Delivery date:

31-12-2015

Authors:

Henrik Rusche
Heinz A Preisig

MoDeNa

Deliverable D5.6

Production Release - Orchestrator and Interface Library

Principal investigators:

Henrik Rusche
Hrvoje Jasak
Dominik Christ

Collaborators:

Sigve Karolius

Project coordinator:

Heinz A Preisig

heinz.preisig@chemeng.ntnu.no

Abstract:

The MoDeNa framework enables the implementation of dynamic workflows consisting of a list of computational tools, the nodes in the flow, and adapters tailored realising the information transfer between pairs of computational nodes. The information exchanged is of relatively simple nature meaning scalar or vectors of numbers and their documentation, like physical units, ranges, etc. The report summarises the status of the orchestrator and interface library being part of the first production release.

Contents

1 Introduction

This is a brief report on the production release of the MoDeNa software as it relates to the orchestrator and interface library as they are currently available on MoDeNa's git repository (MoDeNa consortium, 2015) commit ID 053d5b. Currently there are two simulations available namely foam simulation & foam ageing. A third one is under development, mechanical properties. These simulations utilise the software framework to provide the interactions with between the applications through surrogate models. The software framework which is facilitates the interaction is described briefly in this report.

The software framework consists of an orchestrator, a database and a interface library. The orchestrator is based on FireWorks (Jain, 2013; Jain et al., 2015) and constitutes the backbone of the software framework in that it schedules simulations as well as design of experiments & parameter estimation operations which make up the work-flow of the overall simulation. It is very much like a dynamic work-flow engine, in which the different applications are "orchestrated" to obtain information, analyse and pass it to the other operations. The design of experiments & parameter estimation operations are described in D4.3 and D5.7 (Karolius, 2015b,a).

The NoSQL database MongoDB (MongoDB, Inc., 2016) is used to store the state of the work-flow as well as the surrogate models together with associated data such as model parameters, data used for parameter estimation, and meta-data. These data structures are described in D4.2 (Karolius et al., 2015)

The interface library consists of two parts: A high-level python module providing access to the database as well as design of experiments and regression analysis capabilities by building on MongoEngine (Lawley, 2014) and R (R Core Team, 2016), respectively. The high-level python module is described in D4.2 and D4.3 (Karolius et al., 2015; Karolius, 2015b). The second part of the interface library is a low-level library providing unified access to the surrogate models.

2 Low-level interface library

The low-level interface library is written in C to ensure interoperability across platforms and target applications while providing the computationally efficient model execution required by the applications. The library is loaded as a shared library by the macroscopic-scale applications or as a native python extension by the high-level python module ensuring that all components use identical model implementations. Complex operations such as database access are referred back to the high-level python module using the Python/C API.

The following sections list the implemented data structures and functions. The usage of these low-level data structures and functions is exemplified in section 4.

2.1 Data Structures

- struct modena_function_t
- struct modena_index_set_t
- struct modena_inputs_t
- struct modena_outputs_t
- struct modena_substitute_model_t

  stores a model and mapping for substitution
- struct modena_model_t

  stores a surrogate model

2.2 Functions

- modena_function_t * modena_function_new (const char *functionId)
- modena_function_t * modena_function_new_from_model (const struct modena_model_t *self)
- modena_index_set_t * modena_function_get_index_set (const modena_function_t *self, const char *name)
- void modena_function_destroy (modena_function_t *model)
- modena_index_set_t * modena_index_set_new (const char *indexSetId)
- size_t modena_index_set_get_index (const modena_index_set_t *self, const char *name)
- const char * modena_index_set_get_name (const modena_index_set_t *self, const size_t index)
- size_t modena_index_set_iterator_start (const modena_index_set_t *self)
- size_t modena_index_set_iterator_end (const modena_index_set_t *self)
- void modena_index_set_destroy (modena_index_set_t *indexSet)
- modena_inputs_t * modena_inputs_new (const struct modena_model_t *self)
- modena_outputs_t * modena_outputs_new (const struct modena_model_t *self)
- void modena_inputs_destroy (modena_inputs_t *inputs)
- void modena_outputs_destroy (modena_outputs_t *outputs)
- INLINE_DECL void modena_inputs_set (modena_inputs_t *self, const size_t i, double x)
- INLINE_DECL double modena_inputs_get (const modena_inputs_t *self, const size_t i)
- INLINE_DECL double modena_outputs_get (const modena_outputs_t *self, const size_t i)
- modena_model_t * modena_model_new (const char *modelId)

  Function fetching a surrogate model from MongoDB.
- size_t modena_model_inputs_argPos (const modena_model_t *self, const char *name)

  Function determining position of an argument in the input vector.
- void modena_model_argPos_check (const modena_model_t *self)

  Function checking that the user has queried all input positions.
- size_t modena_model_outputs_argPos (const modena_model_t *self, const char *name)

  Function determining position of a result in the output vector.
- size_t modena_model_inputs_size (const modena_model_t *self)

  Function returning the size of the input vector.
- size_t modena_model_outputs_size (const modena_model_t *self)

  Function returning the size of the output vector.
- int modena_model_call (modena_model_t *model, modena_inputs_t *inputs, modena_outputs_t *outputs)

  Function calling the surrogate model and checking for errors.
- void modena_model_call_no_check (modena_model_t *model, modena_inputs_t *inputs, modena_outputs_t *outputs)

  Function calling the surrogate model w/o checking for errors.
- void modena_model_destroy (modena_model_t *model)

  Function deallocating the memory allocated for the surrogate model.

