Delivery date: 31.3.2014

Authors:

Henrik Rusche
Wikki
E-mail : h.rusche@wikki.co.uk

Adriana Reyes Lua
NTNU

Heinz Preisig
NTNU
E-mail :
heinz.preisig@chemeng.ntnu.no

MoDeNa

Deliverable 5.1
Software Evaluation

WP's leader: Wikki

Principal investigators:
Henrik Rusche, **Wikki** (UK)

Non-permanent students and collaborators:
Adriana Reyes Lua, **NTNU** (N)

Project's coordinator:
Heinz Preisig, **NTNU** (N)

1. Description of deliverable

This software evaluation has been carried out in order to assess open-source software which forms a suitable basis for the software development tasks planned within MoDeNa (task 5.1).

2. Summary of contribution of involved partners

Wikki carried out the software evaluation and test installations. NTNU contributed a survey of coupling software and programming languages for the interface library as well as beening involved in the discussion of the results. The deliverable forms the basis of the will be used by used by Wikki and NTNU.

3. Introduction

MoDeNa aims at developing, demonstrating and assessing an easy-to-use multi-scale software-modelling framework application under an open-source licensing scheme that delivers models with feasible computational loads for process and product design of complex materials. The use of the software will lead to novel research and development avenues that fundamentally improve the properties of these nanomaterials.

The concept of MoDeNa is an interconnected multi-scale modelling-software framework. Four scales are linked together by this framework namely the nano-, micro-, meso-, and macroscale. This unifying software framework will allow for enhanced product- and process design across these scales. As is shown in Figure 1, the modelling framework is intimately coupled with the software framework. In this project, the software framework will facilitate and greatly enhance the modelling activities. The orchestrator enables the linking of all scales which is a necessary condition to obtain an integral approach, in contrast to a series of disconnected phases.

The orchestrator is the backbone of our software suite in that it logically links the specific task-solvers (or applications). These application-specialised codes mostly existing already and will be connected across the scales. The orchestration software is very much like a work-flow generation interface, in which the different task-solvers acting at each single scale are 'orchestrated' by the framework. The orchestrator calls the external software and obtains the necessary information, which is analysed, pre-processed and passed to the next task-solver through a suitable protocol.

This coupling will allow for the application to product and process design as well as the integration of the various computational tasks (see Figure 2) through linking of models,

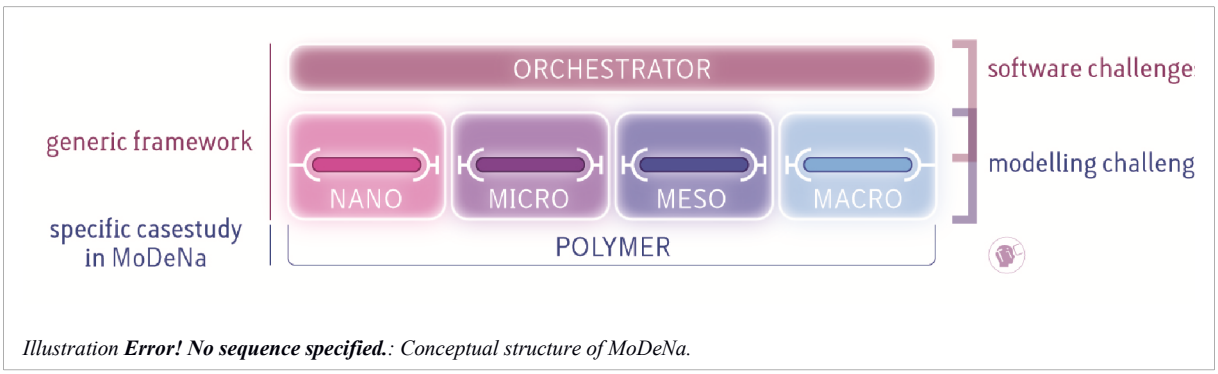

*Illustration **Error! No sequence specified.**: Conceptual structure of MoDeNa.*

associated parameters, data, and the production process of a multi-scale material can be accurately simulated.

Within the project an open-source software-suite is constructed that logically interlinks scale and problem specific software of our university groups, using a software orchestrator that communicates information utilizing our proposed new communication standard in both directions, namely upwards to the higher scale and downwards to the lower scale. This feature is unique, enabling the solution of complex material design problems.

