# Creating a Reusable Cross-Disciplinary Multi-scale and Multi-physics Framework: From AMUSE to OMUSE and Beyond

Inti Pelupessy[1(✉)], Simon Portegies Zwart[2], Arjen van Elteren[2], Henk Dijkstra[3], Fredrik Jansson[3], Daan Crommelin[4,5], Pier Siebesma[6,7], Ben van Werkhoven[1], and Gijs van den Oord[1]

[1] Netherlands eScience Center, Amsterdam, The Netherlands
i.pelupessy@esciencecenter.nl
[2] Leiden Observatory, Leiden, The Netherlands
[3] Institute for Marine and Atmospheric Research Utrecht, Utrecht, The Netherlands
[4] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[5] University of Amsterdam, Amsterdam, The Netherlands
[6] Royal Netherlands Meteorological Institute, De Bilt, The Netherlands
[7] Delft University of Technology, Delft, The Netherlands

**Abstract.** Here, we describe our efforts to create a multi-scale and multi-physics framework that can be retargeted across different disciplines. Currently we have implemented our approach in the astrophysical domain, for which we developed AMUSE (github.com/amusecode/amuse), and generalized this to the oceanographic and climate sciences, which led to the development of OMUSE (bitbucket.org/omuse). The objective of this paper is to document the design choices that led to the successful implementation of these frameworks as well as the future challenges in applying this approach to other domains.

**Keywords:** Multi-scale simulations · Coupling framework · Multi-physics

## 1 Introduction

The current frontier in computational modelling is the simulation of complex phenomena involving different physical processes interacting on vastly different scales. The advent of massively parallel machines and GPU accelerated solvers, has meant that memory and CPU time bounds are less of a limitation as before, the difficulty shifting instead to the intrinsic complexity of the calculations.

A recurring challenge involves the interaction of processes acting on widely different scales. For example, when modelling the formation of planetary systems in a stellar cluster one needs to follow the collapse of interstellar gas, down to the formation of proto-stellar systems. Another example occurs when modelling


J. M. F. Rodrigues et al. (Eds.): ICCS 2019, LNCS 11539, pp. 379–392, 2019.
https://doi.org/10.1007/978-3-030-22747-0_29

the dynamical effects of clouds and convection on the atmospheric circulation. Atmospheric convection and cloud formation are physical processes with small spatial scales, however they affect the properties of the large-scale atmospheric flow, for example through their impact on the distribution of moisture and heat as well as on radiative transfer in the atmosphere.

Many more examples could be given and the conventional approach to these multi-scale problems, i.e. building a single, monolithic program with as much physics as possible, is expensive and difficult to scale. Building multi-scale and multi-physics simulation codes becomes increasingly complex with each new physical ingredient that is added. Furthermore, one is presented with the prospect of duplicating much of this work when a different solver or method is needed, a situation that often arises when a slightly different regime is accessed than that originally envisaged or when results need to be verified with a different method.

Different strategies that attempt to simplify this problem by compartmentalizing processes and combining the resulting building blocks exist. For example [8,18] are coupling frameworks geared towards earth system modelling. Other examples are the more general approaches taken in the toolkits [1,2]. See [6] for a more thorough review. Most of these can be roughly divided into *integrated* and *coupling library* approaches [19]. In the integrated approach, the functionality provided by the components (e.g. by subroutines of the code) is separated out and joined in a new single executable. In the library approach the original codes themselves are adapted to communicate with each other using an Application Programming Interface (API), linking against the coupling library.

We recently developed a promising alternative, especially when the target solvers use completely independent computational methods or discretizations. The fundamental idea of AMUSE [13,15] and OMUSE [14] is the abstraction of the functionality of existing simulation codes - which are often highly specialized and optimized for their domain of application - into physically motivated interfaces and bind these into a modern and flexible scripting language. Our approach has the benefit of the parallelism and flexibility provided by a coupling library approach, and the benefit of abstracting much of the bookkeeping inherent to code couplings using modern high-level constructs. In this way, complex simulations can be described in compact scripts, that can be easily understood and communicated between peers.

