Easing Multiscale Model Design and Coupling with MUSCLE 3

Lourens E. Veen[1(✉)] and Alfons G. Hoekstra[2]

[1] Netherlands eScience Center, Amsterdam, The Netherlands
l.veen@esciencecenter.nl
[2] University of Amsterdam, Amsterdam, The Netherlands
a.g.hoekstra@uva.nl

Abstract. Multiscale modelling and simulation typically entails coupling multiple simulation codes into a single program. Doing this in an ad-hoc fashion tends to result in a tightly coupled, difficult-to-change computer program. This makes it difficult to experiment with different submodels, or to implement advanced techniques such as surrogate modelling. Furthermore, building the coupling itself is time-consuming. The MUltiScale Coupling Library and Environment version 3 (MUSCLE 3) aims to alleviate these problems. It allows the coupling to be specified in a simple configuration file, which specifies the components of the simulation and how they should be connected together. At runtime a simulation manager takes care of coordination of submodels, while data is exchanged over the network in a peer-to-peer fashion via the MUSCLE library. Submodels need to be linked to this library, but this is minimally invasive and restructuring simulation codes is usually not needed. Once operational, the model may be rewired or augmented by changing the configuration, without further changes to the submodels. MUSCLE 3 is developed openly on GitHub, and is available as Open Source software under the Apache 2.0 license.

Keywords: Multiscale modelling · Model coupling · Software

1 Introduction

Natural systems consist of many interacting processes, each taking place at different scales in time and space. Such multiscale systems are studied for instance in materials science, astrophysics, biomedicine, and nuclear physics [11,16,24,25]. Multiscale systems may extend across different kinds of physics, and beyond into social systems. For example, electricity production and distribution covers processes at time scales ranging from less than a second to several decades, covering physical properties of the infrastructure, weather, and economic aspects [21]. The behaviour of such systems, especially where emergent phenomena are present,

This work was supported by the Netherlands eScience Center and NWO under the e-MUSC project.

V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12142, pp. 425–438, 2020.
https://doi.org/10.1007/978-3-030-50433-5_33

may be understood better through simulation. Simulation models of multiscale systems (multiscale models for short), are typically coupled simulations: they consist of several submodels between which information is exchanged.

Constructing multiscale models is a non-trivial task. In addition to the challenge of constructing and verifying a sufficiently accurate model of each of the individual processes in the system, scale bridging techniques must be used to preserve key invariants while exchanging information between different spatiotemporal scales. If submodels that use different domain representations need to communicate, then conversion methods are required to bridge these gaps as well. Multiscale models that exhibit temporal scale separation may require irregular communication patterns, and spatial scale separation results in running multiple instances, possibly varying their number during simulation.

Once verified, the model must be validated and its uncertainty quantified (UQ) [22]. This entails uncertainty propagation (forward UQ) and/or statistical inference of missing parameter values and their uncertainty (inverse UQ). Sensitivity analysis (SA) may also be employed to study the importance of individual model inputs for obtaining a realistic result. Such analysis is often done using ensembles, which is computationally expensive especially if the model on its own already requires significant resources. Recently, semi-intrusive methods have been proposed to improve the efficiency of UQ of multiscale models [18]. These methods leave individual submodels unchanged, but require replacing some of them or augmenting the model with additional components, thus changing the connections between the submodels.

When creating a multiscale model, time and development effort can often be saved by reusing existing submodel implementations. The coupling between the models however is specific to the multiscale model as a whole, and needs to be developed from scratch. Doing this in an ad-hoc fashion tends to result in a tightly coupled, difficult-to-change computer program. Experimenting with different model formulations or performing efficient validation and uncertainty quantification then requires changing the submodel implementations, which in turn makes it difficult to ensure continued interoperability between model components. As a result, significant amounts of time are spent solving technical problems rather than investigating the properties of the system under study.

These issues can be alleviated through the use of a model coupling framework, a software framework which takes care of some of the aspects of coupling submodels together into a coupled simulation. Many coupling frameworks exist, originating from a diversity of fields [2,12,13]. Most of these focus on tightly-coupled scale-overlapping multiphysics simulations, often in a particular domain, and emphasise efficient execution on high-performance computers.