Multi-scale coupling as proposed in MoDeNa requires the exchange of information between software instances developed for specific scales in a consistent way. In order to achieve this, generating consistent representations for models and data that is based on a solid theoretical framework is necessary. The information exchange is governed by protocols and may occur in two ways, namely:

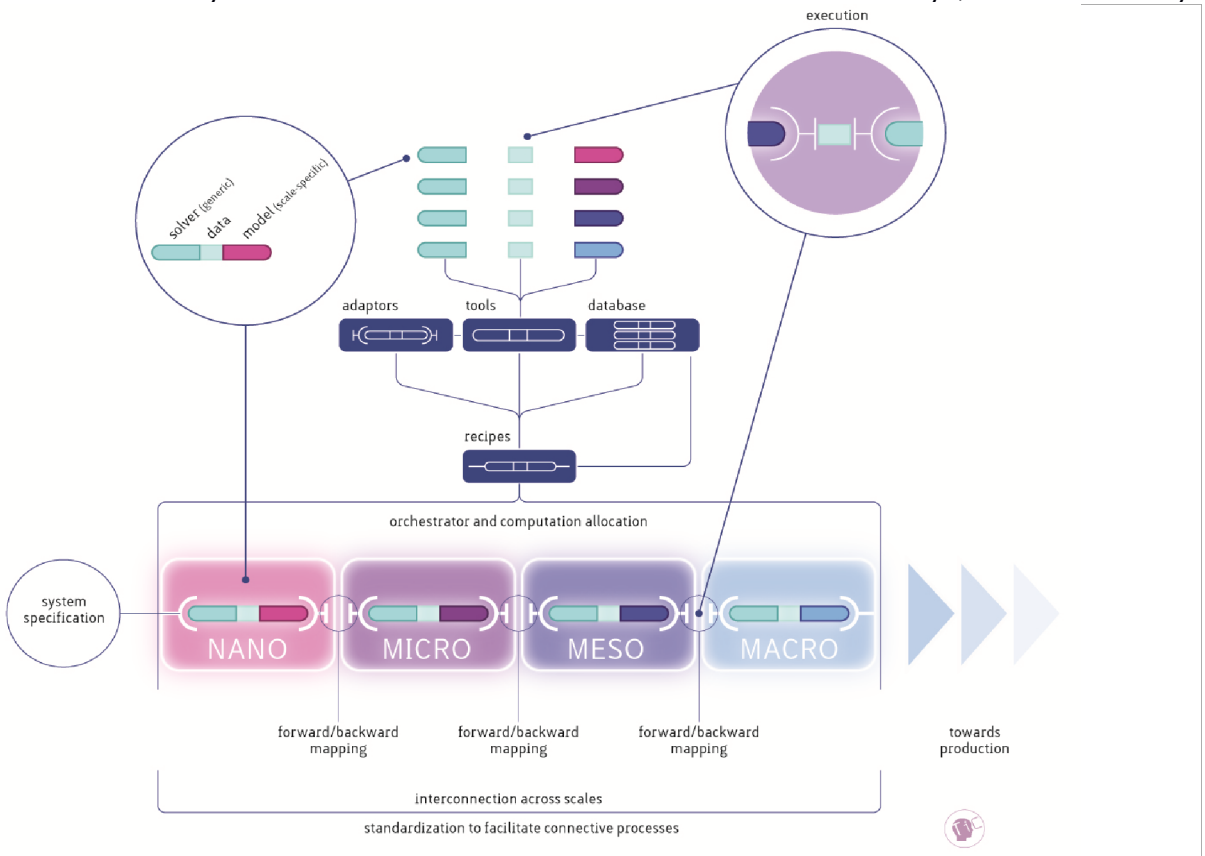

*Illustration **Error! No sequence specified.**: Conceptual structure of MoDeNa, coupling solvers and models into tools, which form sequences through recipes and the orchestrator. The sequence from nano-scale to macro-scale signifies the range of scales.*

- 'forward mapping' (passing information from the fine to the coarse scale, upward direction)
- backward mapping' (passing information from the coarse to the fine scale, downward direction)

'Forward mapping' is relatively straightforward, while *'backward mapping'* inevitably requires iteration since changing the operating conditions at the fine level changes the feedback to the coarse level. 'Backward mapping' can be realised in two ways: *'Two-way coupling'* and *'model fitting'* through a sequence of model based design of experiments and parameter estimation. The first approach usually requires exchange of large amounts of data

during runtime that may be expensive either due to the complexity of the data exchange or the computational cost associated with executing the fine scale software. In such cases, surrogate models, which are 'parameter fitted models' presents the only viable alternative.

