Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1097 – 1105

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Multiscale computing with the multiscale modeling library and runtime environment

Joris Borgdorff[a,*], Mariusz Mamonski[b], Bartosz Bosak[b], Derek Groen[c], Mohamed Ben Belgacem[d], Krzysztof Kurowski[b], Alfons G. Hoekstra[a]

[a] *Computational Science, Faculty of Science, University of Amsterdam, Amsterdam, the Netherlands*
[b] *Poznań Supercomputing and Networking Center, Poznań, Poland*
[c] *Centre for Computational Science, University College London, London, United Kingdom*
[d] *CUI, University of Geneva, Carouge, Switzerland*

Abstract

We introduce a software tool to simulate multiscale models: the Multiscale Coupling Library and Environment 2 (MUSCLE 2). MUSCLE 2 is a component-based modeling tool inspired by the multiscale modeling and simulation framework, with an easy-to-use API which supports Java, C++, C, and Fortran. We present MUSCLE 2's runtime features, such as its distributed computing capabilities, and its benefits to multiscale modelers. We also describe two multiscale models that use MUSCLE 2 to do distributed multiscale computing: an in-stent restenosis and a canal system model. We conclude that MUSCLE 2 is a notable improvement over the previous version of MUSCLE, and that it allows users to more flexibly deploy simulations of multiscale models, while improving their performance.

Keywords: multiscale modeling, distributed multiscale computing, MUSCLE

1. Introduction

Multiscale modeling is a way to gain knowledge about complex systems by explicitly modeling the interaction between phenomena on different scales. It has had attention of diverse research fields [1], amongst others biomedicine [2], biology [3], physics [4], chemistry [5] and earth sciences [6]. The need for a general computational framework that is able to run these types of simulations is expressed by several authors [7–9].

The aspiration for distributed multiscale computing has given impulse to the Multiscale Coupling Library and Environment 2 (MUSCLE 2), which builds upon an earlier environment built by Hegewald *et al.* [10]. It is a portable and lightweight framework to implement and execute multiscale models, and if needed to run them on distributed resources. An overview of MUSCLE is shown in Figure 1.

*Corresponding author
Email addresses: J.Borgdorff@uva.nl (Joris Borgdorff), mamonski@man.poznan.pl (Mariusz Mamonski), bbosak@man.poznan.pl (Bartosz Bosak), d.groen@ucl.ac.uk (Derek Groen), mohamed.benbelgacem@unige.ch (Mohamed Ben Belgacem), krzysztof.kurowski@man.poznan.pl (Krzysztof Kurowski), A.G.Hoekstra@uva.nl (Alfons G. Hoekstra)

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.275

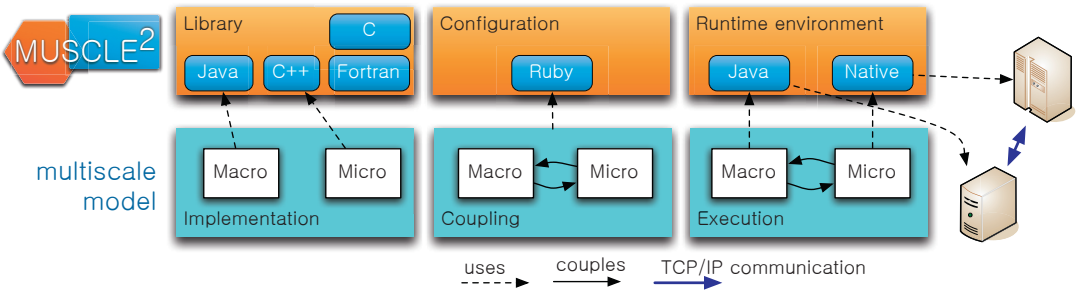

Figure 1: Overview of MUSCLE 2: with the *library*, submodels are implemented with any of the APIs, with the *configuration* those submodels are instantiated and coupled, and in the *runtime environment* they can be executed on a variety of machines.