The MUSCLE framework has taken a somewhat different approach, focusing on scale-separated coupled simulation. These types of coupled simulations have specific communication patterns which occupy a space in between tightly-coupled, high communication intensity multiphysics simulations, and pleasingly parallel computations in which there is no communication between components at all. The aforementioned methods for semi-intrusive UQ entail a similar com-

munication style, but require the ability to handle ensembles of (parts of) the coupled simulation. In this paper, we introduce version 3 of the MUltiScale Coupling Library and Environment (MUSCLE 3 [23]), and explain how it helps multiscale model developers in connecting (existing) submodels together, exchanging information between them, and changing the structure of the multiscale model as required for e.g. uncertainty quantification. We compare and contrast MUSCLE 3 to two representative examples: preCICE [5], an overlapping-scale multiphysics framework, and AMUSE [20], another multiscale-oriented coupling framework.

2 Designing Coupled Simulations with the MMSF

MUSCLE 3 is based on the theory of the Multiscale Modeling and Simulation Framework (MMSF, [4,8]). The MMSF provides a systematic method for deriving the required message exchange pattern from the relative scales of the modelled processes. As an example, we show this process for a 2D simulation of In-Stent Restenosis (ISR2D, [6,7,17,19]). This model models stent deployment in a coronary artery, followed by a healing process involving (slow) cell growth and (fast) blood flow through the artery. The biophysical aspects of the model have been described extensively in the literature; here we will focus on the model architecture and communication pattern. Note that we have slightly simplified both the model (ignoring data conversion) and the method (unifying state and boundary condition updates) for convenience.

Simple coupled simulations consist of two or more sequential model runs, where the output of one model is used as an input of the next model. This suffices if one real-world process takes place before the next, or if there is otherwise effectively a one-way information flow between the modeled processes. The pattern of data exchange in such a model may be described as a Directed Acyclic Graph (DAG)-based workflow.

A more complex case is that of cyclic models, in which two or more submodels influence each other's behaviour as the simulation progresses. Using a DAG to describe such a simulation is possible, but requires modelling each executing submodel as a long sequence of state update steps, making the DAG unwieldy and difficult to analyse. Moreover, the number of steps may not be known in advance if a submodel has a variable step size or runs until it detects convergence.

A more compact but still modular representation is obtained by considering the coupled simulation to be a collection of simultaneously executing programs (components) which exchange information during execution by sending and receiving messages. Designing a coupled simulation then becomes a matter of deciding which component should send which information to which other component at which time. Designing this pattern of information exchange between the components is non-trivial. Each submodel must receive the information it needs to perform its next computation as soon as possible and in the correct form. Moreover, in order to avoid deadlock, message sending and receiving should match up exactly between the submodels.

Figure 1 depicts the derivation of the communication pattern of ISR2D according to the MMSF. Figure 1a) shows the spatial and temporal domains

