Delivery date:

31-08-2015

Authors:

Sigve Karolius
Cansu Birgen
Heinz Preisig
Henrik Rusche

MoDeNa

Deliverable D4.2

Model and data containers

Principal investigators:

Sigve Karolius
Henrik Rusche
Dominik Christ
Hrvoje Jasak
Heinz Preisig

Collaborators:

Cansu Birgen

Project coordinator:

Heinz A Preisig

heinz.preisig@chemeng.ntnu.no

Abstract:

Describes the model and data containers as currently utilised in the MoDeNa project as well as the construction of a layered ontology for the abstract represenation of mathematical models.

© 2015
MoDeNa

Contents

1 Introduction 1

2 High level framework 1

2.1 Surrogate Functions 1

2.1.1 Surogate function Framework Implementation 2

2.2 Surrogate Models 3

2.2.1 Surrogate Model, Framework Implementation 5

3 Low level framework 9

3.1 Modena Function 9

3.2 Modena Model 9

4 Data containers 10

5 Abstract model representation 16

5.1 The basic structural elements of the ontology 16

5.2 Mathematical section 17

5.3 Multiple networks 21

5.4 Conclusion 21

Nomenclature 22

1 Introduction

The report has three major sections, namely the description of the high level framework, the low level framework of the data containers and the abstract mathematical model representation.

The first part considers the ground-rules for the data containers in the MoDeNa framework, which where implicitly set when MongoDB was chosen as the tool for data storage at the top level. However, using a NoSQL database provides flexibility, and allows the design of the framework around the top level data container to be precisely tailored to the application. In fact, because a NoSQL database queries data using key-value pairs, the content of the database can be almost anything.

The top level database is the second theme of the report. It is the location where all the information is being stored, namely input-output data from simulations, as well as all necessary information in order to re-instantiate a surrogate model. In that sense the database can be described as a self-contained environment for all the individual surrogate models that are used in a simulation. However, in order to provide a full description of the data containers it is necessary to look beyond the database itself and consider both the information that is stored and how it is handled in the context of the framework.

The data containers in the MoDeNa framework are described beyond the top level database to include the abstraction from a mathematical representation to computer code; which is the basis for facilitating the interlinking of scales and the storage representation in the top-level database.

Ultimately, the goal of this report is to elaborate on how the database and the underlying framework communicates and interacts in order to facilitate the scale interactions.

2 High level framework

The structure of the top-level database and the properties of NoSQL databases will be introduced later, in section 4. For the this section, it is sufficient to note that a NoSQL database consists of collections of documents. Every collection holds a set of documents which contains key-value pairs.

The MoDeNa framework uses two collections: *surrogate function* and *surrogate model*, both corresponding to the terminology from the introduction of the chapter. The underlying high-level computer framework was based on generalising the set of key-value pairs for the surrogate function and surrogate model documents, respectively. Consequently, the relationship between the database and the computer-framework becomes symbiotic in the sense that the information in database, which is unique to every surrogate function and surrogate model, has no functionality unless it put into context by the general computer-framework.

This section will show examples of how a generic surrogate function and surrogate model is instantiated without discussing the technical aspects behind the implementation of the framework. One of the most notable subjects of the section is the idea of strategies, described in section 2.2, which enables the MoDeNa framework to perform event-handling specific to every model based on what happens during a simulation. Ultimately, however, the goal is to clarify that a data-container in the MoDeNa framework is more than an entry in a database.

2.1 Surrogate Functions

From a programming perspective, a surrogate function is an executable file compiled from C-code that is stored in the database when the function is instantiated. The function has inputs, outputs and parameters and limits to the domain in which the function is applicable, in the form of physical or

mathematical constraints.

