. Reynolds, B. Rivière, U. Rüde, T. Scheibe, J. Shadid, B. Sheehan, M. Shephard, A. Siegel, B. Smith, X. Tang, C. Wilson, B. Wohlmuth. Multiphysics simulations: Challenges and opportunities. *The International Journal of High Performance Computing Applications* 27(1):4–83, 2013. https://doi.org/10.1177/1094342012468181

[Lin19] F. Lindner. *Data Transfer in Partitioned Multi-Physics Simulations: Interpolation & Communication*. Dissertation, University of Stuttgart, Stuttgart, July 2019. http://doi.org/10.18419/opus-10581

[MRS16] R. Milk, S. Rave, F. Schindler. pyMOR – Generic Algorithms and Interfaces for Model Order Reduction. *SIAM Journal on Scientific Computing* 38(5):S194–S216, 2016. https://doi.org/10.1137/15M1026614

[SCU22] F. Simonis, G. Chourdakis, B. Uekermann. Overcoming Complexity in Testing Multiphysics Coupling Software (Better Scientific Software blog). https://bssw.io/blog_posts/overcoming-complexity-in-testing-multiphysics-coupling-software, feb 2022. Accessed: 2024-05-15.

[TKB+15] Y.-H. Tang, S. Kudo, X. Bian, Z. Li, G. E. Karniadakis. Multiscale universal interface: a concurrent framework for coupling heterogeneous solvers. *Journal of Computational Physics* 297:13–31, 2015. https://doi.org/10.1016/j.jcp.2015.05.004

[UBC+17] B. Uekermann, H.-J. Bungartz, L. Cheung Yau, G. Chourdakis, A. Rusch. Official preCICE adapters for standard open-source solvers. Pp. 210 – 213. Stuttgart, Germany, 2017. https://doi.org/10.18419/opus-9334