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

Improving reproducibility of scientific software using Nix/NixOS: A case study on the preCICE ecosystem

Max Hausch, Simon Hauser, Benjamin Uekermann

DOI: 10.14279/eceasst.v83.2613

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/)

Improving reproducibility of scientific software using Nix/NixOS: A case study on the preCICE ecosystem

Max Hausch[1*], Simon Hauser[1*], Benjamin Uekermann[1]

[1] Institute for Parallel and Distributed Systems
University of Stuttgart
benjamin.uekermann@ipvs.uni-stuttgart.de
[*] These authors contributed equally to this work.

Abstract: Ensuring reproducibility of scientific software is crucial for the advancement of research and the validation of scientific findings. However, achieving reproducibility in software-intensive scientific projects is often challenging due to dependencies, system configurations, and software environments. In this paper, we present a possible solution for these challenges by utilizing Nix and NixOS. Nix is a package manager and functional language, which guarantees that a package and all its dependencies can be built reproducibly. NixOS is a purely functional Linux distribution, built on top of Nix, which enables the build of reproducible systems including configuration files, packages, and their dependencies. We study the potential of Nix and NixOS by a case study on the reproducibility of the preCICE ecosystem. preCICE is a coupling library for partitioned multiphysics simulations. The ecosystem includes diverse legacy solvers, adapters, and language bindings besides the coupling library itself making it a challenging and representative testcase. We demonstrate how to create a reproducible and self-contained environment for this ecosystem and discuss the benefits and limitations of using Nix and NixOS.

Keywords: Reproducibility, Nix, NixOS

1 Introduction

In scientific research, it is crucial to be able to reproduce and verify results. Reproducibility ensures that experiments can be repeated and findings can be validated, which is essential for reliable and credible research. However, achieving reproducibility in scientific software has been a challenge due to complex dependencies, conflicting software environments, and changing software systems [Dal12]. Problems arise from dependencies, library versions, and system configurations, leading to inconsistencies across different computing environments. Traditional approaches to reproducibility, such as manual setup instructions or virtualization techniques, are prone to errors and time-consuming at best.

Many scientists aim to solve this situation using Docker[1] (e.g.,[KGS+23]), a software which describes software environments with the help of text files. These text files are made up of imperative commands, which are run inside of containers, one layer at a time. The result is several different layers combined into a single output image, which can be instantiated into a

[1] https://www.docker.com/

running container. Docker images can be copied to different hosts and should then provide the same environment on different machines. Those images are usually based on one of the official Docker images[2], however, which use traditional package managers, such as apt. When a user then specifies to install the python3 package, for instance, a traditional package manager could yield version 3.8 today, but version 3.9 in a few months. Full reproducibility can, thus, only be achieved by storing the complete image. Altering a single dependency (e.g., by a bugfix) causes a rebuild of the image and, thus, destroys reproducibility.

There are, moreover, commercial, domain specific solutions to achieve reproducibility, e.g., CodeOcean[3] mainly for bioinformatics or Weights and Biases[4] for machine learning. With these archiving platforms, experiments can be rerun using technologies such as Docker. The platforms are closed source, however, such that the source code cannot be reviewed nor adjusted [KGS+23].

In past years, the Nix package manager [DJV04] and NixOS [DLP10], a Linux distribution built around it, have emerged as promising alternatives [DDS15]. Nix allows functional descriptions of dependencies up to fixed versions, thus avoiding the issue described above. Similar ideas are followed by the popular high performance computing (HPC) package managers EasyBuild [HTGD12] and Spack [GLC+15]. At FOSDEM 2018, Kenneth Hoste compares Spack, EasyBuild, and Nix with each other [Hos18]. Minor deficits regarding reproducibility of Spack and EasyBuild are that they link against core system libraries, e.g., glibc, which can break independently. The Spack developers are aware of these shortcomings and are working on improvements[5,6]. EasyBuild, on the other hand, even has an option for linking against system libraries called `osdependencies`[7]. Thus, packages need to be vetted and potentially updated prior to usage. All three solutions have in common that they rely on scientific software following best practices concerning building and packaging. Unfortunately, most legacy software projects do not do this. This is why, for example, the xSDK community [xD23] tries to set a standard for policies for math software.