The software framework under development combines complex features and IT infrastructure such as job scheduling, distributed computing, code coupling, model database, (model based) design of experiments and parameter estimation in a unique and unified way. It is unrealistic to develop all of these features from scratch within in the time and budget constraints. Hence the development must start from already existing software.

This report summarises the software evaluation effort undertaken within WP5. First, we will outline the different software components and the role within the software framework. Then, we will present the software that has been evaluated and conclude the report with a summary. Since an open-source implementation is sought only open-source software will be considered.

In the nect sections we present the software evaluation for each component.

4. Grid Middleware Software

Due the large number of specific task-solvers with there specific needs in terms of computational resources as well as licenses we expect that the simulations will have to be performed in a distributed manner, ie. on multiple computer resources in multiple locations and with different scheduling systems. Management and usage of such computer resources is addressed by Grid computing where the grid can be thought of as a distributed system with non-interactive workloads that involve a large number of files. What distinguishes grid computing from conventional high performance computing systems such as cluster computing is that grids tend to be more loosely coupled, heterogeneous, and geographically dispersed. Grids are often constructed with general-purpose grid middleware software libraries which provide execution, storage and information services in a secure and accountable environment (Foster et al., 2002; Foster et al., 2001, Foster and Kesselman (Eds.), 2005)

4.1. Globus® Toolkit

The Globus is software toolkit used for building grids (Globus, 2014; Foster, 2005; Foster and Kesselman, 1997). It is being developed by the Globus Alliance and many others all over the world. A growing number of projects and companies are using the Globus Toolkit to unlock the potential of grids for their cause.

The Open Grid Services Architecture (OGSA) tries to address the challenge of integrating services spread across distributed, heterogeneous, dynamic "virtual organizations" using the concepts and technologies from both the Grid and Web service communities. The Web service community has realized that Web services can reach their full potential only if there exists a mechanism to describe the various interactions between the services and dynamically compose new services out of existing ones. This is true in the case of Grid services as well.

The Grid Services Flow Language (GSFL) is an effort to examine technologies that address work-flow for Web services and leverage this technology for Grid services. This project contains both an XML schema definition for the specification of work-flow in Grid

environments and a reference implementation of the Work-flow engine for use in the Open Grid Services Infrastructure (OGSI).

4.2. UNICORE

UNICORE (UNiform Interface to COmputing REsources) is a grid computing technology for resources such as supercomputers or cluster systems and information stored in databases (UNICORE, 2014). UNICORE consists of three layers: a user, server, and target system tier. The user tier is represented by various clients. The primary clients are the UNICORE Rich Client, a graphical user interface based on the Eclipse framework, and the UNICORE command line client (UCC). The clients use SOAP Web services to communicate with the server tier. XML documents are used to transmit platform and site independent descriptions of computational and data related tasks, resource information, and work-flow specifications between client and server. The servers are accessible only via the Secure Socket Layer protocol.

UNICORE was developed in two projects funded by the German ministry for education and research (BMBF). In European-funded projects UNICORE evolved to a middleware system used at several supercomputer centers. UNICORE served as a basis in other research projects.

4.3. gLite

gLite (pronounced "gee-lite") is a middleware computer software project for grid computing used by the CERN LHC experiments and other scientific domains. It was implemented by collaborative efforts of more than 80 people in 12 different academic and industrial research centers in Europe. gLite provides a framework for building applications tapping into distributed computing and storage resources across the Internet. The gLite services were adopted by more than 250 computing centres and used by more than 15000 researchers in Europe and around the world.