Therefore, a document in the surrogate function collection, shown in Listings 2, specifies a block of C-code in line 2, as well as the global constraints and argument position of every input, output and parameter of the function.

```
f = CFunction(
    Ccode= '''Omitted, working example contains a C-function in this string''',
    # Global bounds for the function
    inputs={
        'input_1': { 'min': 0, 'max': 9e99, 'argPos': 0 },
    },
    outputs={
        'output_1': { 'min': -9e99, 'max': 9e99, 'argPos': 0 },
    },
    parameters={
        'param0': { 'min': 0.0, 'max': 10.0, 'argPos': 0 },
        'param1': { 'min': 0.0, 'max': 10.0, 'argPos': 1 },
    },
)
```

Listing 1: The code block shows an example of how to instantiate a surrogate function in the MoDeNa framework. The surrogate function document represents the code specific to the function as well as the global constraints, position and symbol of the inputs, outputs and parameters that are used in the code. The MoDeNa framework automatically handles compilation and linking of the C-code that is specified in line 1 and updates the database. An illustration of how a document in the surrogate function collection looks like can be seen in Figure 2.

The reason for separating the surrogate functions from the surrogate models is that the function is defined to contain information that is specific to the function itself and will not change. In contrast, the surrogate model in section 2.2 specifies a lot more information that can also be updated at run-time. An illustration showing how surrogate functions, such as the example in Listing 2, is saved in MongoDB is shown in Figure 2.

2.1.1 Surogate function Framework Implementation

MoDeNa's surrogate function implementation is a class which functions as a link between the database and the framework. The instantiation of the class creates an entry in the surrogate function collection of the database based on the keyword input arguments. This means that the class that inherits the `Surrogate Function` class in Listing 2, e.g. `CFunction` in Listing 2, has to perform the necessary pre-processing, such as compiling the function library.

The class consists of five methods that are called both from the high and low-level framework:

`__init__` (line 12): Initialisation method called at the end of the instantiation of the `CFunction` in Listing 2, which takes care of compiling the C library for the function. This method creates an entry for the input in the database.

`indexSet` (line 38): Method returning the index set used in the function.

`checkVariableName` (line 42): Method checking wether the indecies specified by the user are valid, i.e. a part of the index set.

`exceptionLoad` (line 48): Method returning an error code when the low-level framework fails to load the dynamic library for the C-function.

`load` **(line 54):** Method loading a surrogate model based on the `_id`, returns an instance of the model.

```
class SurrogateFunction(DynamicDocument):

    # Database definition
    name = StringField(primary_key=True)
    parameters = MapField(EmbeddedDocumentField(MinMaxArgPos))
    functionName = StringField(required=True)
    libraryName = StringField(required=True)
    indices = MapField(ReferenceField(IndexSet))
    meta = {'allow_inheritance': True}

    @abc.abstractmethod
    def __init__(self, *args, **kwargs):

        for k, v in kwargs['inputs'].iteritems():
            if not isinstance(v, MinMaxArgPos):
                kwargs['inputs'][k] = MinMaxArgPos(**v)

        for k, v in kwargs['outputs'].iteritems():
            if not isinstance(v, MinMaxArgPos):
                kwargs['outputs'][k] = MinMaxArgPos(**v)

        for k, v in kwargs['parameters'].iteritems():
            if not isinstance(v, MinMaxArgPos):
                kwargs['parameters'][k] = MinMaxArgPos(**v)

        DynamicDocument.__init__(self, **kwargs)

        for k in self.inputs.keys():
            self.checkVariableName(k);

        for k in self.outputs.keys():
            self.checkVariableName(k);

        for k in self.parameters.keys():
            self.checkVariableName(k);


    def indexSet(self, name):
        return self.indices[name]


    def checkVariableName(self, name):
        m = re.search('\[(.*)\]', name)
        if m and not m.group(1) in self.indices:
            raise Exception('Index %s not defined' % m.group(1))


    @classmethod
    def exceptionLoad(self, surrogateFunctionId):
        return 201


    @classmethod
    def load(self, surrogateFunctionId):
        return self.objects.get(_id=surrogateFunctionId)
```

2.2 Surrogate Models

A generic example of a document in the surrogate model collection, shown in Listing 3, and compared to the surrogate function in Listing 2 it looks much more complex. As shown in line 3 the surrogate

function is in fact a parameter in the surrogate model.