In this paper, we analyze how well Nix and NixOS can improve reproducibility of scientific software. To this end, we study the preCICE ecosystem [CDR+22] as an example. preCICE is a coupling library for partitioned multiphysics simulations. The ecosystem includes diverse legacy solvers, adapters, language bindings, and tutorials besides the coupling library itself making it a challenging and representative testcase. We try to build the complete ecosystem using Nix and run all tutorials. Section 2 briefly introduces the background of Nix and preCICE. Section 3 then presents the case study including challenges, workarounds, and open problems. Afterwards, we discuss the results in Section 4, followed by the conclusions in Section 5. This paper is the result of a student research project and a streamlined version of its report [HH23]. Beyond the content of this paper, the report also includes a detailed comparison of Nix to Spack and EasyBuild, and a discussion on how to use Nix on HPC systems.

2 https://docs.docker.com/trusted-content/official-images/
3 https://codeocean.com/
4 https://wandb.ai/site
5 https://github.com/spack/spack/issues/39560
6 https://github.com/spack/spack/pull/42082
7 https://docs.easybuild.io/writing-easyconfig-files/#dependency_specs

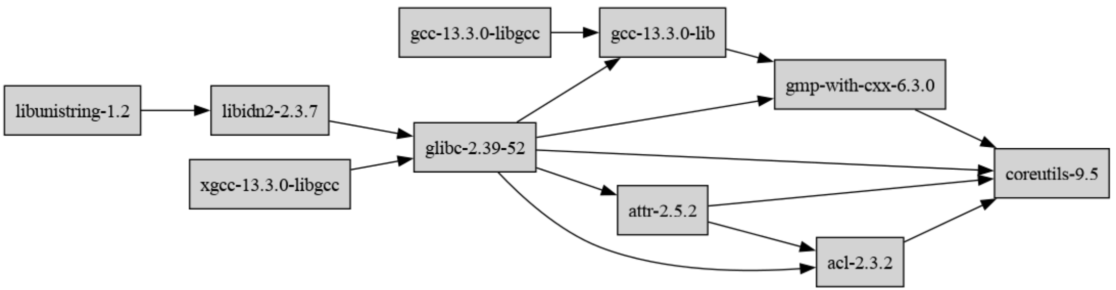

Figure 1: The full dependency graph of the coreutils package, generated by `nix-store --query --graph $(nix build nixpkgs#coreutils --print-out-paths) | dot -Tpng -Grankdir=LR -ocoreutils.png`

2 Background

This section gives a short introduction into Nix and preCICE.

2.1 Nix

Nix is a purely functional package manager, which provides features to build software derivations reproducibly. It does so, by recursively calculating a hash over all inputs of a derivation and its dependencies to ensure the completeness of the whole derivation. If any of the inputs changes, all dependent derivations have to be rebuilt. Fig. 1 shows the dependency graph of the coreutils package as an example.

As Nix is purely functional, it relies on functions that, without side-effects, realize the package derivations. Inputs of these functions are parameters such as the package version or the location of the source code and its dependencies. Nix offers built-in functions for common build systems and languages, such as CMake, Rust, Go, and Python by implementing different phases. Phases are bash scripts and are always executed in the same order. They are predefined for each build system, but can be overwritten if necessary.

For building packages, the four key phases are the `configurePhase`, the `buildPhase`, the `installPhase` and the `checkPhase`. During the `configurePhase`, the build system is configured. The `buildPhase` defines the build steps, such as calling `make` or `cargo build`. Afterwards, the `installPhase` copies the results into the output directory. The `checkPhase` can be enabled to run any tests. Most phases have a pre and post step, which can be uses to make adjustments.

Evaluating these Nix functions with the same inputs yields the same outputs. This is a critical factor for Nix's reproducibility. Outputs, i.e. the build artifacts, never change after being built once. Nix builds are, moreover, sandboxed, meaning that there is no internet access possible during a build.

All contents, including source files and resulting build artifacts, are stored inside the Nix store. Per default, the Nix store resides in the path `/nix/store` on the file system. The naming scheme for packages includes the above mentioned hash, for preCICE v2.5.0, for example, `/nix/store/0a5gw3l...-precice-2.5.0`. This eases checking the Nix store for in-

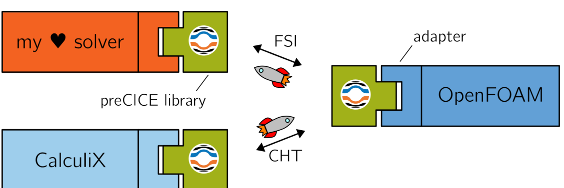

Figure 2: Software setup of preCICE from the preCICE website https://precice.org. The setup shows a custom solver coupled to OpenFOAM and CalculiX for conjugate-heat transfer (CHT) and fluid-structure interaction (FSI) simulations as an example.

tegrity and ensures that builds using the same inputs are performed only once. As all build outputs reference their complete dependency graphs inside the Nix store, they do not interfere with other build outputs, enabling installation of multiple versions of the same software without conflicts.