The gLite middleware distribution was created by the EGEE (Enabling Grids for E-sciencE) project as the foundation of its globally distributed computing infrastructure (gLite, 2014). After the end of EGEE, the middleware components in gLite became part of the EMI distribution and are now managed as independent projects in their own right, providing software to grid infrastructures such as EGI (European Grid Infrastructure).

4.4. ARC

Advanced Resource Connector (ARC) is a grid computing middleware introduced by NorduGrid (ARC, 2014). It provides a common interface for submission of computational tasks to different distributed computing systems and thus can enable grid infrastructures of varying size and complexity. ARC includes data staging and caching functionality, developed in order to support data-intensive grid computing.

5. Software for Work-flow management

Work-flow management enables the user to run a large number of tasks (jobs) with dependencies on each other in a controlled environment. It is this component that logically

links the execution of the specific task-solvers (or applications). Ideally this software component should have the following features:

- Dynamic Work-flows: Due to reverse mapping the core work-flow management must allow loops and conditionals and therefore must be suitable for a-cyclically and cyclically coupled simulation scenarios.
- Reuse: The selected tool shall enable the computational practitioner to design and test his work-flows step-by-step interactively before integrating them into a multi-scale scenario. This approach will ease the development as the sub-work-flows are already tested in isolation.
- Portability: User created work-flows should easily be run in different environments without alteration. The same work-flow should run on a single system or across a heterogeneous set of resources without modification. Any extra steps taken by the concrete work-flow should be generated automatically.
- Provenance: The software keeps track of what has been done including the locations of data used and produced, and which software was used with what parameters. Runtime provenance of the jobs is captured and collected and collected in a database. This data can be used in debugging and for statistical purposes.
- Standalone Use: Work-flow management systems should be employed to carry out day-to-day simulation runs especially if a computational routine involves many steps. Here, the system is primarily used to automate and document the steps taken. In order to "inject" the steps that can not be automated, it is of practical importance that the system acts as an extension of the command line. Operation is similar to that of a software build system in that the computation is taken from one state to another. This requires that the work-flow management system can be executed standalone, ie. It is not relying on a server architecture.
- Data Management: involves case management, replica selection, data transfers and output registrations which are logged in data catalogs. Storage is cleaned as the work-flow is executed so that data-intensive work-flows have enough space to execute on storage-constrained resources.
- Performance: Tasks may be reordered, grouped and prioritized in order to increase the overall work-flow performance.
- Scalability: in terms of the size of the work-flow and the resources the work-flow is distributed over.
- Reliability: Jobs and data transfers are automatically retried in case of failures. Debugging tools help the user to debug the work-flow in case of non-recoverable failures.
- Error Recovery: When errors occur, the software tries to recover when possible. Recovery strategies may include: retrying tasks, retrying the entire work-flow, providing work-flow-level checkpointing, re-mapping portions of the work-flow to other resources, trying alternative data sources for staging data, providing restart capabilities that trigger only the work that remains to be done.

A review of work-flow management systems can be found in Yu and Buyya, 2005.

5.1. ecFlow

ecFlow is a work-flow package that enables users to run a large number of programs (with dependencies on each other and on time) in a controlled environment (cFlow, 2014). It provides reasonable tolerance for hardware and software failures, combined with good restart capabilities.

The user submits tasks (jobs) which are then executed and managed by ecFlow, ie. the software receives acknowledgements from tasks when they change status and when they send events, using child commands embedded in the scripts. In addition, ecflow stores the relationship between tasks (work-flow), and is able to submit tasks dependent on the status of other tasks and attributes like time.

ecFlow runs as a server receiving requests from clients. The command line interface, the graphical interface (ecFlowview), scripts and the Python API (application interface) are the clients. The server is based on C++/boost ASIO and uses TCP/IP for communication. Multiple servers can be run on the same hardware. ecFlow submits tasks (jobs) and receives acknowledgements from tasks via specific commands embedded in the scripts.

ecFlow is developed and maintained by ECMWF - The European Centre for Medium-Range Weather Forecasts and has a long standing history. ecFlow runs in a client-server configuration. Hence it is impractical to use it for day-to-day work automation which is important in the early work-flow development phase.

5.2. YACS

YACS is part of the SALOME platform (SALOME, 2014). SALOME is an open-source software that provides a generic platform for Pre- and Post-Processing for numerical simulation. It is based on an open and flexible architecture made of reusable components. It is a cross-platform solution.