3 Concept of Recipes

A recipe is a application specific code that interweaves the application into the MoDeNa framework, i.e. it allows it to be used in the model based design of experiments procedure by the backward mapping tool from deliverable 4.3 (Karolius, 2015b) A recipe implements five steps :

- Run one simulation of a model.
- Generate the input file for the model.
- Execute the model.
- Analyse the output.
- Return the output to the MoDeNa framework.

A recipe is integrated into MoDeNa work-flow by deriving a customised class as follows:

- Implement the recipe as a class which inherits from "ModenaFireTask"
- Implement a method "task" (this is automatically executed by FireWorks)
- "task" should return nothing. The appropriate FWAction which pushes the point into the database is executed automatically.

The recipe itself is not considered a data container, as described in deliverable 4.2 (Karolius et al., 2015), however, it is included into the surrogate model template under the key "exactTask".

3.1 Example

The example in Listing 1 shows the implementation of a recipe for the density calculation using US's PCSAFT tool. The class is intended to be used by FireWorks, hence the parent class in line 2 is a FireTask that has been modified to fit the purpose of MoDeNa, specifically, it allows the framework to handle events due to surrogate models that fails during the execution of the detailed code. The class contains three methods `task`, `generate_inputfile` and `analyse_output` defined on lines 7, 22 and 51 respectively. FireWorks automatically executes `task`, which generates the inputfile on line 10 and executes the detailed code on line 13 before analysing the output 19. Additionally, the class also uses a decorator, `explicit_serialize`, in line 1, however this is a detail related to FireWorks and not the MoDeNa framework.

The method in line 7 is executed when FireWorks is ready to do the computational job. Note that there will be one-job per simulation, and they can be performed in parallel as they are independent of one another. The execution of computational jobs is handled by FireWorks and the user does not have to consider re-naming input files in order to avoid name-collisions as this is handled by FireWorks.

The command generating the input file is executed in line 10, whereas the template is located in the method `generate_inputfile` starting on line 22. When the inputfile has been generated the code is executed in line 13, before the post processing is performed in line 19 using the method `analyse_output` starting on line 51.

```
1 @explicit_serialize
2 class DensityExactSim(ModenaFireTask):
3     """
4     A FireTask that starts a microscopic code and updates the database.
5     """
```

```
6
7     def task(self, fw_spec):
8
9       # Generate input file
10      self.generate_inputfile()
11
12      # Execute detailed model
13      ret = os.system('../src/PCSAFT_Density')
14
15      # This enables backward mapping capabilities (not needed in this example)
16      self.handleReturnCode(ret)
17
18      # Analyse output
19      self.analyse_output()
20
21
22      def generate_inputfile(self):
23        """Method generating a input file using the Jinja2 template engine."""
24
25        Template("""
26        {#
27            Write inputs to the template, one per line.
28        #}
29        {% for k,v in s['point'].iteritems() %}
30            {{ v }}
31        {% endfor %}
32        {#
33            The number of species, one integer.
34        #}
35            {{ s['indices'].__len__() }}
36        {#
37            Write the species (lower case) one per line.
38        #}
39        {% for k,v in s['indices'].iteritems() %}
40            {{ v.lower() }}
41        {% endfor %}
42        {#
43            Set initial feed molar fractions to zero.
44        #}
45        {% for k,v in s['indices'].iteritems() %}
46            {{ 0.0 }}
47        {% endfor %}
48        """, trim_blocks=True,
49           lstrip_blocks=True).stream(s=self).dump('in.txt')
50
51      def analyse_output(self):
52        """Method analysing the output of the file."""
53        with open('out.txt', 'r') as FILE:
54          self['point']['rho'] = float(FILE.readline())
```

Listing 1: Example of a recipe that embedds a complex model computation into the MoDeNa framework.

4 Concept of a Adaptor

An adaptor is an application specific code fragment that implements/embeds a surrogate model into an application using the MoDeNa interface library. The adaptor is unique for every surrogate model in the application, but all adaptors can generally be condensed into three phases of operation:

Creation

- Generate a modena model by querying the database using the name of the surrogate model.

- Check for errors, exit by returning error-code (triggered if a surrogate model has not been instantiated).
- Allocate memory for input and output vectors.
- Fetch position for each argument to the surrogate function in the input vector.

Execution

- Set the individual positions in the input vectors.
- Call the surrogate models.
- Check for errors, exit application with error-code to framework.
- Fetch output.

Destruction

- Deallocate memory.

Note that the adaptor is not idle, i.e. it is not waiting to be fed information by the framework, but actively seeking information. In this sense it can be described like an ethernet plug and not a power supply.