## 2 Genesis

AMUSE was conceived within MODEST, a tight knit astrophysical community of modellers interested in dense stellar system. AMUSE was envisioned as the need of going beyond purely gravitational N-body calculations became apparent, and the MODEST community sought ways to incorporate the effects of stellar evolution and the dynamics of the interstellar gas into their models. It quickly became evident that many simulation codes already existed, specifically developed to study these processes separately. More practically speaking, the

MODEST community realized that they did not have the manpower nor the expertise to develop these from scratch.

Even so, such a body of existing scientific codes, which we refer to as the community code base, is not trivial to interface with. The codes are written in different languages, have different requirements and are not necessarily written in a way that allows easy interfacing.

The way this problem was solved in AMUSE was by defining a thin interface layer in Python for these codes, which integrates with a framework layer tying the various components together, minimizing the necessary changes to the community codes. The use of these codes is simplified by standardizing the interface for the different relevant domains (e.g. gravitational dynamics, gas dynamics or radiative transfer) and a high degree of automation. The framework is designed with parallel simulation codes in mind and allows for running in a distributed environment. In practice the computational effort is in the highly tuned codes, allowing for high performance. In this way, AMUSE allowed codes to be retargeted for novel interactions and couplings with other component codes.

AMUSE was developed by a small team of astrophysicist and software engineers over a couple of years. In our experience, it is crucial to seek active involvement from the community early on, by organizing workshops and tutorials. This allows for early feedback, fosters involvement and helps creating a forgiving user base.

### 2.1 Development of OMUSE

While the original goal of AMUSE was to allow for realistic simulations of star cluster formation and evolution, no limits were imposed on the design and many published results using AMUSE had no relation to star cluster physics. In discussion with researchers in other scientific disciplines it became apparent that they struggled with fundamentally the same multi-scale coupling problems. At this point (around 2013), the Netherlands eScience Center, the national center for academic research software, funded a project to generalize the AMUSE framework, which resulted in OMUSE.

OMUSE was developed as an extension of the AMUSE framework, exposing a `omuse` name space with similar structure as for AMUSE, using the underlying infrastructure of AMUSE (Practically speaking this means that to use OMUSE, the user first has to install AMUSE, which is not ideal, since by default many component of AMUSE are installed that are not needed by OMUSE). The development of OMUSE involved transplanting the experiences gained in astrophysics to another scientific fields, which is as much a cultural challenge as it is a technical one.

To support earth science applications a number of features were added to the AMUSE framework: the data model was extended with a hierarchy of grid types, such that codes with various grid types, ranging from regular Cartesian grids to unstructured grids, can be supported (within AMUSE only Cartesian grids were available). For the data transfer between these grids data channels can be defined which perform grid remapping and functional transforms (in addition to

simple copy channels for use between equal formed grids). A number of domain specific units and utility functions were also added. As the initial focus was on oceanographic applications, the codes that are included range from simple conceptual ocean models to global circulation models.

## 3 About the Design

Here we will illustrate the design of AMUSE stepping through an example script to evolve a model star cluster using pure N-body dynamics.

As mentioned above, AMUSE and OMUSE are implemented in Python. The requirement of the high level interactions defined in the framework layer is not so much performance but one of algorithmic flexibility and ease of programming. This suggests the use of a modern interpreted scripting language with object oriented features. Python also provides for excellent integration with existing scientific computation tools and libraries.

The following example could be typed in an interactive Python session, but usually saved in a script or Jupyter notebook. As usual for a Python script, an AMUSE script starts with the necessary imports,

```
from amuse.units import nbody_system
from amuse.ic.plummer import new_plummer_model
from amuse.community.huayno.interface import Huayno
```

In this case three modules are imported from the AMUSE, a module for scaleless N-body units[1], an initial condition generator and a basic N-body gravitational integrator.

The following two lines instantiate the simulation code and prepare the code for the problem at hand by setting (one of) its parameters:

```
code=Huayno()
code.parameters.epsilon_squared= (0.01 | nbody_system.length)**2
```

At this point a separate worker process for the integrator code is started and running in the background. The worker consists of the original simulation code with a layer of native code to capture calls from the framework (see Fig. 1 for a general schematic of the interface design as discussed below).