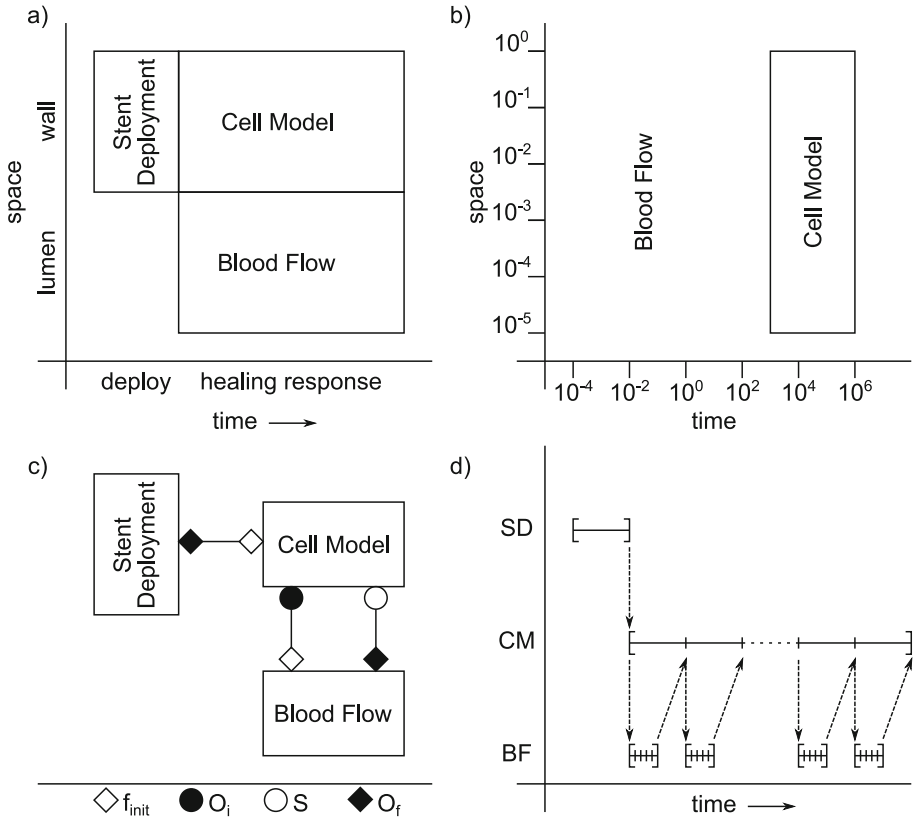

Fig. 1. a) Spatiotemporal domains (ordinal scales), b) Scale Separation Map, c) MML diagram, and d) simulation timeline of the ISR2D model.

in which the three processes comprising the model take place. Temporally, the model can be divided into a deployment phase followed by a healing phase. Spatially, deployment and cell growth act on the arterial wall, while blood flow acts on the lumen (the open space inside the artery). Figure 1b) shows a Scale Separation Map [15] for the healing phase of the model. On the temporal axis, it shows that blood flow occurs on a scale of milliseconds to a second, while cell growth is a process of hours to weeks. Thus, the temporal scales are separated [9]. Spatially, the scales overlap, with the smallest agents in the cell growth model as well as the blood flow model's grid spacing on the order of 10 μm, while the domains are both on the order of millimeters.

According to the MMSF, the required communication pattern for the coupled simulation can be derived from the above information. The MMSF assumes that each submodel executes a Submodel Execution Loop (SEL). The SEL starts with an initialisation step (f_{init}), then proceeds to repeatedly observe the state (O_i) and then update the state (S). After a number of repetitions of these two steps,

iteration stops and the final state is observed (O_f). During observation steps, (some of the) state of the model may be sent to another simulation component, while during initialisation and state update steps messages may be received.

For the ISR2D model, causality dictates that the deployment phase is simulated before the healing phase, and therefore that the final state of the deployment (O_f) is fed into the initial conditions (f_{init}) of the healing simulation. In the MMSF, this is known as a *dispatch coupling template*. Within the healing phase, there are two submodels which are timescale separated. This calls for the use of the *call* (O_i to f_{init}) and *release* (O_f to S) coupling templates.

Figure 1c) shows the resulting connections between the submodels using the Multiscale Modeling Language [10]. In this diagram, the submodels are drawn as boxes, with lines indicating conduits between them through which messages may be transmitted. Decorations at the end of the lines indicate the SEL steps (or *operators*) between which the messages are sent. Note that conduits are unidirectional.

Figure 1d) shows the corresponding timeline of execution. First, deployment is simulated, then the cell growth and blood flow models start. At every timestep of the cell growth submodel (slow dynamics), part of its state is observed (O_i) and used to initialise (f_{init}) the blood flow submodel (fast dynamics). The blood flow model repeatedly updates its state until it converges, then sends (part of) its final state (at its O_f) back to the cell growth model's next state update (S).

3 MUSCLE 3

While the above demonstrates how to design a multiscale model from individual submodels, it does not explain how to implement one. In this section, we introduce MUSCLE 3 and yMMSL, and show how they ease building a complex coupled simulation. MUSCLE 3 is the third incarnation of the MUltiScale Coupling Library and Environment, and is thus the successor of MUSCLE [14] and MUSCLE 2 [3]. MUSCLE 3 consists of two main components: `libmuscle` and the MUSCLE Manager.

Figure 2 shows how `libmuscle` and the MUSCLE Manager work together with each other and with the submodels to enact the simulation. At start-up, the MUSCLE Manager reads in a description of the model and then waits for the submodels to register. The submodels are linked with `libmuscle`, which offers an API through which they can interact with the outside world using *ports*, which are gateways through which messages may be sent and received. To start the simulation, the Manager is started first, passing the configuration, and then the submodels are all started and passed the location of the Manager.