The adaptor can be implemented in different languages depending on the requirements of the application and preference of the author. However, the three phases of operation and corresponding execution sequence must all be there in order for the adaptor to completely embed the surrogate into an application the MoDeNa way. Moreover, there are some clever ways of implementing the creation and destruction procedures. It is particularly important that the error check is performed after each surrogate model has been instantiated, since this is how the framework will trigger the model initialisation in cases where the model has not been stored in the database before executing the application. This procedure may involve model based design as specified in the initialisation procedure of the model. Fortran users may prefer to implement creation and destruction blocks as subroutines outside the main code, while C programmers may want to define a memory block for the creation and a function for the destruction.

4.1 Example

As mentioned earlier, an adaptor depends on the personal preferences of the author of an application. This example shows in three code-stubs how to implement the three phases in C.

Listing 2 shows the creation phase of the adaptor. The creation phase only needs to happen once, when the memory has been allocated and the surrogate model has been retrieved from the database it can be called as many times as necessary. The surrogate moden is instantiated from the database in line 2, and then the framework is checked for errors in line 12. The memory for the input and output vectors is allocated in lines 11 and 12 respectively, and the position for the argument "T" in the input vector is fetched in line 15 and then check that all positions have been queried.

```
1  // Instantiate model(s)
2  modena_model_t *model = modena_model_new("polymerViscosity");
3
4  // Check if any of the models are not instantiated
5  if(modena_error_occurred())
6  {
7    return modena_error();
8  }
9
10 // Allocate memory and fetch arg positions
```

```
11 modena_inputs_t *inputs = modena_inputs_new(model);
12 modena_outputs_t *outputs = modena_outputs_new(model);
13
14 // Get argument position
15 size_t Tpos = modena_model_inputs_argPos(model, "T");
16
17 // Check that all positions have been queried.
18 modena_model_argPos_check(model);
```

Listing 2: Code block example of the creation of a surrogate model.

Listing 3 shows the execution phase. The execution phase will commonly be situated in the bowels of a application code, typically in the main loop where it is called multiple time whereas the creation and destruction processes should be situated elsewhere and called only once. The input vector must first be set as shown in line 4 before calling the surrogate modell in line 7. After all the models have been called the framework should be checked for errors as shown in line 10, and the error code is returned in line 16. If the error check is clear the output from the surrogate model is fetched from the output vector and it can be used in a calculation.

```
1  double T = 298
2
3  // Set input vector
4  modena_inputs_set(inputs, Tpos, T);
5
6  // Call the model
7  int ret = modena_model_call(model, inputs, outputs);
8
9  // Terminate, if requested
10 if(modena_error_occurred())
11 {
12   modena_inputs_destroy(inputs);
13   modena_outputs_destroy(outputs);
14   modena_model_destroy(model);
15
16   return modena_error();
17 }
18
19 // Fetch result
20 double mdot = modena_outputs_get(outputs, 0);
```

Listing 3: A code block showing how to write the execution phase of the adaptor.

The last Listing 4 shows the destruction phase of the adaptor, which deallocates memory. The deallocation procedure frees the memory of the input and output vectors and the surrogate model object in lines 1, 2 and 3 respectively. Note that the destruction sequence is also performed in the case of an error in the execution phase.

```
1 modena_inputs_destroy(inputs);
2 modena_outputs_destroy(outputs);
3 modena_model_destroy(model);
```

Listing 4: A code block showing how to write the destruction phase of the adaptor.

5 Documentation

MoDeNa has the policy to integrate the software documentation primarily into the software itself and use an automatic documentation tool to generate the on-line documentation. In addition, where appropriate, inprocess is added to describe the implemented model, and on how to build application. A current snapshot of the documentation is available from http://henrus.github.io/MoDeNa/index.html.

References

Jain, A. (2013). Fireworks project web-site. http://pythonhosted.org/FireWorks/. 1

Jain, A., Ong, S. P., Chen, W., Medasani, B., Qu, X., Kocher, M., Brafman, M., Petretto, G., Rignanese, G.-M., Hautier, G., Gunter, D., and Persson, K. A. (2015). Fireworks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a. 1

Karolius, S. (2015a). Deliverable D5.7: Production release – Parameter estimation framework. Technical report, MoDeNa project. 1

Karolius, S. (2015b). Modena - deliverable D4.3: Backward mapping tool. Technical report, FP7 MoDeNa. 1, 3

Karolius, S., Birgen, C., Preisig, H. A., and Rusche, H. (2015). Modena - deliverable D4.2: Model and data containers. Technical report, FP7 MoDeNa. 1, 3

Lawley, R. (2014). Mongoengine project web-site. http://www.mongoengine.org/. 1

MoDeNa consortium (2015). Modena github repository. https://github.com/MoDeNa-EUProject/MoDeNa. 1

MongoDB, Inc. (2016). Mongodb project web-site. http://www.mongodb.org/. 1

R Core Team (2016). R project web-site. http://www.r-project.org/. 1