The standardized set of methods on the `Huayno` object defines the interface to the code. It is designed to communicate the *physical* quantities relevant to that domain, as opposed to numerical concepts. Codes from a given physical domain conform to the same interface, and the interface to different domains are developed along similar concepts.

The communication with the code uses a remote function protocol with different transport channels available. The default is a channel based on MPI for computations on a local compute cluster, but a channel based on the eStep platform for distributed computing is also available.

[1] The N-body unit system is useful to interact with codes that internally use a unit system where the gravitational constant $G = 1$. Other codes, for example stellar evolution codes or radiative transfer codes often work with a definite set of units, and then the normal SI unit system is used.

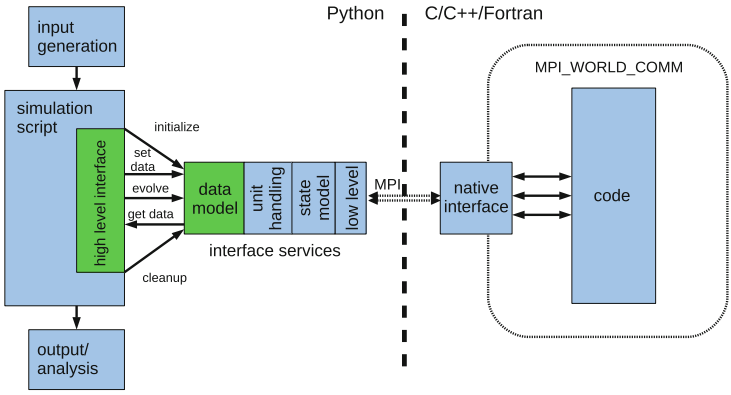


**Fig. 1.** Design of the AMUSE/OMUSE framework. This schematic representation shows the way a simulation code (the "code") is accessed from the framework. The code has a thin layer of interface functions in its native language (e.g. Fortran) which communicates through a message channel with the Python host process. The framework layer provides a number of services such as unit conversion, maintaining the simulation in a consistent state and converting the internal state data of the code to a common object oriented data representation. The user script ("simulation script") makes only generic calls to the high-level interface. Adapted from [14].

At the lowest level, the interface functions provide means to set and query model state variables and code parameters, as well as functions that change the state of the code (like initialization, triggering construction of data structures internal to the code) and functions that affect the state of the physical model (evolve forward in time).

On the basis of this low level interface a high level interface is build that provides a number of services that minimize the burden placed on the user in interacting with the code. The goal of these services is to eliminate common sources of error when running numerical simulations, by hiding the particulars of a given code and automating the interactions with the codes as much as possible.

First, the framework provides for the conversion of units and data structures. Every physical quantity that is used needs to have an attached unit. In this example the `epsilon_squared` parameter (describing the amount of smoothing applied to the gravitational force) is specified in the nbody unit of length.

The internal data representation of codes is translated to modern object oriented data structures. For this, two basic data stores are available, particle sets and grids. In the current example we proceed by initializing our model with a star cluster of 100 stars, distributed according to Plummer model (a theoretical equilibrium model used in stellar astrophysics). We send these particles to the code and define a channel to efficiently copy back the results to the particle set in the memory space of the Python script:

```
stars=new_plummer_model(100)
stars_in_code=code.particles.add_particles(stars)
channel=stars_in_code.new_channel_to(stars)
```

The framework provides another set of services to ensure that the simulation code always maintains a consistent state in terms of the numerics of the code, regardless of the order of user input, and to maintain the integrity of the simulation. The framework keeps track of the state the code is in (in terms of a predefined state model). The following call instructs the code to evolve the model for a set amount of time, but also implies a call to the low level `commit_particles` function - this function may be necessary (or not) to e.g. copy data to the GPU (for codes that use a GPU) or initialize an internal tree structure (for codes that accelerate gravitational interaction calculations using tree based data structures):

```
code.evolve_model( 0.5 | nbody_system.time)
channel.copy_attributes(["x","y","z","vx","vy","vz"])
```

At the end of the script, data is copied back to the master script's memory where it remains available for further analysis (or to be written to disk).

The high level interface in this example is the preferred way to interact with codes within AMUSE. At this level the interactions with the code are standardized as much as possible, with much of the tedious bookkeeping automated. This minimizes the possibility of programming errors and makes it easier to switch to a different model.