YACS is a parallel and distributed programming interpreted language for coupling computational codes. It allows to build, edit and execute calculation schemes (work-flows). A calculation scheme defines a chain or a coupling of computer codes (SALOME components or calculation components). The language permits to define a schema (a graph) composed with nodes and links between nodes. Links permit to define a work-flow. The work-flow is executed by the YACS engine. Data are exchanged between nodes through typed ports. Composite nodes (Block, Loop, Switch) are used to modularise a calculation scheme and define iterative processes, parametric calculations or branches. This assembly is made by connecting input and output ports of calculation nodes. Furthermore, containers can be used to define where SALOME components will be executed (on a network or in a cluster). The CORBA middleware provides communication among distributed components, servers and clients: dynamic loading of a distributed component, execution of a component and data exchange between components.

SALOME is developed and maintained by OpenCASCADE a subsidiary of Areva group. YACS requires all computational codes to be wrapped into SALOME components. Ideally this is achieved by adding wrapper code to the application which opens the communication channels (ports). An alternative is to use the python component and call the code from within. The SALOME components are then executed within the SALOME kernel which needs to run on each computational node. Although the design is intriguing it is felt that it is quite

intrusive and adds a high level of complexity especially when it comes to distributed computing.

5.3. Pegasus

The Pegasus project (Pegasus, 2014; Deelman et al., 2005) encompasses a set of technologies that help work-flow-based applications execute in a number of different environments including desktops, campus clusters, grids, and clouds. Pegasus bridges the scientific domain and the execution environment by automatically mapping high-level work-flow descriptions onto distributed resources. It automatically locates the necessary input data and computational resources necessary for work-flow execution. Pegasus enables scientists to construct work-flows in abstract terms without worrying about the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware (Condor, Globus, or Amazon EC2). Pegasus also bridges the current cyberinfrastructure by effectively coordinating multiple distributed resources.

Pegasus has been used in a number of scientific domains including astronomy, bioinformatics, earthquake science, gravitational wave physics, ocean science, limnology, and others.

Development and maintenance of Pegasus is lead by the pegasus collaborative computing group at the computer science department of the university of southern california. It is a very mature system with many features and good interfaces to various queuing systems and grid middleware. However, it is based on DAGMan (HTCondor, 2014) to execute the concrete work-flow compiled from abstract one. This extra software layer makes it difficult to interpret and debug the actual runs. Furthermore, the work-flow must be represented by directed a-cyclic graphs which is insufficient for the work-flows to be developed in MoDeNa. In discussion with the key developers it became clear that this shortcoming can be surmounted by dynamically generating sub-work-flows – so a loop would become a recursive evocation of sub-work-flows terminated by a null work-flow adding another level of complexity.

5.4. FireWorks

FireWorks is a code for defining, managing, and executing scientific work-flows. It can be used to automate calculations over arbitrary computing resources, including those that have a queueing system.

Some features that distinguish FireWorks are dynamic work-flows, failure-detection routines, and built-in tools and execution modes for running high-throughput computations at large computing centers. Some of its features include:

- Clean and flexible Python API, a powerful command-line interface, and a *built-in* web service for monitoring work-flows
- A database back-end (MongoDB) lets you add, remove, and search the status of work-flows
- Detect failed jobs (both soft and hard failures), and rerun them as needed
- Multiple execution modes – directly on a multi-core machines or through a queue, on a single machine or multiple machines. Assign priorities and where jobs run

- Support for *dynamic* work-flows - work-flows that modify themselves or create new ones based on what happens during execution
- Automatic duplicate handling at the sub-work-flow level - skip duplicated portions between two work-flows while still running unique sections
- Built-in tasks for creating templated inputs, running scripts, and copying files to remote machines
- Remotely track the status of output files during execution
- Package many small jobs into a single large job (e.g., *automatically* run 100 serial work-flows in parallel over 100 cores)
- Support for several queueing systems such as PBS/Torque, Sun Grid Engine, SLURM, and IBM LoadLeveler

Firework is developed and maintained primarily by Anubhav Jain at Lawrence Berkeley National Lab for the Materials Project (Materials Project, 2014). It is a relatively young project with a fresh approach – tasks are expressed as python objects, worker query a central database for new tasks. This approach also facilitates its use for day-to-day work automation which naturally leads to work-flow development. Although Pegasus supports queuing systems, it lacks integration with grid middleware stacks.