In addition to the surrogate function, which is defined in line 3, line 169 specifies the exact simulation, whose input-output data should be used in order to define the parameters of the surrogate function. This means that the underlying framework of the backward mapping model can be coded using a general syntax to call a for instance a "surrogate function", and the information from the database ensures that the general call corresponds to a specific surrogate function.

```
m = BackwardMappingModel(
    _id= 'test-surrogate-model',
    surrogateFunction= f,
    exactTask= FlowRateExactSim(),
    substituteModels= [ ],
    initialisationStrategy= Strategy.InitialPoints(
        initialPoints=
        {
            'input_1': list,
        },
    ),
    outOfBoundsStrategy= Strategy.ExtendSpaceStochasticSampling(
        nNewPoints= 4
    ),
    parameterFittingStrategy= Strategy.NonLinFitWithErrorContol(
        testDataPercentage= 0.2,
        maxError= 0.05,
        improveErrorStrategy= Strategy.StochasticSampling(
            nNewPoints= 2
        ),
    ),
)
```

Listing 3: The code block shows an example of the syntax for instantiating a "backward mapping" surrogate model in Python. The key-value pairs in the example are all required in order to instantiate the model, and it will automatically be uploaded as a document in the "surrogate model" collection of the MongoDB database. Once the model has been instantiated however, it contains more information, such as input-output from the "exact model", and consequently several key-value pairs are automatically created by the framework to accomodate the information. Figure 3 shows an illustration of how a document corresponding to a surrogate model will look like in the database.

An important feature of the surrogate model is that it introduces the concept of strategies. The idea behind strategies in the MoDeNa framework was to enable each individual surrogate model to handle events that the framework can solve, but whose trigger can not be planned; thus a surrogate model is essentially allowed to fail at any point during a simulation, triggering standardised procedures as necessary in order to address the problem. In other words, when a surrogate model fails the framework will stop the simulation and take action according to the defined strategies ensuring that the surrogate model will not fail at the same point during the next run of the simulation.

The use of strategies also contributes to reducing the total computational cost of a simulation because a surrogate model can be instantiated in a domain that is smaller than the global boundaries defined in the surrogate function in Listing 2. This ensures that fewer number of "exact simulations", specified in line 169, have to be performed in order to fit the the surrogate model to a domain that is larger than necessary. Instead, the surrogate model is instantiated to a more practical domain and the strategies for handling events for the model being out of bounds and parameter fitting in line 12 and 15 will automatically be used to expand the domain when necessary. Additionally, the database will automatically be updated with the input-output data from all the "exact simulations", which ensures that the "improved" surrogate model can be re-used at a later point in time.

2.2.1 Surrogate Model, Framework Implementation

The framework implementation of the surrogate model class, shown in in Listing 4, is designed to serve as the spine for the backward and forward mapping models; thus it is intended to inhereted by those classes and provide common functionality, such as the method `callModel` in line 143 of Listing 4, which calls the surrogate function.

There are a total of nine methods in the `Surrogate Model` class in Listing 4:

`__init__` (line 12): Method instantiating the surrogate method, appending a reference to the surrogate model instance to a list which keeps track of all instances of surrogate model.

`parseIndices` (line 15): A method that searches the surrogate functions name in order to determine wether it contains an index set, and returnd the indeces.

`outputsToModels` (line 29): Method used in order to replace output with that from substitute models.

`inputs_argPos` (line 36): Checks wether an input variable that is specified by the user exist in the surrogate function definition. Returns the position of the input argument in the surrogate function.

`outputs_argPos` (line 48): Checks wether an output variable that is specified by the user exist in the surrogate function definition. Returns the position of the input argument in the surrogate function.

`parameters_argPos` (line 60): Checks wether a parameter that is specified by the user exist in the surrogate function definition. Returns the position of the input argument in the surrogate function.