It assumes that a multiscale model is split into multiple coupled single scale submodels as proposed by the Multiscale Modeling and Simulation Framework [11–13], preceded by the Complex Automata theory [7, 8]. As a result, MUSCLE is a component-based framework, where each submodel has inputs and outputs that can be coupled in a general way. It recognizes that submodels execute on different temporal scales and keeps simulation time between submodels in sync when they communicate. So-called mappers apply scale bridging to in-transit data so that the submodels themselves really have a single scale without knowledge of other single scale models that may be part of the overall multiscale model.

From a runtime perspective on MUSCLE 2, the submodels may be written in Java, C, C++, or Fortran. Submodels may be implemented with OpenMP or MPI and only need MUSCLE (and consequently Java and Ruby) to be run on a node reachable with TCP/IP. Submodels themselves can be as computationally costly as needed. Within one simulation, one submodel could for instance use hundreds of cores on a supercomputer, whereas another may have to make use of GPU-computing, and yet another needs high I/O performance. Execution between distributed resources and bypassing of firewalls is made possible by the message forwarder MUSCLE Transport Overlay, which is developed and packaged with MUSCLE. This is a change from traditional model coupling toolkits such as CCaffeine [14, 15] or the Model Coupling Toolkit [16, 17], which are not focussed on multiscale modeling and do not tend to run in a distributed fashion.

In this contribution, we will describe the way in which a multiscale model can be implemented with MUSCLE 2 in Section 2, and what runtime options it possesses in Section 3, also in comparison with the original MUSCLE. Finally, Section 4 shows how it has been applied to a multiscale model of in-stent restenosis. MUSCLE 2 is publicly available at `http://apps.man.poznan.pl/trac/muscle`.

2. Coupling library

The foundations that MUSCLE is based on are described by the Multiscale Modeling and Simulation Framework (MMSF) [11–13] and their computational descriptions in the multiscale modeling language (MML) [13, 18]. To better understand further sections, we will shortly repeat the contents of the framework before we introduce the features of MUSCLE. The API of MUSCLE 2 as well as part of its implementation is derived from MUSCLE 1 [10], which was based on the Complex Automata theory [7, 8].

2.1. Theoretical background

At the end of the modeling stage, MMSF suggests that a multiscale model should be built up out of coupled single scale models, or submodels. Moreover, a multiscale model should have a notion of time, and the time of different submodels has to remain consistent. These single scale models should be dependent only on their input and output, and not of other single scale models. As a consequence, scale bridging has to be done between single scale models, to make sure that they receive data that are relevant on their scale. For example, a micro-submodel will usually not need to process data about the

entire domain, but only needs to receive a small portion of the domain to do its computations. The macro-submodel on the other hand should not concern itself with splitting up the data so that it exactly matches the domain of the micro-submodels. The inputs and outputs of submodels are coupled with so-called conduits.

Scale bridging methods can be applied to data that is being sent over the conduits, either with conduit filters for data transformation or time averaging or interpolation, or by mappers, which may combine the data of multiple conduits, and send multiple outputs. To make a model self-contained, sources and sinks may also be attached to conduits so that submodels can be tested in isolation or external data sources can be consulted. In the example of the micro-submodels and a macro-submodel, the macro-submodel would send its domain over a conduit. The domain would then be split into multiple subdomains in a model-specific way, and divided amongst the micro-submodels. These micro-submodels would send their data to a mapper again, which would combine their data into one dataset and send that as a single input to the macro-submodel.