5.5. Work-flow management in the Globus® Toolkit

Many work-flow management systems Askalon (Askalon, 2014), Gridbus (Gridbus work-flow, 2014), Kepler (Kepler, 2014) and Pegasus (Pegasus, 2014) integrate with the Globus Toolkit. Some of them such as Askalon (2014) and Kepler (2014) are targeted towards the orchestration of web applications and are therefore not suitable for MoDeNa where the applications are not accessible through the web while Gridbus's work-flow language lacks the generality required for MoDeNa. Globus's native workflow project GSFL (Krishnan et al., 2002) does not seem to be active since 2002.

5.6. Work-flow management in gLite

The Workload Management System (WMS) is the gLite 3 component that allows users to submit jobs, and performs all tasks required to execute them, without exposing the user to the complexity of the Grid. It uses the Job Description Language (JDL) – a high-level language based on the Classified Advertisement (ClassAd) language, used to describe jobs and aggregates of jobs with arbitrary dependency relation. Although orchestration is restricted to directed a-cyclic graphs which is insufficient for the work-flows to be developed in MoDeNa. WMS runs in a client-server configuration. Hence it is impractical to use it for day-to-day work automation which is important in the early work-flow development phase.

5.7. Work-flow management in UNICORE

The UNICORE grid middleware stack contains a work-flow system which provides advanced work-flow processing capabilities using UNICORE Grid resources. Its main components are the Work-flow Engine and the Service Orchestrator. While the Work-flow Engine provides high-level control constructs (for-each, while, if-then-else, etc), the Service

Orchestrator contains a powerful, extensible resource broker, and deals with execution of single UNICORE jobs.

Although the system shows great potential including fully dynamic work-flows as well as easy local deployment, it was not deemed to "heavy" to be employed in the initial stages of MeDeNa. Furthermore, this choice would restrict MoDeNa towards using of UNICORE as grid middleware. However, at this stage in the project it is impossible to say what computing resources will be employed for MoDeNa. Hence, the restriction to a particular grid middleware is premature.

6. Software for Design of Experiments and Parameter Estimation

The design and experiment and parameter estimation components are required for the mapping strategies to be developed in the project, namely forward and reverse mapping.

Parameter estimation is used to estimating the values of model parameters based on data derived from lower-scale simulation results. Design of experiments is used in reverse mapping to create "experiments" - Here, lower-scale simulations.

6.1. DAKOTA toolkit

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit a Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis developed at Sandia National Laboratories (DAKOTA, 2014). It provides a flexible, extensible interface between analysis codes and iterative systems analysis methods. DAKOTA contains algorithms for:

- optimization with gradient and non-gradient-based methods;
- uncertainty quantification with sampling, reliability, stochastic expansion, and epistemic methods;
- parameter estimation with non-linear least squares methods;
- and sensitivity/variance analysis with design of experiments and parameter study methods.

These capabilities may be used on their own or as components within advanced strategies such as hybrid optimization, surrogate-based optimization, mixed integer non-linear programming, or optimization under uncertainty.

6.2. GNU Octave

GNU Octave is a high-level interpreted language, primarily intended for numerical computations (GNU Octave, 2014). It provides capabilities for the numerical solution of linear and non-linear problems, and for performing other numerical experiments. It also provides extensive graphics capabilities for data visualization and manipulation. Octave is normally used through its interactive command line interface, but it can also be used to write non-interactive programs. The Octave language is quite similar to Matlab so that most programs are easily portable.

7. Database Toolkits

The database is used to store and exchange models and their respective parameters as well as the results of lower-scale simulation results together with a respective set of meta-data. During execution of the work-flow (forward mapping), the results of the computational experiments are stored in the database. When all experiments are finished, then the model parameters are updated by the parameter estimation component and finally read by the higher-scale application to instantiate the model.