Currently AMUSE is distributed with more than 50 codes drawn from various astrophysical application domains (gravitational dynamics, stellar evolution, hydrodynamics and radiative transfer). In addition to this, it contains domain specific support and utility functions. These allow a researcher or student to get started quickly. Examples are: generators for initial conditions (e.g. generators for various particle models for solar systems, stellar clusters, galaxies), common analysis tools (e.g. identifying binaries, determining orbital elements) and file input/output functions. A number of common coupling schemes are also included.

## 4 Experiences

Using the AMUSE/OMUSE interfaces allows researchers (students and experienced researchers alike) to quickly develop, run and analyze computational experiments. The fact that the interfaces are homogeneous allows trivially switching between codes. Experiments are written as concise Python scripts which can easily be communicated between peers. Other use cases of the interfaces are the scripting of simulations for parameter searches and optimization, or the detection of special events. The access to the internal state of the simulation allows the integration of data analysis with a running simulation. In addition, the interfaces expose enough of the internal state such that new solvers can be developed. These can combine different physics and/or to bridge different scales. Below, we present two case studies detailing our experiences with the framework.

### 4.1 Online Data Analysis

Large-scale simulations are capable of generating enormous amounts of data. Usually, it is only possible to store a limited subset for offline analysis. The alternative is online data analysis, where the analysis code runs at the same time as the model. This offers several opportunities, including inspecting the model internal data at spatial and temporal resolutions that are not available offline.

We have created an example application for OMUSE that uses the Parallel Ocean Program (POP) and an ocean eddy tracking analysis code. The POP model is a parallel global circulation model for ocean flows [17]. POP is often used to calculate strongly eddying ocean circulation models.

The interest in ocean eddies comes from the fact that eddies transport considerable amounts of energy and mass and thus influence the dynamics of large-scale ocean circulation and the climate e.g. [5,20]. To understand eddy properties and variability, several mesoscale eddy tracking algorithms have been proposed in recent years. We have adapted a sea surface height-based eddy tracking code by Mason et al. [11]. The interface to this code allows the user to interact with the code using the high-level data structures, such as grids and units that are used in OMUSE.

Figure 2 shows the Python code of our example application for online data analysis. The application first instantiates the POP interface for high-resolution to run on a large computing cluster. After that we set the analysis interval for the eddy tracker to 7 days of simulation.

The EddyTracker is initialized using the same grid as is used in POP for the sea surface height values. From this grid object, the EddyTracker automatically extracts the coordinates of the grid points and the sea surface height values, and performs unit conversions if needed. Note that while POP is running a global simulation, the eddy tracker is set to only track the eddies in a particular region. Our application alternatingly runs the POP model for 7 simulation days and calls EddyTracker to track the eddies at the current time in the model for a full simulation year.

Figure 3 shows the output of our online eddy tracking program. In this image, we can clearly see the large anticyclonic eddies that result from the retroflection of the Agulhas Current, as well as many smaller eddies being tracked over time by the eddy tracker algorithm. The data generated by the online eddy tracker can, for example, be used to compare the statistics of the simulated eddies to the statistics of eddies found in altimetry data.

### 4.2 Multi-scale Coupling of Atmospheric LES Models to OpenIFS

As an example of the use of OMUSE for multi-scale coupling, we present here the use of the framework in a project on cloud-resolving atmospheric modelling. The project aims to couple the global atmospheric model OpenIFS [3] with a local, high-resolution (cloud-resolving) Large Eddy Simulation (LES) model, DALES [7]. The reason for this coupling is that global atmospheric models, such

```
from omuse.units import units
from omuse.ext.eddy_tracker.interface import EddyTracker
from omuse.community.pop.interface import POP

# start and initialize POP
p=POP(channel_type="distributed", mode="3600x2400x42",
      number_of_workers=592)
p.parameters. ...  # set all input files needed by POP

# set the analysis interval & initialize the EddyTracker
dt_analysis = 7 | units.day
tracker = EddyTracker(grid=p.nodes, domain="Regional",
            lonmin=0. | units.deg, lonmax=50. | units.deg,
            latmin=-45. | units.deg, latmax=-20. | units.deg,
            dt_analysis)

# evolve the simulation until the set end time
tend = p.model_time + (1 | units.yr)
while (p.model_time < tend):
    p.evolve_model(p.model_time + dt_analysis)
    tracker.find_eddies(ssh=p.nodes.ssh, rtime=p.model_time)

tracker.stop()
p.stop()
```