2.2. Library

Developers can implement submodels of a multiscale model with the Java, C, C++, or Fortran API, or a combination thereof. Among them, the Java API is closest to the MUSCLE core and as such is the most expressive. It offers an API for sending and receiving any type of objects, advanced logging, output redirection, time manipulation, and formal MMSF constructs such as formal submodels, mappers, conduit filters, sources, and sinks. In complex topologies, simulation time can be manipulated to ensure that all submodels are processing at the correct simulation time. In C++, free-form submodels and mappers can be written, which can send and receive strings, arrays, matrices, maps and lists, and do logging and output redirection. For C and Fortran the functionality is similar but only arrays and strings can be sent and received. For C++, C, or Fortran an accompanying Java class can be written that handles the more advanced capabilities of MUSCLE, such as advanced data structures or time manipulation. For each of the functionalities, example code is provided with MUSCLE and documentation is provided on the public MUSCLE webpage[1].

Only little code is needed to write a submodel or filter in MUSCLE, and it is straightforward to integrate in existing code, since only the send and receive statements are necessary to add. In Java, the statements to receive data from an input port `dataInput` and send it to a different port `dataOuput` would look like:

```
double[] data = (double[])in("dataInput").receive();
out("dataOutput").send(data);
```

The names of the ports are arbitrary, but they will be used for the coupling in the next section. For more advanced type-checking, each port can be initialized beforehand with the respective Java classes they should receive.

Programming languages that do not have a MUSCLE 2 API, such as Ruby, Python or Scala, may make use of MUSCLE by writing an intermediate interface, at the sacrifice of slightly reducing portability and introducing dependencies to other software.

Compared to MUSCLE 1, the submodel implementation may be much more succinct, by handling certain default initialization methods in the runtime system. Especially the C++ code is much clearer, since it no longer requires use of the Java Native Interface. The C and Fortran routines are newly added. By separating the implementation and library part of MUSCLE 1, MUSCLE 2 now also does not confront the user with MUSCLE implementation details, such as the interface with the Java runtime environment in C or C++, or the Java Agent Development Environment (JADE) library in the Java code. A beneficial side-effect is that code written by users is less susceptible to breaking due to changes in the runtime environment. Additionally, integrating MUSCLE into existing code has thus been simplified.

[1] MUSCLE 2 documentation: `http://apps.man.poznan.pl/trac/muscle/wiki/Documentation`

2.3. Configuration

The parameterization and setting up the coupling topology of the multiscale model is done in a Ruby file with MUSCLE helper classes. From the legacy of MUSCLE 1, this is called the CxA (Complex Automata) file. Any of the Ruby language features may be used to make the final coupling topology, for instance precomputing a mesh-like network, reading environment variables, or reading the topology from a separate file. Conduit filters are applied by adding them as an array to any conduit.

Parameters may be given as submodel local variables or as global variables, to make sure that all submodels use the same domain definitions for instance. Again, the file and thus the parameters may be scripted or precomputed. Time and space scales can be written in human readable terms (e.g., `'2 hr'`, `'1 meter'`) or in SI notation (e.g., `'1 ps'`, `'4.8E-3 m'`), without loss of precision.

If convenient, the Java classpath, library path, or the MPI executable can be specified in the configuration file. This will, however, make the configuration file partially dependent on the runtime environment. To make it more independent from the runtime environment, environment variables can be substituted in these paths.

An example of a configuration file is given below, with instances Macro and Micro coupled only from Macro to Micro.

```
# instance name and class name
cxa.add_instance('Macro', 'nl.uva.cs.Macro')

# C++ submodels can use the MUSCLE NativeKernel that will manage the executable
cxa.add_instance('Micro', 'muscle.core.standalone.NativeKernel')
# Let the NativeKernel know where the executable is
cxa.env['Micro:command'] = ENV['MODEL_HOME'] + '/bin/micro'

# Set the time scales
cxa.env['max_timesteps'] = '2 hr'
cxa.env['Macro:dt'] = '4 min'
cxa.env['Micro:dt'] = '1 ms'
cxa.env['Micro:T'] = '1 s'

# Attach Macro.data_out to Micro.data_in
cxa.cs.attach('Macro' => 'Micro') {
  tie('dataOutput', 'dataInput')
}
```