Software patches are easy to apply in Nix. One can simply provide a list of `.patch` files as inputs, Nix then includes the patch files in the build and in the calculation of the hash. Users can override all inputs, so that every package can be flexibly adjusted.

2.2 preCICE

preCICE (Precise Code Interaction Coupling Environment) is an open-source software library designed to facilitate coupling of different simulation software packages. It provides an API, which allows different simulation tools to exchange data and work together in a collaborative manner, enabling multi-physics as well as multi-scale simulations. In fact, many scientific simulations require complex combinations of multiple solvers, each specialized in a particular aspect of the problem. For example, in fluid-structure interaction problems, where fluid flow interacts with deformable structures, separate solvers can be used to model the fluid dynamics and structural mechanics.

preCICE supports a wide range of existing (Legacy) solvers. For each solver, there is typically a specific adapter: an either independent software package or simply a source code patch that integrates the preCICE API into the solver, see Fig. 2. The preCICE library itself is implemented in C++, bindings to other languages are either supported through native bindings (C, Fortran) or through independent software packages (Python, Julia, Matlab, Fortran). Many official adapters, all bindings, the library itself, tutorials, additional tools, and the website including user documentation are collected and released in so-called preCICE distributions[8]. In our case study, we target the distribution v2211.0 [CDD+23]. To execute simulations, such as the included tutorials, the actual solvers are required besides the distribution. These are included, for example, in the preCICE VM[9] and also part of our case study. Table 1 lists all software packages of the case study. We try to reproducibly build all of these packages with Nix and try to run all tutorial cases. The diverse languages and diverse solver packages using various build system makes the preCICE ecosystem a challenging testcase, which is, however, representative for current research

[8] https://precice.org/installation-distribution.html

[9] https://precice.org/installation-vm.html

Table 1: Software packages of the case study

Package	Type	Build system	Version
preCICE	library	CMake	2.5.0
ASTE	tooling	CMake	3.0.0
Config visualizer	tooling	setup.py	60f2165
MATLAB bindings	bindings	matlab build script	2.5.0.0
Fortran module	bindings	make	9e3f405
Python bindings	bindings	setup.py	2.5.0.1
Julia bindings	bindings	julia	2.5.0
preCICE-CalculiX adapter	solver + adapter	Makefile	2.20
Code_Aster	solver	custom system in python	14.6.0
Code_Aster-preCICE adapter	adapter	custom system in python	ce995e0
deal.II	solver	CMake	9.4.1
preCICE-deal.II adapter	adapter	CMake	dbb25bea
DUNE-FEM	solver library	setup.py	2.8.0
preCICE-DUNE adapter	adapter	scripts + CMake	5f2364d
FEniCS	solver library	CMake + setup.py	2019.1.0
preCICE-FEniCS adapter	adapter	setup.py	1.4.0
Nutils	solver library	setup.py	7.0.0
OpenFOAM	solver	wmake/Allwmake	2206
preCICE-OpenFOAM adapter	adapter	wmake/Allwmake	1.2.1
preCICE-SU2 adapter	solver + adapter	patch script + Autoconf	6.0.0

software in our experience.

3 Case study

We now study the packaging of all components of the preCICE ecosystem one by one, including challenges, solutions, and workarounds. We first look at preCICE itself, followed by additional tooling and bindings. Afterwards, we study all solvers and their respected adapters. Next, we show that Nix can also be used to generate iso files as well as qemu and Vagrant VM images. Finally, we try running all preCICE tutorials. The Nix code for all components is available online[10] and can be used to download and run any of the presented packages.

3.1 preCICE

The preCICE library itself is already available in the nixpkgs repository[11], so we do not have to package the software. Looking at the package definition and the preCICE source code, the library is quite easy to build and package as it uses CMake as a build system.

[10] https://github.com/precice/nix-packages/releases/tag/deRSE24-paper-submission-v2

[11] https://github.com/NixOS/nixpkgs/blob/nixos-24.05/pkgs/development/libraries/precice/default.nix#L41

3.2 Tools

The preCICE distribution includes two tools that enhance the preCICE user experience.