At submodel start-up, `libmuscle` connects to the MUSCLE Manager via TCP, describes how it can be contacted by other components, and then receives a description for each of its ports of which other component it should communicate with and where it may be found. The MUSCLE Manager derives this information from the model topology description and from the registration information sent by the other submodels. The submodels then set up direct peer-to-peer network connections to exchange messages. These connections currently use TCP,

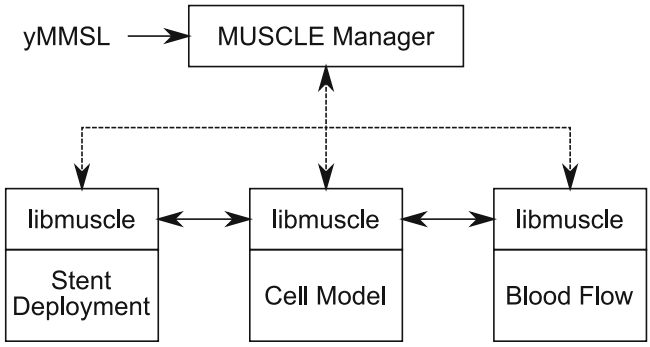

Fig. 2. MUSCLE 3 run-time architecture for ISR2D.

but a negotiation mechanism allows for future addition of faster transports without changes to user code. Submodels may use MPI for internal communication independently of their use of MUSCLE 3 for external communication. In this case, MUSCLE 3 uses a spinloop-free receive barrier to allow resource sharing between submodels that do not run concurrently. In MUSCLE 2, each non-Java model instance is accompanied by a Java minder process which handles communication, an additional complexity that has been removed in MUSCLE 3 in favour of a native `libmuscle` implementation with language bindings.

3.1 Model Description

The Manager is in charge of setting up the connections between the submodels. The model is described to the Manager using yMMSL, a YAML-based serialisation of the Multiscale Modelling and Simulation Language (MMSL). MMSL is a somewhat simplified successor to the MML [10], still based on the same concepts from the MMSF. Listing 1 shows an example yMMSL file for ISR2D. The model is described with its name, the compute elements making up the model, and the conduits between them. The name of each compute element is given, as well as a second identifier which identifies the implementation to use to instantiate this compute element. Conduits are listed in the form `component1.port1: component2.port2`, which means that any messages sent by `component1` on its `port1` are to be sent to `component2` on its `port2`. The components referred to in the `conduits` section must be listed in the `compute_elements` section. MUSCLE 3 reads this file directly, unlike MUSCLE 2 which was configured using a Ruby script that could be derived from the MML XML file.

The yMMSL file also contains settings for the simulation. These can be global settings, like `length` of the simulated artery section, or addressed to a specific submodel, e.g. `bf.velocity`. Submodel-specific settings override global settings if both are given. Settings may be of types float, integer, boolean, string, and 1D or 2D array of float.

Listing 1. yMMSL file for ISR2D

```
ymmsl_version: v0.1

model:
  name: ISR2D
  compute_elements:
    deploy: deploy_stent
    smc: cell_model
    bf: blood_flow
  conduits:
    deploy.final_state_out: smc.initial_state_in
    smc.geom_out: bf.domain_in
    bf.wss_out: smc.wss_in

settings:
  length: 1.5
  lumen_width: 1.0
  deploy.max_depth: 0.11
  bf.velocity: 0.48
  smc.re_recovered_time: 1987200    # 23 days in seconds
  # Many other settings here
```

3.2 Libmuscle

The submodels need to coordinate with the Manager, and communicate with each other. They do this using `libmuscle`, which is a library currently available in Python 3 (via pip), C++ and Fortran. Unlike in MUSCLE 2, whose Java API differed significantly from the native one, the same features are available in all supported languages. Listing 2 shows an example in Python. First, an `Instance` object is created and given a description of the ports that this submodel will use. At this point, `libmuscle` will connect to the Manager to register itself, using an instance name and contact information passed on the command line. Next, the *reuse loop* is entered. If a submodel is used as a micromodel, then it will need to run many times over the course of the simulation. The required number of runs equals the macromodel's number of timesteps, which the micromodel should not have any knowledge of if modularity is to be preserved. A shared setting could solve that, but will not work if the macromodel has varying timesteps or runs until it detects convergence. Determining whether to do another run is therefore taken care of by MUSCLE 3, and the submodel simply calls its `reuse_instance()` function to determine if another run is needed. In most cases, MUSCLE 2 relied on a global end time to shut down the simulation, which is less flexible and potentially error-prone.