This inherently constitutes a transfer of data between specific task-solvers (or applications) which may not run on the same system or even site. To this end, client/server based databases provide transport layers which will automatically solve the problem of data transfer in a heterogeneous environment. It should be noted, however, that the required communication channels (ports) may be restricted – especially in grid scenario.

Traditional, relational databases (usually SQL based) are ideally suited to store structured data while providing atomic access, scalability and security. However, they require fixed table schemas which complicates the software development as they lack adaptivity when the data changes during the development process or is heterogeneous such as the model specifications and associated parameters.

NoSQL databases offer a solution to this problem as they are document-oriented. They are often very fast and do not require fixed table schemas, avoid join operations by storing denormalized data, and are designed to scale horizontally.

There are a number of relational database implementations in the open-source world with mySQL being the most mature and most widely used. However, in order to be database agnostic, it is possible to base the client implementation of the data containers on a SQL Toolkit such as SQLAlchemy. The exact toolkit would be dependant on the programming languages used for the interface library and parameter estimation component.

The of the most popular NoSQL systems is MongoDB. MongoDB comes with APIs for C, C++, C#, Erlang, Haskell, Java, node.js, PHP, Perl, Python, Ruby and Scala. As NoSQL databases are not standardised, there are no toolkits.

8. Programming Language for the Interface Library

The interface library provides a consistent programming interface which will be used to connect the adapters to the database. Adapters are application specific and exist as outgoing and incoming adapters. Outgoing adapters are relatively straight forward in that they perform a mapping operation (such as averaging) and write the results to the database. The averaging process may have to started and performed within the application (e.g. for time averaging). However, the results can usually be submitted in a separate process after the simulation is terminated. Incoming adapters are more complicated since they are essentially code which is executed within the target application. In order to allow for a flexible model description (stored in and read from the database), the adapter code has to be embedded into the target application. This also ensures consistency between the parameter estimation and the adapter as both are using an identical model implementation. However, this also implies that the implementation of the model execution has to be computationally efficient since the model may be called very often.

8.1. C

C bindings are a classical choice for this kind of application. This approach probably poses the least risks when it comes to interoperability – a C-compiler exists on almost any platform and the object can be called from almost any computer language. Of course, name mangling problems must be avoided. However, C is not the ideal language to program the high level constructs needed to facilitate model evaluation and automatic differentiation. Although it may be possible to levitate this by inclusion of suitable libraries.

8.2. LUA

Lua is a fast, lightweight, embeddable scripting language (LUA, 2013). Some of Lua's advantages mentioned by it's developers are:

- Robust language: used in industrial applications with emphasis on embedded systems and games.
- Fast
- Portable: runs on Unix, Windows, mobile devices, embedded microprocessors, among others.
- Embeddable

This last feature allows strong integration with code written in other languages. Lua is intended to be embedded into other applications, and accordingly it provides a robust, easy-to-use API8. This works in both senses, Lua can be extended with libraries written in other languages and programs written in other languages can be extended with Lua. Languages include: C/C++, Java, C#, Smalltalk, Fortran, Ada, Erlang, Perl and Ruby.

Many of Lua applications are within games programming. However, some applications developed in Lua have a relevant use in an engineering area. At the Aerospace, Kennedy Space Center, (Florida, USA) Lua was integrated into the Windows application that provides remote control of leak detection equipment installed at the launch pad. This system monitors gas concentration levels during space shuttle launch operation (LUA, 2012).

An application in science was done at the Computational Biology Research Center, AIST (Tokyo). Lua was embedded in a program called Genetic Understanding Perspective Preview system, which helps visualizing information of sequence databases in molecular biology. Lua facilitates data processing and scripting functions for layout of the sequence map (LUA, 2012).

8.3. Julia

Julia is a programming language developed for scientific computing. It intends to be a extensible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically typed languages. Julia's syntax is intended to be familiar to users of other technical computing environments. Julia's developers claim that Julia outperforms other languages for numerical and scientific computing in terms of speed. Julia is distributed under an MIT-type license. However, various libraries used by the Julia environment include their own licenses such as the GPL, LGPL and BSD (Julia, 2013).

Despite it is not explicitly developed for coupling applications, it has the potential to be used as a coupler. One feature that makes Julia interesting for coupling is its capacity for parallelism and distributed computation. Julia handles parallel computing in an efficient way

using the resources efficiently. Parallelism oers advantages in terms of data management, code editing and sharing, execution, debugging, collaboration, analysis, data exploration, and visualization.