**Fig. 2.** This example demonstrates how to build an application that analyses data from a running simulation using OMUSE. This code executes an eddy tracking program that tracks the eddies based on sea surface height every seven days of simulation time of a running POP model.

as OpenIFS, typically cannot resolve individual clouds, as the clouds are smaller than the model grid size. The global models instead rely on parameterizations to account for processes on sub-grid scales.

Replacing a parameterization scheme (e.g. for convection) by a full microscopic model that resolves, rather than parameterizes, the small-scale process is in this context known as a superparameterization [4]. More specifically, superparameterization concerns the two-way nesting of a high-resolution model with limited spatial domain (in our case, DALES) in model columns of a global model of lower resolution (e.g. OpenIFS). Separate instances (or copies) of the high-resolution model are nested in the different model columns of the global model. This approach is seen as one possible route to understanding the feedbacks between cloud processes and climate [16] - one of the largest remaining uncertainties in climate modelling. Figure 4 shows a snapshot of a superparameterized simulation, with 72 DALES models over the Netherlands coupled to the global OpenIFS.

Both OpenIFS and DALES are implemented mainly in Fortran. For coupling the two models we considered a number of different strategies. The most straightforward might have been to directly embed DALES in the physics routines of OpenIFS. However, it was desirable to create the option of using superparameterization only for a selected number of OpenIFS model columns, to allow using a high resolution for the local models at a reasonable total computational cost. The selective superparameterization would have made load balancing in the existing parallelization of OpenIFS complicated. For this reason, and also for increased

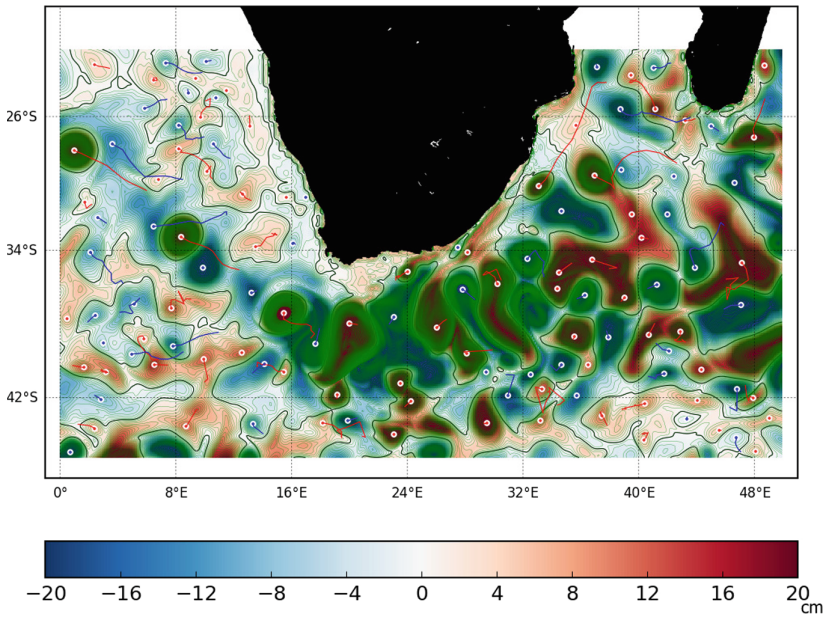


**Fig. 3.** Output of the online eddy tracking application using data from a running POP simulation, showing a region around the southern tip of Africa. The green lines show the contours between areas of different sea level anomaly values. Red indicates areas of elevated sea level, and is used to indicate anticyclonic eddies. Similarly, blue indicates a lower sea level, and is used to identify cyclonic eddies. The red or blue lines indicate the track that an eddy has traveled since it was first detected. (Color figure online)

flexibility, we chose to keep the two models separate, give each a library interface, and implement the coupling as a separate program.