One of the tools is ASTE[12], which stands for Artificial Solver Testing Environment. It is a thin wrapper around the preCICE API, which allows, for instance, testing of data mapping with real geometries. ASTE requires VTK[13], a visualization toolkit, which is built without python support in the nixpkgs repository to reduce compilation time. This feature is needed for ASTE, however. As already mentioned, Nix allows overriding of inputs, so in this case, we can look at the package definition of VTK. The parameter `enablePython` can simply be set to `true`. We then also need to supply the version of python, which we set to python3.

The second tool is the preCICE config visualizer. As the name suggests, it visualizes preCICE configuration files. Similarly to preCICE itself, the tool is already packaged upstream[14]. This tool is easy to handle with Nix, as it is a python package.

3.3 Bindings

Bindings are used to offer interfaces to other programming languages.

3.3.1 MATLAB bindings

There were several attempts to package MATLAB[15] for NixOS in the past[16], yet there was no success so far. This might be due to the fact, that MATLAB needs to be installed by running an installation wizard that downloads files during the installation process and verifies the license. As MATLAB has some licensing issues and is also not installed in the preCICE VM, we do not proceed in packaging the preCICE bindings for the language.

3.3.2 Fortran module

We were able to package the Fortran module of preCICE by providing custom contents for the `buildPhase` and the `installPhase` of Nix. The resulting binary could run the example that comes with the module.

3.3.3 Python bindings

The pyprecice package is already available upstream, but does not compile as it is not able to find the python module pkgconfig. After adding this single dependency as an input to the package definition, the python package builds and can be used as expected. This solution was contributed to the upstream nixpkgs repository[17].

[12] https://github.com/precice/aste
[13] https://vtk.org/
[14] https://github.com/NixOS/nixpkgs/blob/nixos-24.05/pkgs/tools/misc/precice-config-visualizer/default.nix#L25
[15] https://de.mathworks.com/products/matlab.html
[16] https://github.com/NixOS/nixpkgs/issues/56887
[17] https://github.com/NixOS/nixpkgs/commit/fd8962162ac21be59fc3a05fb6a250eeab6b2bec

3.3.4 Julia bindings

Julia [BEKS17] support in NixOS is currently still in its early stages and cannot be declaratively defined by Nix at the current state[18].

3.4 Solvers and adapters

3.4.1 CalculiX

The preCICE adapter for the structural mechanics code CalculiX [UBC+17] requires the original CalculiX source code and provides a new Makefile in the adapter repository. By default, this Makefile expects the source code to reside in `$HOME`, but it is possible to override this location with a make variable. It then builds the original code and the adapter code together, resulting in a combined binary.

The Makefile also has hard-coded locations of its dependencies: SPOOLES, ARPACK, and yaml-cpp. These need to be replaced with the equivalent pkg-config calls. Additionally, there is no install target provided by the Makefile, thus the resulting binary needs to be installed manually by copying it to the `$out` placeholder, which maps to the resulting store path.

3.4.2 Code_Aster

The Code_Aster-preCICE adapter [UBC+17] is a python file, which needs to be placed in the Code_Aster lib directory. For this, the Code_Aster solver needs to be packaged at version 14.6. It uses a custom build system based on a `setup.py` file, which invokes build and install phases for dependencies and in the end for the solver itself.

The Code_Aster package distributes its dependencies as pinned tarballs. On Nix, almost none of these dependencies complete the build stage. We, thus, need to pull them, package them separately as Nix packages, and configure the build system with a `setup.cfg`. In this file, we are able to disable the installation of the pinned dependencies and provide paths to the dependencies instead.

More precisely, we disable HDF5, as it is already packaged in the upstream nixpkgs repository, even at the required version 5.1.10. Additionally, Code_Aster requires Medfile and Scotch, which are both available upstream and can be used as dependencies. Another dependency is METIS, which partially builds. Thus, we decide to not replace it, but to fix the remainder of the build. The issue is precisely that the `CMakeLists.txt` for the METIS programs hard-codes `link_directories` to `/home/karypis/local/lib`. We unpack the METIS tarball, remove this line with sed, and repack the directory.

The remaining two dependencies, Mumps and Tfel are not available upstream and need to be packaged. Mumps uses a Makefile and can be configured using `Makefile.inc`. We write our own custom `Makefile.inc` by providing the locations for dependencies, such as Scotch, METIS, ParMETIS, or Blacs. Almost all dependencies are available upstream, but we need to recompile Scotch with additional build flags. Moreover, for packaging Blacs, which also uses a Makefile, we configure the build using a custom `Bmake.inc` file, which is used to set compiler

[18] https://github.com/NixOS/nixpkgs/issues/20649

flags, the path to MPI, and the bash installation. Tfel, on the other hand, uses CMake as build system making it easy to package it with Nix.