Within the reuse loop is the implementation of the Submodel Execution Loop. First, the model is initialised (lines 10–17). Settings are requested from `libmuscle`, passing an (optional) type description so that `libmuscle` can generate an appropriate error message if the submodel is configured incorrectly.

Listing 2. Using libmuscle from Python

```
1  def cell_model() -> None:
2      instance = Instance({
3              Operator.F_INIT: ['initial_state_in'],
4              Operator.O_I: ['geom_out'],
5              Operator.S: ['wss_in'],
6              Operator.O_F: ['final_state_out']})
7
8      while instance.reuse_instance():
9          # F_INIT
10         length = instance.get_setting('length', 'float')
11         width = instance.get_setting('lumen_width', 'float')
12         rec_time = instance.get_setting('re_recovered_time', 'int')
13         t_max = instance.get_setting('t_max', 'float')
14
15         init_msg = instance.receive('initial_state_in')
16         t_cur = msg.timestamp
17         init_model(length, width, init_msg.data)
18
19         while t_cur + dt < t_max:
20             # O_I
21             t_next = t_cur + dt if t_cur + dt < t_max else None
22             msg_out = Message(t_cur, t_next, calc_geometry())
23             instance.send('geom_out', msg_out)
24
25             # S
26             wss_msg = instance.receive('wss_in')
27             update_state(wss_msg.data)
28             t_cur += dt
29
30         # O_F
31         instance.send('final_state', Message(t_cur, None, get_state()))
```

Note that `re_recovered_time` is specified without the prefix; `libmuscle` will automatically resolve the setting name to either a submodel-specific or a global setting. A message containing the initial state is received on the relevant port (line 15), and the submodel's simulation time is initialised using the corresponding timestamp. The obtained data is then used to initialise the simulation state in a model-specific way, as represented here by an abstract `init_model()` function (line 17).

Next is the iteration part of the SEL, in which the state is repeatedly observed and updated (lines 19–28). In addition to the simulation time corresponding to the current state, the timestamp for the next state is calculated here (line 21). This is unused here, but is required in case two submodels with overlapping timescales are to be coupled [4] and so improves reusability of the model. In ISR2D's O_i operator, the current geometry of the artery is calculated and sent on the `geom_out` port (lines 22–23). Next, the wall shear stress is received and used in the model's state update, after which the simulation time is incremented and the next observation may occur (lines 26–28). Once the final state is reached, it is sent on the corresponding port (line 31). In this example, this port is not

connected, which causes MUSCLE 3 to simply ignore the send operation. In practice, a component would be attached which saves this final state to disk, or postprocesses it in some way, possibly via an in-situ/in-transit analysis framework.

Message data may consist of floating point numbers, integers, Booleans, strings, raw byte arrays, or lists or dictionaries containing these, as well as grids of floating point or integer numbers or Booleans, where MUSCLE 2 only supported 1D arrays of numbers. Internally, MUSCLE 3 uses MessagePack for encoding the data before it is sent.

4 Uncertainty Quantification

Uncertainty Quantification of simulation models is an important part of their evaluation. Intrusive methods provide an efficient solution in some cases, but UQ is most often done using Monte Carlo (MC) ensembles. An important innovation in MUSCLE 3 compared to MUSCLE 2 is its flexible support for Monte Carlo-based algorithms. This takes the form of two orthogonal features: instance sets and settings injection.

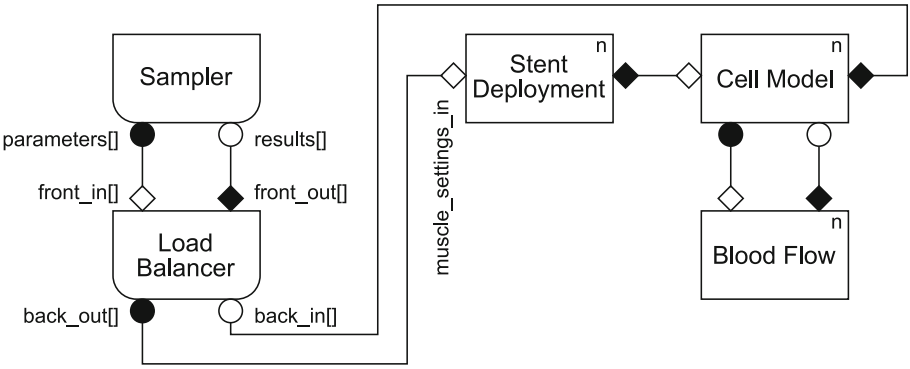

Fig. 3. MMSL diagram for Uncertainty Quantification of ISR2D.