3. Runtime environment

The runtime environment of MUSCLE 2 is portable and designed for distributed computing. Compared to the release of MUSCLE 1, MUSCLE 2 excels in portability, distributed computing, performance, and usability. It can be executed on a local machine or on supercomputers, through the command-line interface or with queueing systems [19]. Data transmission is decentralized so communication performance can be optimized by the user by running closely tied submodels on machines that are physically close. To launch a submodel or a set of submodels, the user runs the `muscle2` command. When this command is invoked, as shown in Figure 2, a Ruby script first parses the command-line arguments and evaluates the configuration file, passing a fixed coupling description to the MUSCLE core. One or more submodels are then started by a so-called Local Manager, which runs in the Java Virtual Machine. The tasks of the Local Manager are to check for error conditions within a single Java Virtual Machine and to listen for data connections. One so-called Simulation Manager per multiscale simulation acts as a white pages service and it is the only centralized component of MUSCLE. To limit its runtime overhead, results of queries to the Simulation Manager are cached by the Local Managers.

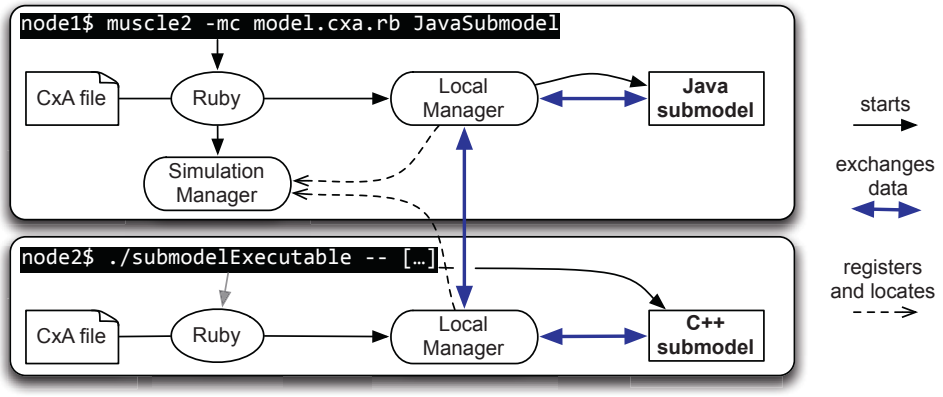

Figure 2: Example of the execution of a multiscale model containing a Java submodel and a C++ submodel. In the top left corner of each rounded rectangle is the command that is executed on different nodes. In the top rectangle, a Java submodel and the Simulation Manager are started in a single command. In the bottom rectangle, an executable is directly started, and Ruby (and thus MUSCLE) is started indirectly through the initialization method of the MUSCLE 2 API.

To keep submodels and their runtime behavior separated, each submodel is managed by its own instance controller. The instance controller registers itself with the Simulation Manager and handles any errors that a submodel might cause. The controller is also an intermediary for any messages that a submodel sends or receives. For C++, C, and Fortran submodels, the controller first converts data to Java objects and then sends it to other submodels. If necessary, it also starts the executable with the submodel, and checks if it keeps running. In the current implementation, each controller runs in its own thread, so that each submodel does its computations independently. Consequently, a limit to the number of threads in many operating systems (for example, a limit of about 2000 threads per process on OS X) causes the number of submodels per Local Manager to be restricted. For most purposes, where submodels do heavy calculations, this is not a problem, since the CPU-cores will be overtaxed far below this limit. However, for instance a network simulation of a multitude of very simple nodes might find it necessary to group several nodes into a single submodel to limit the number of created threads.

Exceptions in most parts of the system work fail-fast, so that if a submodel fails with an exception, all other submodels are also halted. The instance controller does this by notifying the Local Manager, all attached conduits, and the Simulation Manager, so that all submodels in the current Java Virtual Machine are halted, as well as attached submodels in other Java Virtual Machines. Submodels that are started after the exception occurred are notified by the Simulation Manager. The reason for this behavior is to limit the amount of runtime resources consumed after a part of the simulation has become invalid.