A direct benefit is the convenience of writing the coupling code in Python. Performance-wise, typically more than 90% of the time of the whole simulation is spent in the DALES models. Neither the Python code nor the communication has been a significant bottle-neck so far, helped by the fact that the coupling is formulated in terms of vertical profiles and thus does not require the exchange of 3D fields.

The DALES models in our setup are time-stepped in parallel. The asynchronous function call mechanism in AMUSE works very well for this - the DALES interface contains a function to perform a time step, the coupler makes an asynchronous call to this function for every DALES model, then waits for them all to complete.

We have mainly used the Cray system at ECMWF for the simulations. One practical difficulty we encountered there is that the Cray MPI so far does not support spawning new MPI processes, which is how AMUSE normally launches its worker codes. Support for MPI spawning is scheduled to be available this year, until then we are using a work-around where all workers are launched at

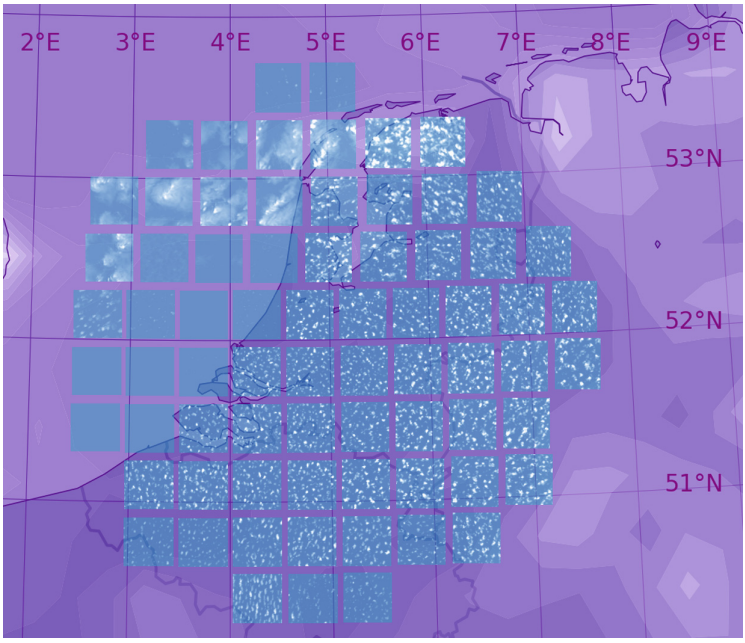


**Fig. 4.** Superparameterized weather simulation over the Netherlands, with the global OpenIFS model in purple and local, cloud-resolving DALES models in blue. Both models show the cloud fields in shades of white in the form of liquid water path. (Color figure online)

the start of the simulation as a regular MPI job, after which the appropriate MPI communicators are created. This solution is possible for us since we know how many workers are needed for a particular simulation prior to starting the simulation.

### 4.3 Testing and Validation

AMUSE and OMUSE try to foster good scientific computation practices: framework code and interfaces are tested and basic verification/ validation is done. A natural question remains is to what extent we can guarantee the correctness of the resulting simulations (see also the related discussion in [9]). Our philosophy is that this remains the responsibility of the researcher. A new AMUSE/OMUSE application should be thoroughly tested, especially where it involves new interactions between components. This is unavoidable since, while the framework does simplify the design and implementation of couplings (the "engineering" aspect), developing e.g. a new coupling involves an element of scientific inquiry (which couplings are physically sound). For example, in the OpenIFS-DALES coupling there are various ways to downscale the input variations from the global circulation model to the local LES simulation. This has a significant impact on the

results, and uncovers subtle issues about the numerical representation of both models.

### 4.4 Performance

An important concern of any coupling framework is the computational performance. The architecture of the AMUSE framework is designed with a high degree of parallelism, and individual simulation codes are often highly optimized. In the applications we have developed, the performance of the AMUSE/OMUSE framework is rarely a concern: the overhead imposed by the framework is measured to be rather small (less than a few percent [15]), However, this is strongly problem dependent. In particular massively parallel codes that use thousands of processes can generate big data transfers, and it is not difficult to formulate problems where the strength of the coupling is intrinsically so strong that very frequent communication between the component solvers is necessary.