Figure 3 shows an MMSL diagram of a Monte Carlo forward UQ of ISR2D. The simulation has been augmented with a sampler and a load balancer, and there are now multiple instances of each of the three submodels. The sampler samples the uncertain parameters from their respective distributions, and generates a settings object for each ensemble member. These objects are sent to the load balancer, which distributes them evenly among the available model instances. The settings are then sent into a special port on the submodel instances named `muscle_settings_in`, from where the receiving `libmuscle` automatically overlays them on top of the centrally provided settings. The settings are then transparently passed on to the corresponding other submodel

instances. Final results are passed back via the load balancer to the sampler, which can then compute the required statistics.

To enable communication with sets of instances, MUSCLE 3 offers *vector ports*, recognisable by the square brackets in the name. A vector port allows sending or receiving on any of a number of *slots*, which correspond to instances if the port is connected to a set of them. Vector ports may also be connected to each other, in which case each sending slot corresponds to a receiving slot. In the example, the Sampler component resizes its `parameters[]` port to the number of samples it intends to generate, then generates the settings objects and sends one on each slot. The load balancer receives each object on the corresponding slot of its `front_in` port, and passes it on to a slot on its `back_out` port. It is then received by the corresponding ensemble member, which runs and produces a result for the load balancer to receive on `back_in`. The load balancer then reverses the earlier slot mapping, and passes the result back to the sampler on the same slot on its `front_out` that the sampler sent the corresponding settings object on.

With the exception of the mapping inside the load balancer, all addressing in this use case is done transparently by MUSCLE 3, and components are not aware of the rest of the simulation. In particular, the submodels are not aware of the fact that they are part of an ensemble, and can be completely unmodified.

5 Discussion

An important advantage of the use of a coupling framework is the increase in modularity of the model. In MUSCLE 3, submodels do not know of each other's existence, instead communicating through abstract ports. This gives a large amount of flexibility in how many submodels and submodel instances there are and how they are connected, as demonstrated by the UQ example. Modularity can be further improved by inserting helper components into the simulation. For instance, the full ISR2D model has two *mappers*, components which convert from the agent-based representation of the cell model to the lattice-based representation of the blood flow model and back. These are implemented in the same way as submodels, but being simple functions only implement the `F_INIT` and `O_F` parts of the SEL. The use of mappers allows submodels to interact with the outside world on their own terms from a semantic perspective as well as with respect to connectivity. Separate scale bridging components may be used in the same way, except converting between scales rather than between domain representations.

Other coupling libraries and frameworks exist. While a full review is beyond the scope of this paper (see e.g. [12]), we provide a brief comparison here with two other such frameworks in order to show how MUSCLE 3 relates to other solutions.

preCICE is a framework for coupled multiphysics simulations [5]. It comes with adapters for a variety of CFD and finite element solvers, as well as scale bridging algorithms and coupling schemes. Data is exchanged between submodels in the form of variables defined on meshes, which can be written to by one

component and read by another. Connections are described in an XML-based configuration file. Like MUSCLE 3, preCICE links submodels to the framework by adding calls to a library to them. For more generic packages, a more extensive adapter is created to enable more configurability. Submodels are started separately, and discover each other via files in a known directory on a shared file system, after which peer-to-peer connections are set up.

preCICE differs from MUSCLE 3 in that it is intended primarily for scale-overlapping, tightly-coupled physics simulations. MUSCLE 3 can do this as well, but is mainly designed for loosely-coupled multiscale models of any kind. For instance, it is not clear how an agent-based cell simulation as used in ISR2D would fit in the preCICE data model. MUSCLE 3's central management of model settings and its support for sets of instances allows it to run ensembles, thus providing support for Uncertainty Quantification. preCICE does not seem to have any features in this direction.