A few different paths of communication exist within MUSCLE. First of all, any messages within the same Local Manager are passed through shared memory, with a copy operation if needed to guarantee isolation of data of different submodels. Messages between one Local Manager and another use the serialization library MessagePack [20] over TCP/IP, because of its small encoding length and high speed. Between instance controllers and C/C++/Fortran submodels the XDR [21] serialization library is used, which has a higher computational overhead but is widely available, since it is installed with most *nix/BSD operating systems.

The dependencies for the runtime environment are Ruby and Java, to be installed on a node with a direct TCP/IP connection to the node that should be executed on. Installation is done with a single command, with only the CMake build tool and basic build environment required.

The command-line interface for MUSCLE is clean and simple. For instance, the command

```
muscle2 --main --allinstances --cxa model.cxa.rb
```

runs the Simulation Manager (`--main`) and a Local Manager with all instances (`--allinstances`) using the configuration file `model.cxa.rb` (`--cxa`).

3.1. Distributed computing

When heterogeneous or large scale computing is considered, distributed computing is an option. However, many HPC systems use a private IPv4 addressing scheme which is incompatible with direct communications between systems. Therefore, the MUSCLE Transport Overlay (MTO) is packaged with MUSCLE 2. The MTO is started on all systems that must participate in a simulation. Using a port-mapping technique, data transfers that are intended for different clusters are then forwarded by the MTO.

MTO relies on the ASIO (ASynchronous Input Output) subsystem of the Boost library. There is an ongoing effort to integrate MPWide [22] into the MTO to increase messaging speed between different clusters and to remove the dependency on Boost.

To facilitate cross-cluster simulations, MUSCLE is integrated with the QosCosGrid middleware stack [23]. This stack provides automation of the process of submitting cross-cluster simulations by: co-allocating resources on multiple sites (using the Advance Reservation mechanism); staging input/output files to/from every system involved in the simulation; managing the submission of sub-jobs; providing a locator service; and finally, allowing to have a peek at the current output of every submodel from a single location.

To run submodel Macro on node1 and Micro on node2 with the MTO, the following two commands are sufficient, given that the MTO has been set up:

```
node1$ muscle2 -mc model.cxa.rb --intercluster Macro
node2$ muscle2 --manager node1:9000 -c model.cxa.rb --intercluster Micro
```

Here, the Simulation Manager is started only once, and other MUSCLE execution just specifies the managers location and port. If there is a direct TCP/IP connection between node1 and node2, the MTO is not needed, nor is the `--intercluster` argument.

3.2. Comparison with the previous version of MUSCLE

In contrast with the previous MUSCLE implementation, MUSCLE 2 is compatible with Ruby versions 1.8.7 and 1.9.x, and Java 6 and 7. The number of Java library dependencies has been drastically decreased. For instance, it does not rely on the Java Agent Development Environment (JADE) for its communication, which makes it possible to optimize communication protocols and serialization algorithms. In addition, without JADE, MUSCLE 2 has more control over the initialization and finalization of a simulation and does so with less overhead.

Unlike its predecessor, it does not use the Java Native Interface, since this proved much less portable and user-friendly than providing a separate C++ library. Moreover, this C++ library is compatible with MPI and OpenMP, which the JNI interface was not.

The build system has been generalized using CMake as an engine, and it is now customizable per site by simply setting environment variables. Likewise, runtime settings such as library paths are now recognized as environment variables by MUSCLE 2, to make it more customizable for middleware.

Quantitatively, the message latency of MUSCLE 2 is 45 times lower, at 15 μs, the message throughput is five times higher at 1.9 GiB/s, and the initialization and finalization overhead 6.6 times lower at 0.68 seconds. The experiments for these results were conducted with a basic back-and-forth messaging Java code, sending messages with sizes from 1 kB to 8 MB, taking the minimum communication time as a latency and using a linear regression to get the throughput. Startup times are calculated by starting these submodels, let them send one empty message and quit. This was measured on an Apple iMac with a dual-core Intel i3 3.2 GHz processor.