The current design of AMUSE and OMUSE uses parallel data structures for storing the data inside parallel models. A limitation of the current design is the fact that the communication between solvers is handled by a single process user script. One of the main performance challenges that will be addressed in the near future is the development of distributed data transport. Note that such distributed communication channels would not change the semantics of the use of a channel between data structures in the user script.

## 5 Challenges

While the current implementations of AMUSE and OMUSE provide a core functionality, challenges remain for more general applications. These range from technical ones to challenges related to organizational and human factors.

**Generalization and Extension.** Currently we are working on making sure the core framework is domain agnostic, such that an implementation for another field can be more easily developed. The concrete motivation stems from the development of an hydrological version ("HyMUSE") as computational core of the eWaterCycle project [10].

OMUSE was built as an extension of AMUSE, with some of the development related to OMUSE done inside the code base of AMUSE. This brought a clearer picture of the difference between general framework components within AMUSE, and astrophysics specific code. It also brought to light that this model of development would not scale to multiple domains, as the AMUSE framework framework code would have to support and intermingle code of multiple domains. With the application to a third domain (Hydrology) it becomes more urgent to refactor the code in clearly separated general framework code and domain specific packages. This generalization of the framework involves new challenges in the coordination and prioritization of the development goals.

**Interoperability.** Over the years a number of frameworks have been developed that have some overlap and are conceptually close to the AMUSE/OMUSE system (e.g. CSDMS, [12,18]). One of the future goals is to attain a level of interoperability. For example, within the CSDMS the Basic Model Interface (BMI) has been defined as a minimal code interface to interact with a model code. BMI shares some characteristics with the AMUSE low-level interface. We have developed an automatic interface generator for codes that support BMI. This makes it possible to automatically generate low-level AMUSE interfaces and serves as a starting point for building a high-level AMUSE interface.

**Maintainability.** Another challenge involves the maintainability of the framework. This involves adapting the framework to keep it up to date with the evolving software ecosystem in on which it depends, i.e. making sure breakages from compiler updates etc. are fixed and keeping up with changes in software development practices and trends in software usage and distribution.

For sustainability of the interfaces, it is important to consider how the code is archived, especially when the AMUSE interface depends on a modified version of a third-party code (such as OpenIFS). In other cases the code of the model itself is open source and the modifications needed in the code to support the AMUSE interface are rather small.

## 6 Conclusions

The AMUSE/OMUSE system is a unique tool for scientific discovery. It encapsulates existing legacy codes into a modern simulation environment. This allows researchers to quickly translate conceptual ideas into numerical experiments. These are documented in portable scripts that can easily be communicated among peers. This lowers the barrier for verification and validation, and the framework thus aids in making simulations more reproducible. We have also shown here that the system is quite general and is easily retargeted for different scientific disciplines.

During the development of AMUSE and OMUSE we have found that the engagement of the community and domain scientist is crucial. Since the resources available for development in an academic setting are quite modest the involvement of domain scientist helps developers to focus on features that have most immediate use.

The generalization of the framework provides an opportunity - different disciplines sharing the same code base should lower the burden in the development and maintaining the framework, as well as a challenge: it becomes more difficult to coordinate development efforts and different fields may have conflicting needs - and thus attention must be paid to prioritizing the development of new features. Within our project, we found that the involvement of the Netherlands eScience Center, as an entity dedicated to the generalization of research software, provides a good focal point for these tasks. This way, scientists can spend more of their limited time on developing and testing scientific ideas.

**Acknowledgments.** FJ, DC, PS and GvdO acknowledge support from the Netherlands eScience Center (grant 027-015-G03) for their project *Towards Large-Scale Cloud-Resolving Climate Simulations*. Furthermore, acknowledgment is made for the use of ECMWF's computing and archive facilities in this project.