After successfully packaging all dependencies, the `buildPhase` of Code_Aster completes, but the installation part has another issue, which also affects new versions of Ubuntu and other distributions using a python version greater than 3.9. There is a forum post[19] without resolution, so we need to manually patch the bug. The issue is precisely, that the custom build system does not correctly calculate the python site-packages directory. It unconditionally slices the first 3 chars from `sys.version`, which works for python version 3.9.x, but not for version 3.10.x. After patching, Code_Aster successfully installed into `$out/14.6/`, so we move around some files until we have a valid directory structure with `$out/bin`, `$out/lib`, providing symlinks for `$out/14.6/` and `$out/stable/`.

3.4.3 deal.II

To package the deal.II-preCICE adapter, we first need to package deal.II [ABB+23] itself, which uses CMake as build system and works without any adjustments. The same then applies for the adapter, which also uses CMake. As an example for the Nix language, we provide the Nix code for the deal.II-preCICE adapter in Fig. 3.

3.4.4 DUNE

DUNE [BBD+21] uses a combination of CMake files and custom build scripts making the build process in Nix tedious. There is an ongoing migration from Autotools to CMake for DUNE, which might improve the situation in the future. Another minor issue is that the DUNE project is rather a collection of repositories than a monorepo. This means, users have to correctly clone all repositories such that the build system finds all relevant information. We clone all of the required DUNE repositories into a directory and additionally clone the DUNE-preCICE adapter into the same directory. For the build and install process, we need to manually set the `$DUNE_CONTROL_PATH` and the `$DUNE_PY_DIR` environment variables. Additionally, we need to patch the python install process because the current CMake file tries to access the internet with `pip install`, which is not allowed in Nix's sandboxed builds. The two environment variables must also be set at run-time. This can be achieved by sourcing the `set-dune-vars` script, which we provide.

3.4.5 FEniCS

FEniCS [fen12] is already packaged upstream, however, not with all features the FEniCs-preCICE adapter [RDH+21] relies on, in particular PETSc support and mshr, the FEniCS mesh generator. This is why we need to package PETSc4py, the python bindings package of PETSc, which is not available upstream. The build process uses the internal `buildPythonApplication` build tool and runs successfully once we add the option `build_src --force` to rebuild cython code and pin the cython version to 0.29.34 [20]. We could not enable tests for PETSc4py, however,

[19] https://forum.code-aster.org/public/d/26475-problem-installing-code-aster-version-14-6/11
[20] https://gitlab.com/petsc/petsc/-/issues/1359

```
{
  # Inputs including nix functions/packages or additional flags
  lib, stdenv, fetchFromGitHub, cmake, dealii,
  # Optional features that users can enable when overriding
  enable3d ? false
}:
stdenv.mkDerivation rec {
  pname = "precice-dealii-adapter";
  version = "unstable-2022-09-23"; # could also be a git tag

  src = fetchFromGitHub { # Defining where to get the source from
    owner = "precice"; repo = "dealii-adapter";
    rev = "dbb25...8367c"; sha256 = "sha256-pPQ2...2jlflgUE=";
  };

  # dependencies only available at build-time
  nativeBuildInputs = [ cmake ];
  # dependencies that also need to be installed at run-time
  # mostly libraries needed for linking and for execution
  buildInputs = [ precice dealii ];

  cmakeFlags = lib.optionals enable3d [ "-DDIM=3" ];

  # nix' default phases can be overwritten such as:
  installPhase = ''
    # $out contains the final path inside the nix store.
    # The resulting build files are copied into this directory
    mkdir -p $out/bin && cp elasticity $out/bin/elasticity
  '';
}
```

Figure 3: Nix code to package the deal.II-preCICE adapter

because they depend on OpenMPI and the network, which is not available within Nix's build sandbox. Afterwards, we specify PETSc4py as dependency for FEniCS and enable the PETSc support. Additionally, we need to recompile PETSc with additional features enabled that are not enabled by the upstream Nix package. These include ParMETIS, HYPRE and ScaLAPACK. We use `overrideAttrs`, a Nix feature that allows us to change an existing package. Lastly, we also package mshr, so all required features for the adapter package are now are available.

Building the FEniCS-preCICE adapter is now directly possible with `buildPythonPackage`. We then validate the correctness by including an import check as well as successfully running the full test suite.