MUSCLE 2 is also able to handle larger messages, up to a gigabyte, while MUSCLE 1 is not made to handle messages larger than 10 MB.

4. Use cases

Within the MAPPER project [24], at least four applications are successfully using MUSCLE 2: a cardiovascular biomedical model of in-stent restenosis (ISR3D) [25], a hydrological canal system model [26], a systems biology optimizer of a gene regulatory network, and a high-energy physics plasma simulator.

The model of in-stent restenosis couples, amongst others, smooth muscle cell proliferation in the coronary artery wall to the blood flow through the vessel [25]. This interaction is hypothesized to be associated to a restenosis after stenting [27]. The blood flow submodel code is highly parallelized using MPI, while the smooth muscle cell proliferation code is less scalable and uses OpenMP. It is successfully being run on distributed computing resources with MUSCLE 2 [28, 29]. Before MUSCLE 2, the model did run in specifically set up environments but not on e-Infrastructure in general.

Water management is a main concern in our modern society, in particular for water supply, electricity production and transport. The canal application simulates canal systems like irrigation canals or rivers by coupling canal segments together through water junctions. Due to the size of the canal system and the large variation of the flow, a fully resolved 3D free surface flow computation is not feasible and, thus, a multiscale computational approach is needed [26]. For example, some canal sections where the flow is stable can simply be modeled by a 1D shallow model, whereas other sections need a 3D free surface model to precisely capture the flow properties, such as studying the water behavior around a gate. With tools developed in the MAPPER project, notably MUSCLE 2, those different sections can be connected and easily executed on distributed grid infrastructure.

Each of these models have submodels with high computational demands, ranging from tens up to thousands of CPU cores. Moreover, they transfer messages with sizes from less than 1 MB up to 100 MB. In the case of ISR3D, the overhead of using MUSCLE 2 has been shown to be insignificant compared to the computational cost of the submodels [29].

5. Conclusions and Future Work

We have presented the design and features of the MUSCLE 2 coupling environment and reviewed its application in the multiscale modeling of in-stent restenosis and a canal system. MUSCLE 2 is based on the strong foundations of the multiscale modeling and simulation framework, as well as the multiscale modeling language. Because of MUSCLE 2's modular setup, which clearly separates the API, the coupling, and the runtime environment, users can easily modify parts of a multiscale model in a plug and play fashion. A multiscale model, consisting of two or more coupled single scale models, can be conveniently deployed and executed on a set of distributed computing resources. We conclude that MUSCLE 2 is a valuable addition for multiscale modeling and simulation in cases where flexibility and performance are important, and especially when the multiscale model relies on two or more different simulation codes.

Acknowledgements

This research presented in this contribution is partially supported by the MAPPER project, which receives funding from the EC's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° RI-261507.

References

[1] D. Groen, S. J. Zasada, P. V. Coveney, Taxonomy of Multiscale Computing Communities, in: e-Science Workshops (eScienceW), 2011 IEEE Seventh International Conference on, 5-8 Dec. 2011, Stockholm, Sweden, 2011, pp. 120–127. doi:10.1109/eScienceW.2011.11.

[2] P. M. A. Sloot, A. G. Hoekstra, Multi-scale modelling in computational biomedicine, Briefings in bioinformatics 11 (1) (2010) 142–152. doi:10.1093/bib/bbp038.

[3] J. Southern, J. Pitt-Francis, J. Whiteley, D. Stokeley, H. Kobashi, R. Nobes, K. Yoshimasa, D. Gavaghan, Multi-scale computational modelling in biology and physiology, Progress in Biophysics and Molecular Biology (96) (2008) 60–89. doi:10.1016/j.pbiomolbio.2007.07.019.