The Astrophysical Multipurpose Software Environment (AMUSE) is a framework for coupled multiscale astrophysics simulations [20,25]. It comprises a library of well-known astrophysics models wrapped in Python modules, facilities for unit handling and data conversion, and infrastructure for spawning these models and communicating with them at runtime.

Data exchange between AMUSE submodels is in the form of either grids or particle collections, both of which store objects with arbitrary attributes. With respect to linking submodels, AMUSE takes the opposite approach to MUSCLE 3 and preCICE. Instead of linking the model code to a library, the model code is made into a library, and where MUSCLE 3 and preCICE have a built-in configurable coupling paradigm, in AMUSE coupling is done by an arbitrary user-written Python script which calls the model code. This script also starts the submodels, and performs communication by reading and writing to variables in the models.

Linking a submodel to AMUSE is more complex than doing this in MUSCLE 3, because an API needs to be implemented that can access many parts of the model. This API enables access to the model's parameters as well as to its state. AMUSE comes with many existing astrophysics codes however, which will likely suffice for most users. Coupling via a Python script gives the user more flexibility, but also places the responsibility for implementing the coupling completely on the user. Uncertainty quantification could be implemented, although scalability to large ensembles may be affected by the lack of peer-to-peer communication.

6 Conclusions and Future Work

MUSCLE 3, as the latest version of MUSCLE, builds on almost fourteen years of work on the Multiscale Modelling and Simulation Framework and the MUSCLE paradigm. It is mainly designed for building loosely coupled multiscale simulations, rather than scale-overlapping multi-physics simulations. Models are

described by a yMMSL configuration file, which can be quickly modified to change the model structure. Linking existing codes to the framework can be done quickly and easily due to its library-based design. Other frameworks have more existing integrations however. Which framework is best will thus depend on which kind of problem the user is trying to solve.

MUSCLE 3 is Open Source software available under the Apache 2.0 license, and it is being developed openly on GitHub [23]. Compared to MUSCLE 2, the code base is entirely new and while enough functionality exists for it to be useful, more work remains to be done. We are currently working on getting the first models ported to MUSCLE 3, and we plan to further extend support for Uncertainty Quantification, implementing model components to support the recently-proposed semi-intrusive UQ algorithms [18]. We will also finish implementing semi-intrusive benchmarking of models, which will enable performance measurement and support performance improvements as well as enabling future static scheduling of complex simulations. Other future features could include dynamic instantiation and more efficient load balancing of submodels in order to support the Heterogeneous Multiscale Computing paradigm [1].

References

1. Alowayyed, S., et al.: Patterns for high performance multiscale computing. Fut. Gener. Comput. Syst. **91**, 335–346 (2019). https://doi.org/10.1016/j.future.2018.08.045
2. Babur, O., Verhoeff, T., Van Den Brand, M.G.J.: Multiphysics and multiscale software frameworks: an annotated bibliography. Computer Science Reports, Technische Universiteit Eindhoven (2015)
3. Borgdorff, J., et al.: Distributed multiscale computing with 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
4. Borgdorff, J., Falcone, J.L., Lorenz, E., Bona-Casas, C., Chopard, B., Hoekstra, A.G.: Foundations of distributed multiscale computing: formalization, specification, and analysis. J. Parallel Distrib. Comput. **73**(4), 465–483 (2013). https://doi.org/10.1016/j.jpdc.2012.12.011
5. Bungartz, H.J., et al.: preCICE a fully parallel library for multi-physics surface coupling. Comput. Fluids **141**, 250–258 (2016). http://www.sciencedirect.com/science/article/pii/S0045793016300974
6. Caiazzo, A., et al.: A complex automata approach for in-stent restenosis: two-dimensional multiscale modelling and simulations. J. Comput. Sci. **2**(1), 9–17 (2011). https://doi.org/10.1016/j.jocs.2010.09.002
7. Caiazzo, A., et al.: Towards a complex automata multiscale model of in-stent restenosis. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5544, pp. 705–714. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01970-8_70
8. Chopard, B., Borgdorff, J., Hoekstra, A.G.: A framework for multi-scale modelling. Phil. Trans. Roy. Soc. A Math. Phys. Eng. Sci. **372**(2021), 20130378 (2014). https://doi.org/10.1098/rsta.2013.0378
9. Weinan, E.: Principles of Multiscale Modeling. Cambridge University Press, Cambridge (2011)