3.4.6 Nutils

The Nutils [ZZH22] solver can be built using `buildPythonPackage`. One test needs to be disabled, but the rest of the testsuite passes without issues. There is no dedicated Nutils-preCICE adapter, but preCICE is typically directly called from Nutils application scripts.

3.4.7 OpenFOAM

The preCICE-OpenFOAM adapter [CSU23] supports multiple flavors of OpenFOAM. We restrict our analysis to the OpenFOAM fork of OpenCFD Ltd[21].

OpenFOAM uses its own custom build system called wmake, which is typically called with a Allwmake wrapper script. The build system sets 36 environment variables, one step at a time by checking several parameters, e.g. the CPU architecture or the location of the source code. Also during run-time, these variables need to be set, either by a wrapper script or by manually sourcing a file inside the installation directory of OpenFOAM. For Nix, these properties are unfortunate. To compile OpenFOAM with Nix, we need to patch the shebangs[22] of wmake to make it run during the build. We also use a shell script which exports the environment variables to the current shell session. For simplicity, we hard-code all parameters, such as the processor architecture. These could be parametrized, however, based on the Nix inputs to allow for optimized builds. We use the script during the `buildPhase` to source all variables such that, for example, `OPENFOAM_SRC_PATH` points to the default location `/build/openfoam`. Afterwards, `./Allwmake -j -q` is sufficient to start the build. The `installPhase` then copies the necessary files and directories to `$out`, replaces the mock `OPENFOAM_SRC_PATH` by the value of `$out`, and creates a wrapper for the `openfoam` shell script.

The preCICE-OpenFOAM adapter is an OpenFOAM function object, requiring OpenFOAM and wmake as build inputs. The environment variables have to be set again through the shell script mentioned above. Afterwards, we set the target directory for the adapter to `$out/lib/` as the adapter is a shared library. Building then works successfully.

[21] https://www.openfoam.com/
[22] https://foldoc.org/shebang

3.4.8 SU2

The SU2-preCICE adapter [UBC+17] patches and extends the original CFD code SU2 [PCA+13]. To this end, the adapter provides a script to run before the `buildPhase`. We do this in Nix' `patchPhase`, which is responsible for patching the source before running the `buildPhase`. Afterwards, `stdenv` automatically recognizes that SU2 uses Autotools for building and runs the `configurePhase`. Finally, to find the preCICE installation, we need to set the configuration flags `--with-include` and `--with-lib`.

Since preCICE distribution v202404, this adapter is replaced by a python-based solution

3.5 preCICE VM

The current preCICE VM[23] is built on top of Vagrant[24] using Ubuntu as a base image. When provisioned for the first time, it further installs software, compiles programs, and clones the preCICE tutorials[25]. This means that we only have reproducibility when fetching the VM from vagrant cloud [26], as we can not rebuild the VM from scratch, because the provision step currently fetches files from the internet that are at this very point in time the *latest* version, i.e. the main branch of the preCICE repository rather than a pinned tag or commit. Making us dependent on vagrant cloud to continue to serve the box.

Nix comes with the built-in functionality of producing qemu[27] VM images. We use Nix to define a VM image with NixOS as a base, which can be built reproducibly. The image contains all components of the case study and some additional custom tools that can only be seen in the official VM[28], such as the `preciceToPNG` command. Also, we generate an iso and a Vagrant VirtualBox file of the VM as additional outputs.

3.6 preCICE tutorials

The preCICE tutorials included in the distribution cannot only be used as reference examples on how to couple different solvers, but also to verify functionality of solver and adapter installations by executing them. To this end, we execute all 23 tutorials of the distribution such that each coupled solver option is at least run once. We do not compare results, but only test whether the cases complete. We test inside the NixOS VM and also in an ad-hoc shell environment provided by the `nix develop` command. All tests pass except the *Turek-Hron FSI3*, parts of the *partitioned heat conduction*, parts of the *perpendicular flap*, and the *flow over heated plate steady state* cases. We could not build swak4foam, an add-on to OpenFOAM, which is needed in the Turek-Hron FSI3 tutorial and the OpenFOAM solver of the partitioned heat conduction tutorial. The official preCICE VM uses a prebuilt version of swak4foam, but also patching the prebuilt binary did not work. The dependency is, however, no longer required in the v202404.0 release[29] of

[23] https://precice.org/installation-vm.html
[24] https://www.vagrantup.com/
[25] https://github.com/precice/tutorials/
[26] https://app.vagrantup.com/precice/boxes/precice-vm
[27] https://www.qemu.org/
[28] https://github.com/precice/vm
[29] https://precice.org/installation-distribution.html#v24040

the preCICE distribution. We could, moreover, not package the third-party solids4foam solver, which is one of many options in the perpendicular flap tutorial. The flow over heated plate steady state tutorial requires Code_Aster. Even though we manage to package the solver, we observe a segmentation fault at run-time when the python code tries to call a Fortran subroutine – a problem we did not manage to debug.