Another interesting feature is that Julia allows to call code written in C/C++ and in Fortran without any "glue code", code generation, or compilation. The requirement is that the code must be in a shared library. In the examples provided by the developers, this can be done in very few lines (Bezanson et al., 2012; Julia, 2013).

9. Conclusions

After a thorough review and analysis, the following approach is adopted for MoDeNa. **FireWorks** will be used as a basis of the work-flow management. It is grid middle ware and high performance computing system agnostic which is important at this stage in the project as the the computing resources will be employed are undefined, yet. MoDeNa will interface to **grid middleware** or **HPC resources** as required. Furthermore, it offers the potential to be used standalone which, we believe, is key to the early adoption of work-flows. FireWorks offers the basic functionalities and the necessary flexibility, however, some modifications will be necessary to arrive at a truely abstract work-flow description.

FireWorks uses **MongoDB** for internal storage of the work-flows. Therefore, the most practical approach is to use MongoDB for the storage of models, parameters and responses of the lower scale simulations. MongoDB comes with APIs for many programming languages.

The choice of a suitable programming language for the interface library is still somewhat open and depends primarily on the complexity of the model descriptions. It is clear, however, that it should present itself to the embedding application in form of **C-bindings**. The first version of the software for design of experiments and parameter estimation will be based on the **GNU Octave** framework which allows rapid development of mathematical algorithms. However, we may consider re-implementation of the tested algorithms in an optimisation toolbox such a Dakota at a later stage in the project. This step would allow easier integration into complex environments as well as ease wider adoption by the community.

10. References

Askalon (2014): http://www.dps.uibk.ac.at/projects/askalon/

ARC (2014): http://www.nordugrid.org/arc/

Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. (2012): Julia: A fast dynamic language for technical computing. CoRR, abs/1209.5145

HTCondor Version 8.0.6 Manual (2014): Center for High Throughput Computing, University of Wisconsin-Madison,
http://research.cs.wisc.edu/htcondor/manual/v8.0/condor-V8_0_6-Manual.pdf

Dakota (2014): http://dakota.sandia.gov/

ecFlow (2014): https://software.ecmwf.int/wiki/display/ECFLOW/Home

E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, D. S. Katz (2005): Pegasus: a Framework for Mapping Complex Scientific Work-flows onto Distributed Systems - Scientific Programming Journal, Vol 13(3), Pages 219-237.

FireWorks (2015): http://pythonhosted.org/FireWorks/

gLite (2014): http://glite.web.cern.ch/glite/

Globus (2014): http://toolkit.globus.org/toolkit/

GNU Octave (2014): https://www.gnu.org/software/octave/

S. Krishnan, P. Wagstrom, G. von Laszewski (2002): GSFL: A Workflow Framework for Grid Services, ANL Tech report

GridBus work-flow (2014): http://www.gridbus.org/

I. Foster (2005): Globus Toolkit Version 4: Software for Service-Oriented Systems. IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13

I. Foster, C. Kesselman (1997): Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115-128

I. Foster, C. Kesselman (Eds.) (2005): The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers Inc. San Francisco

I. Foster, C. Kesselman, J. Nick, S. Tuecke, The Physiology of the Grid (2002): An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22

I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid (2001): Enabling Scalable Virtual Organizations. International J. Supercomputer Applications, 15(3)

Julia (2013): Julia Documentation

Karajan (2014): http://wiki.cogkit.org/wiki/Main_Page

Kepler (2014): https://kepler-project.org/

Lua (2012): Lua Uses

Lua (2013): The Programming Language Lua. http://www.lua.org/

Materials Project (2014): http://www.materialsproject.org/

Pegasus (2014): http://pegasus.isi.edu/

SALOME (2014): http://www.salome-platform.org/

UNICORE (2014): http://www.unicore.eu/

J. Yu and R. Buyya, Taxonomy of Work-flow Management Systems for Grid Computing (2005): Journal of Grid Computing, Volume 3, Numbers 3-4, Pages: 171-200, Springer Science+Business Media B.V., New York, USA, Sept.