[4] W. E, B. Engquist, X. Li, W. Ren, E. Vanden-Eijnden, The heterogeneous multiscale method: A review, Communications in Computational Physics 2 (3) (2007) 367–450.

[5] G. D. Ingram, I. T. Cameron, K. M. Hangos, Classification and analysis of integrating frameworks in multiscale modelling, Chemical engineering science 59 (2004) 2171–2187. doi:10.1016/j.ces.2004.02.010.

[6] C. W. Armstrong, R. W. Ford, G. D. Riley, Coupling integrated Earth System Model components with BFG2, Concurrency and Computation: Practice and Experience 21 (6) (2009) 767–791. doi:10.1002/cpe.1348.

[7] A. G. Hoekstra, E. Lorenz, J.-L. Falcone, B. Chopard, Toward a Complex Automata Formalism for MultiScale Modeling, International Journal for Multiscale Computational Engineering 5 (6) (2007) 491–502. doi:10.1615/IntJMultCompEng.v5.i6.60.

[8] A. G. Hoekstra, A. Caiazzo, E. Lorenz, J.-L. Falcone, Complex automata: multi-scale modeling with coupled cellular automata, in: A. G. Hoekstra, J. Kroc, P. M. A. Sloot (Eds.), Simulating Complex Systems by Cellular Automata, Springer-Verlag Berlin, Heidelberg, 2010, pp. 29–57. doi:10.1007/978-3-642-12203-3_3.

[9] J. Dada, P. Mendes, Multi-scale modelling and simulation in systems biology, Integrative Biology (3) (2011) 86–96. doi:10.1039/c0ib00075b.

[10] J. Hegewald, M. Krafczyk, J. Tölke, A. G. Hoekstra, An agent-based coupling platform for complex automata, in: ICCS 2008, LNCS 5102, Springer-Verlag Berlin Heidelberg, 2008, pp. 227–233. doi:10.1007/978-3-540-69387-1_25.

[11] B. Chopard, J.-L. Falcone, A. G. Hoekstra, J. Borgdorff, A Framework for Multiscale and Multiscience Modeling and Numerical Simulations, in: C. Calude, J. Kari, I. Petre, G. Rozenberg (Eds.), LNCS 6714, Springer-Verlag Berlin Heidelberg, 2011, pp. 2–8. doi:10.1007/978-3-642-21341-0_2.

[12] J. Borgdorff, J.-L. Falcone, E. Lorenz, B. Chopard, A. G. Hoekstra, A principled approach to distributed multiscale computing, from formalization to execution, in: Proceedings of the IEEE 7th International Conference on e-Science Workshops, IEEE Computer Society Press, Stockholm, Sweden, 2011, pp. 97–104. doi:10.1109/eScienceW.2011.9.

[13] J. Borgdorff, J.-L. Falcone, E. Lorenz, C. Bona-Casas, B. Chopard, A. G. Hoekstra, Foundations of distributed multiscale computing: Formalization, specification, and analysis, Journal of Parallel and Distributed Computing 73 (2013) 465–483. doi:10.1016/j.jpdc.2012.12.011.

[14] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. B. Damevski, W. R. Elwasif, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. Mcinnes, J. Nieplocha, B. Norris, J. Ray, T. L. Windus, S. Zhou, A Component Architecture for High-Performance Scientific Computing, International Journal of High-Performance Computing Applications 20 (2) (2006) 163–202. doi:10.1177/1094342006064488.

[15] B. A. Allan, R. Armstrong, Ccaffeine Framework: Composing and Debugging Applications Iteratively and Running them Statically. , Tech. Rep. SAND2005-1135C, Sandia National Laboratories (2005).

[16] D. Kim, J. W. Larson, K. Chiu, Toward Malleable Model Coupling, Procedia Computer Science 4 (2011) 312–321. doi:10.1016/j.procs.2011.04.033.