10. Falcone, J.L., Chopard, B., Hoekstra, A.: MML: towards a multiscale modeling language. Procedia Comput. Sci. **1**(1), 819–826 (2010). https://doi.org/10.1016/j.procs.2010.04.089. iCCS 2010
11. Gaston, D.R., et al.: Physics-based multiscale coupling for full core nuclear reactor simulation. Ann. Nucl. Energy **84**, 45–54 (2015). https://doi.org/10.1016/j.anucene.2014.09.060. Multi-Physics Modelling of LWR Static and Transient Behaviour
12. Groen, D., Knap, J., Neumann, P., Suleimenova, D., Veen, L., Leiter, K.: Mastering the scales: a survey on the benefits of multiscale computing software. Phil. Trans. Roy. Soc. A Math. Phys. Eng. Sci. **377**(2142), 20180147 (2019). https://doi.org/10.1098/rsta.2018.0147
13. Groen, D., Zasada, S., Coveney, P.: Survey of multiscale and multiphysics applications and communities. Comput. Sci. Eng. **16**, (2012). https://doi.org/10.1109/MCSE.2013.47
14. Hegewald, J., Krafczyk, M., Tölke, J., Hoekstra, A., Chopard, B.: An agent-based coupling platform for complex automata. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008. LNCS, vol. 5102, pp. 227–233. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69387-1_25
15. Hoekstra, A.G., Lorenz, E., Falcone, J.-L., Chopard, B.: Towards a complex automata framework for multi-scale modeling: formalism and the scale separation map. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007. LNCS, vol. 4487, pp. 922–930. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72584-8_121
16. Karakasidis, T.E., Charitidis, C.A.: Multiscale modeling in nanomaterials science. Mater. Sci. Eng. C **27**(5), 1082–1089 (2007). https://doi.org/10.1016/j.msec.2006.06.029. eMRS 2006 Symposium A: Current Trends in Nanoscience - from Materials to Applications
17. Nikishova, A., Veen, L., Zun, P., Hoekstra, A.G.: Semi-intrusive multiscale metamodelling uncertainty quantification with application to a model of in-stent restenosis. Phil. Trans. Roy. Soc. A Math. Phys. Eng. Sci. **377**(2142), 20180154 (2019). https://doi.org/10.1098/rsta.2018.0154
18. Nikishova, A., Hoekstra, A.G.: Semi-intrusive uncertainty propagation for multiscale models. J. Comput. Sci. **35**, 80–90 (2019). https://doi.org/10.1016/j.jocs.2019.06.007
19. Nikishova, A., Veen, L., Zun, P., Hoekstra, A.G.: Uncertainty quantification of a multiscale model for in-stent restenosis. Cardiovasc. Eng. Technol. **9**(4), 761–774 (2018). https://doi.org/10.1007/s13239-018-00372-4
20. Pelupessy, F.I., van Elteren, A., de Vries, N., McMillan, S.L.W., Drost, N., Zwart, S.F.P.: The astrophysical multipurpose software environment. Astron. Astrophys. **557**, 84 (2013). https://doi.org/10.1051/0004-6361/201321252
21. Ringkjøb, H.K., Haugan, P.M., Solbrekke, I.M.: A review of modelling tools for energy and electricity systems with large shares of variable renewables. Renew. Sustain. Energy Rev. **96**, 440–459 (2018). https://doi.org/10.1016/j.rser.2018.08.002
22. Roy, C.J., Oberkampf, W.L.: A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing. Comput. Methods Appl. Mech. Eng. **200**(25), 2131–2144 (2011). https://doi.org/10.1016/j.cma.2011.03.016
23. Veen, L.: MUSCLE 3 (2019). https://doi.org/10.5281/zenodo.3260941, https://github.com/multiscale/muscle3

24. Walpole, J., Papin, J.A., Peirce, S.M.: Multiscale computational models of complex biological systems. Ann. Rev. Biomed. Eng. **15**(1), 137–154 (2013). https://doi.org/10.1146/annurev-bioeng-071811-150104. pMID: 23642247
25. Zwart, S.F.P., McMillan, S.L.W., van Elteren, A., Pelupessy, F.I., de Vries, N.: Multi-physics simulations using a hierarchical interchangeable software interface. Comput. Phys. Commun. **184**(3), 456–468 (2013). https://doi.org/10.1016/j.cpc.2012.09.024