4 Discussion

After detailing the challenges of building and packaging all components of the preCICE ecosystem and running the tutorials as testcases, we want to discuss common points between these scientific software packages. The required effort to package the individual components varies drastically. Major problems can be traced back to the build system. Custom build systems require an disproportional amount of additional work to make them runnable on any system. OpenFOAM and Allwmake are prime examples of this issue. Package managers have to understand a whole new build system for a single package – a fact that is often ignored when considering the trade-off between maintaining a custom build system versus pivoting to an industry standard, such as CMake or Autotools. Such standard build tools generally provide interfaces for dependency management and optional features. This is often better than only documenting these, since documentation needs to be kept in sync with the underlying code.

The authors of xSDK[30] come to a similar conclusion. They provide a list of package policies [xD23], which, for example, specify that packages must have an appropriate build system, which includes CMake and Autotools as examples. If a package supports all policies, they can be added to the growing list of packages in xSDK. It is no surprise that xSDK packages (e.g., PETSc, deal.II, or preCICE) are also easy to package with Nix.

The xSDK policies also requires portable installations. Several components in the preCICE ecosystem require installation into `$HOME` or any required files to be located at specific locations. Both, the CalculiX-preCICE adapter and the SU2-preCICE adapter require the original source code to be present as they add new features on top or patch the source code. The current solution of requiring to clone the code into a specific directory works, but it might be better to look into Git submodules, as this would result in a more portable solution. This would, moreover, allow pinning the version of the original code compared to only documenting it.

Finally, hard-coding libraries to `/usr/lib` and `/usr/include` might work on one system, but might make it hard or even impossible to install a package on a different system. Configurable solutions, such as pkg-config, should always be preferred and work out of the box with common paths such as `/usr/lib`.

5 Conclusion

We investigated on whether Nix und NixOS are a potential solution to enable full reproducibility of research software environments. As a case study, we tried to package and test all components of the preCICE ecosystem – a very heterogeneous set of legacy software packages, which is,

[30] https://xsdk.info/

however, representative for the state of research software today in our experience.

Out of the 20 components of the preCICE distribution, we were able to package 14 components ourselves, four were already packaged upstream, and two were not packageable in their current state. In total, there are 22 tutorials in the preCICE v2211 distribution, many having several different coupled solvers, resulting in a total of 60 coupled solvers. We were able to run 52 out of these. Many packages required workarounds, however, which might be difficult to achieve for non-experienced Nix users. Nix also comes with a few peculiarities, which complicate workarounds. In fact, every piece of software has to lie within `/nix/store` and each path therein is its own-isolated tree inspired by the Filesystem Hierarchy Standard (FHS).

Most problems, however, can be traced back to a lack of standardization in research software, especially regarding build systems. Software packages that follow best practices for their programming language are also straightforward to package. The xSDK initiative defined such standardization policies for math software. Components of the preCICE ecosystem that already adhere to this standard, were among the easiest ones to package – even though xSDK ultimately targets Spack and not Nix.

Despite encountering challenges in packaging the numerous software tools, we were able to observe the adaptability of the Nix package manager. Our results indicate that, in general, Nix is a suitable solution for reproducing software environments and a viable solution to package management in a scientific domain.

Acknowledgements: 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

[ABB+23] D. Arndt, W. Bangerth, M. Bergbauer, M. Feder, M. Fehling, J. Heinz, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, B. Turcksin, D. Wells, S. Zampini. The `deal.II` Library, Version 9.5. *Journal of Numerical Mathematics* 31(3):231–246, 2023.
doi:10.1515/jnma-2023-0089

[BBD+21] P. Bastian, M. Blatt, A. Dedner, N.-A. Dreier, C. Engwer, R. Fritze, C. Gräser, C. Grüninger, D. Kempf, R. Klöfkorn, M. Ohlberger, O. Sander. The DUNE Framework: Basic Concepts and Recent Developments. *Computers & Mathematics with Applications* 81:75–112, 2021.
doi:10.1016/j.camwa.2020.06.007