[17] J. W. Larson, R. L. Jacob, I. Foster, J. Guo, The model coupling toolkit, in: V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, C. J. K. Tan (Eds.), ICCS 2001, LNCS 2073, Springer-Verlag Berlin Heidelberg, 2001, pp. 185–194. doi:10.1007/3-540-45545-0_27.

[18] J.-L. Falcone, B. Chopard, A. G. Hoekstra, MML: towards a Multiscale Modeling Language, Procedia Computer Science 1 (1) (2010) 819–826. doi:10.1016/j.procs.2010.04.089.

[19] S. J. Zasada, M. Mamonski, D. Groen, J. Borgdorff, I. Saverchenko, T. Piontek, K. Kurowski, P. V. Coveney, Distributed Infrastructure for Multiscale Computing, in: 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications (DS-RT), IEEE, 2012, pp. 65–74. doi:10.1109/DS-RT.2012.17.

[20] MessagePack library, Messagepack 0.6.6, http://msgpack.org (2012).

[21] Sun Microsystems, XDR: External Data Representation standard, RFC 1014 (Jun. 1987).
URL http://www.ietf.org/rfc/rfc1014.txt

[22] D. Groen, S. Rieder, P. Grosso, C. d. Laat, S. P. Zwart, A lightweight communication library for distributed computing, Computational Science & Discovery 3 (1) (2010) 015002. doi:10.1088/1749-4699/3/1/015002.

[23] B. Bosak, J. Komasa, P. Kopta, K. Kurowski, M. Mamoński, T. Piontek, New capabilities in QosCosGrid middleware for advanced job management, advance reservation and co-allocation of computing resources–quantum chemistry application use case, Building a National Distributed e-Infrastructure–PL-Grid (2012) 40–55.

[24] The MAPPER project, http://www.mapper-project.eu/ (2010).

[25] A. Caiazzo, D. J. W. Evans, J.-L. Falcone, J. Hegewald, E. Lorenz, B. Stahl, D. Wang, J. Bernsdorf, B. Chopard, J. Gunn, D. R. Hose, M. Krafczyk, P. V. Lawford, R. H. Smallwood, D. Walker, A. G. Hoekstra, A Complex Automata approach for In-stent Restenosis: two-dimensional multiscale modeling and simulations, Journal of Computational Science 2 (1) (2011) 9–17. doi:10.1016/j.jocs.2010.09.002.

[26] M. Ben Belgacem, B. Chopard, A. Parmigiani, Coupling Method for Building a Network of Irrigation Canals on a Distributed Computing Environment, in: Cellular Automata, LNCS 7495, Springer, Berlin, Heidelberg, 2012, pp. 309–318. doi:10.1007/978-3-642-33350-7_32.

[27] D. J. W. Evans, P. V. Lawford, J. Gunn, D. Walker, D. R. Hose, R. H. Smallwood, B. Chopard, M. Krafczyk, J. Bernsdorf, A. G. Hoekstra, The application of multiscale modelling to the process of development and prevention of stenosis in a stented coronary artery, Philosophical Transactions of the Royal Society A 366 (2008) 3343–3360. doi:10.1098/rsta.2008.0081.

[28] J. Borgdorff, C. Bona-Casas, M. Mamonski, K. Kurowski, T. Piontek, B. Bosak, K. Rycerz, E. Ciepiela, T. Gubała, D. Harezlak, M. Bubak, E. Lorenz, A. G. Hoekstra, A Distributed Multiscale Computation of a Tightly Coupled Model Using the Multiscale Modeling Language, Procedia Computer Science 9 (2012) 596–605. doi:10.1016/j.procs.2012.04.064.

[29] D. Groen, J. Borgdorff, C. Bona-Casas, J. Hetherington, R. W. Nash, S. J. Zasada, I. Saverchenko, M. Mamonski, K. Kurowski, M. O. Bernabeu, A. G. Hoekstra, P. V. Coveney, Flexible composition and execution of high performance, high fidelity multiscale biomedical simulations, Interface Focus, accepted (2013) arXiv:1211.2963.