## References

1. Borgdorff, J., et al.: Distributed multiscale computingwith muscle 2, the multiscale coupling library and environment. J. Comput. Sci. **5**(5), 719 – 731 (2014). https://doi.org/10.1016/j.jocs.2014.04.004. http://www.sciencedirect.com/science/article/pii/S1877750314000465
2. Buis, S., Piacentini, A., Déclat, D.: The PALM Group: PALM: a computational framework for assembling high-performance computing applications. Concurrency Comput. Pract. Exp. **18**(2), 231–245 (2006)
3. Carver, G., et al.: The ECMWF OpenIFS numerical weather prediction model release cycle 40r1: description and use cases. To be submitted to GMDD (2018, in preparation)
4. Grabowski, W.W.: An improved framework for superparameterization. J. Atmos. Sci. **61**(15), 1940–1952 (2004) https://doi.org/10.1175/1520-0469(2004)061⟨1940:AIFFS⟩2.0.CO;2
5. Griffies, S.M., et al.: Impacts on ocean heat from transient mesoscale eddies in a hierarchy of climate models. J. Clim. **28**(3), 952–977 (2015)
6. Groen, D., Zasada, S.J., Coveney, P.V.: Survey of multiscale and multiphysics applications and communities. Comput. Sci. Eng. **16**(2), 34–43 (2014). https://doi.org/10.1109/MCSE.2013.47
7. Heus, T., et al.: Formulation of the Dutch atmospheric large-eddy simulation (DALES) and overview of its applications. Geosci. Model Dev. **3**(2), 415–444 (2010). https://doi.org/10.5194/gmd-3-415-2010
8. Hill, C., DeLuca, C., Suarez, M., Da Silva, A., et al.: The architecture of the earth system modeling framework. Comput. Sci. Eng. **6**(1), 18–28 (2004)
9. Hoekstra, A., Chopard, B., Coveney, P.: Multiscale modelling and simulation: a position paper. Philos. Trans. Roy. Soc. A: Math. Phys. Eng. Sci. **372**(2021), 20130377 (2014). https://doi.org/10.1098/rsta.2013.0377
10. Hut, R., van de Giesen, N., Drost, N.: The future of global is local. eWaterCycle II: bridging the gap between catchment hydrologists and global hydrologists (2018). https://meetingorganizer.copernicus.org/EGU2018/EGU2018-10614.pdf
11. Mason, E., Pascual, A., McWilliams, J.C.: A new sea surface height-based code for oceanic mesoscale eddy tracking. J. Atmos. Ocean. Technol. **31**(5), 1181–1188 (2014)
12. Peckham, S.D., Hutton, E.W., Norris, B.: A component-based approach to integrated modeling in the geosciences: the design of CSDMS. Comput. Geosci. **53**, 3–12 (2013). https://doi.org/10.1016/j.cageo.2012.04.002
13. Pelupessy, F.I., van Elteren, A., de Vries, N., McMillan, S.L.W., Drost, N., Portegies Zwart, S.: The astrophysical multipurpose software environment. Astron. Astrophys. **557**, 84 (2013)
14. Pelupessy, I., et al.: The oceanographic multipurpose software environment (omuse v1.0). Geosci. Model Dev. **10**(8), 3167–3187 (2017). https://doi.org/10.5194/gmd-10-3167-2017. https://www.geosci-model-dev.net/10/3167/2017/

15. Portegies Zwart, S., McMillan, S.L.W., van Elteren, E., Pelupessy, I., de Vries, N.: Multi-physics simulations using a hierarchical interchangeable software interface. Comput. Phys. Commun. **183**, 456–468 (2013)
16. Schneider, T., et al.: Climate goals and computing the future of clouds. Nat. Clim. Change **7**, 3 (2017). https://doi.org/10.1038/nclimate3190
17. Smith, R.D., et al.: The Parallel Ocean Program (POP) Reference Manual. Los Alamos National Laboratory, LAUR-10-01853 (2010)
18. Valcke, S.: The OASIS3 coupler: a European climate modelling community software. Geosci. Model Dev. **6**(2), 373–388 (2013). https://doi.org/10.5194/gmd-6-373-2013
19. Valcke, S., et al.: Coupling technologies for earth system modelling. Geosci. Model Dev. **5**(6), 1589–1596 (2012). https://doi.org/10.5194/gmd-5-1589-2012
20. Viebahn, J., Eden, C.: Towards the impact of eddies on the response of the southern ocean to climate change. Ocean Model. **34**(3–4), 150–165 (2010)