[BEKS17] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM review* 59(1):65–98, 2017.
doi:10.1137/141000671

[CDD+23] G. Chourdakis, K. Davis, I. Desai, B. Rodenberg, D. Schneider, F. Simonis, B. Uekermann, B. Ariguib, P. Cardiff, A. Jaust, P. Kharitenko, R. Klöfkorn, N. Kotarsky,

B. Martin, E. Scheurer, V. Schüller, G. van Zwieten, K. Yurt. preCICE Distribution Version v2211.0. 2023.
doi:10.18419/darus-3576

[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.
doi:10.12688/openreseurope.14445.2

[CSU23] G. Chourdakis, D. Schneider, B. Uekermann. OpenFOAM-preCICE: Coupling OpenFOAM with external solvers for multi-physics simulations. *OpenFOAM® Journal* 3:1–25, 2023.
doi:10.51560/ofj.v3.88

[Dal12] O. Dalle. On reproducibility and traceability of simulations. *Proceedings - Winter Simulation Conference*, 2012.
doi:10.1109/WSC.2012.6465284

[DDS15] A. Devresse, F. Delalondre, F. Schürmann. Nix based fully automated workflows and ecosystem to guarantee scientific result reproducibility across software environments and systems. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. ACM, 2015.
doi:10.1145/2830168.2830172

[xD23] xSDK Developers. xSDK Community Package Policies 1.0.0. 2023.
doi:10.6084/m9.figshare.13087196.v1

[DJV04] E. Dolstra, M. de Jonge, E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th USENIX Conference on System Administration*. LISA '04, p. 79–92. USENIX Association, USA, 2004.
doi:10.5555/1052676.1052686

[DLP10] E. Dolstra, A. Löh, N. Pierron. NixOS: A purely functional Linux distribution. Volume 20(5-6), p. 577–615. Cambridge University Press, 2010.
doi:10.1017/S0956796810000195

[fen12] *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Springer Berlin Heidelberg, 2012.
doi:10.1007/978-3-642-23099-8

[GLC+15] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, S. Futral. The Spack package manager: bringing order to HPC software chaos. In *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Pp. 1–12. IEEE Computer Society, Los Alamitos, CA, USA,

2015.
doi:10.1145/2807591.2807623

[HH23] M. Hausch, S. Hauser. Improving reproducibility of scientific software using Nix/NixOS. Technical report, University of Stuttgart, 2023.
https://github.com/precice/nix-packages/releases/tag/initial-paper-release

[Hos18] K. Hoste. Installing software for scientists on a multi-user HPC system. FOSDEM, 2018.
https://archive.fosdem.org/2018/schedule/event/installing_software_for_scientists/

[HTGD12] K. Hoste, J. Timmerman, A. Georges, S. De Weirdt. EasyBuild: Building Software with Ease. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. Pp. 572–582. 2012.
doi:10.1109/SC.Companion.2012.81

[KGS+23] T. Koch, D. Gläser, A. Seeland, S. Roy, K. Schulze, K. Weishaupt, D. Boehringer, S. Hermann, B. Flemisch. A sustainable infrastructure concept for improved accessibility, reusability, and archival of research software. 2023.
doi:10.48550/arXiv.2301.12830

[PCA+13] F. Palacios, M. Colonno, A. Aranake, A. Campos, S. Copeland, T. Economon, A. Lonkar, T. Lukaczyk, T. Taylor, J. Alonso. Stanford University Unstructured (SU2): An open-source integrated computational environment for multi-physics simulation and design. *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition 2013*, 2013.
doi:10.2514/6.2013-287

[RDH+21] B. Rodenberg, I. Desai, R. Hertrich, A. Jaust, B. Uekermann. FEniCS–preCICE: Coupling FEniCS to other simulation software. *SoftwareX* 16:100807, 2021.
doi:10.1016/j.softx.2021.100807

[UBC+17] B. Uekermann, H.-J. Bungartz, L. Cheung Yau, G. Chourdakis, A. Rusch. Official preCICE Adapters for Standard Open-Source Solvers. In *Proceedings of the 7th GACM Colloquium on Computational Mechanics for Young Scientists from Academia*. 2017.
https://www.gacm2017.uni-stuttgart.de/registration/Upload/ExtendedAbstracts/ExtendedAbstract_0138.pdf

[ZZH22] J. S. B. van Zwieten, G. J. van Zwieten, W. Hoitinga. Nutils. Feb. 2022.
doi:10.5281/zenodo.6006701