`inputs_max_argPos` (line 72): Returns the number of input variables to a surrogate function.

`calculate_maps` (line 76): Method mapping inputs and outputs from the surrogate function to the substitute models.

`minMax` (line 97): Returns two lists respectively containing the minimum and maximum values for every input variable. In other words, the method returns the design space in which the surrogate function can be used. The position of the variables in the list correspond to `inputs_argPos`.

`__getattribute__` (line 108): Method defining rules for how to accsess the objects attribute values.

`__setattribute__` (line 115): Method defining rules for how to set an objects attribute values.

`exceptionLoad` (line 123): Method called when a model has not been instantiated, returns 201 for the return handler.

`exceptionOutOfBounds` (line 133): Saves the point that triggered the out of bounds exception in the database, and returns the exception ”200” for the return handler.

`callModel` (line 143): Method for evaluating a surrogate model, i.e. executing `modena_model_t`, in the low-level framework.

`load` (line 164): A class method which locates a surrogate model in the database using the `_id` reference and returns the model.

`loadFailing` (line 169): The method the surrogate model that is out of bounds and returns the model.

`loadFromModule` (line 175): Finds the surrogate model which has not yet been instantiated, i.e. the C-code has not been compiled, and returns an instance of the model equivalent to that shown in Listings 3.

`get_instances` (line 184): Returns an itterator containing references to all instances of the `SurrogateModel` class.

```
1  class SurrogateModel(DynamicDocument):
2
3      # List of all instances (for initialisation)
4      ___refs___ = []
5
6      # Database definition
7      _id = StringField(primary_key=True)
8      surrogateFunction = ReferenceField(SurrogateFunction, required=True)
9      parameters = ListField(FloatField())
10     meta = {'allow_inheritance': True}
11
12     def __init__(self, *args, **kwargs):
13         self.___refs___.append(weakref.ref(self))
14
15     def parseIndices(self):
16         indices = {}
17         m = re.search('.*\[(.*)\]', self._id)
18         if m:
19             for exp in m.group(1).split(','):
20                 m = re.search('(.*)=(.*)', exp)
21                 if m:
22                     indices[m.group(1)] = m.group(2)
23                 else:
24                     raise Exception('Unable to parse %s' % exp)
25
26         return indices
27
28
29     def outputsToModels(self):
30         o = { k: self for k in self.outputs }
31         for m in self.substituteModels:
32             o.update(m.outputsToModels())
33         return o
34
35
36     def inputs_argPos(self, name):
37         try:
38             return existsAndHasArgPos(self.inputs, name)
39         except:
40             try:
41                 return existsAndHasArgPos(
42                     self.surrogateFunction.inputs,
43                     name )
44             except:
45                 raise Exception(name + ' not found in inputs')
46
47
48     def outputs_argPos(self, name):
49         try:
50             return existsAndHasArgPos(self.outputs, name)
51         except:
52             try:
```

```
53                  return existsAndHasArgPos(
54                      self.surrogateFunction.outputs,
55                      name )
56              except:
57                  raise Exception(name + ' not found in outputs')
58 
59 
60      def parameters_argPos(self, name):
61          try:
62              return existsAndHasArgPos(self.parameters, name)
63          except:
64              try:
65                  return existsAndHasArgPos(
66                      self.surrogateFunction.parameters,
67                      name )
68              except:
69                  raise Exception(name + ' not found in parameters')
70 
71 
72      def inputs_max_argPos(self):
73          return max(self.inputs_argPos(k) for k in self.inputs)
74 
75 
76      def calculate_maps(self, sm):
77          map_outputs = []
78          map_inputs = []
79 
80          for k in self.inputs:
81              try:
82                  map_inputs.extend(
83                      [self.inputs_argPos(k), sm.inputs_argPos(k)]
84                  )
85              except:
86                  pass
87 
88          for k, v in self.surrogateFunction.inputs.iteritems():
89              try:
90                  map_outputs.extend([sm.outputs_argPos(k), v.argPos])
91              except:
92                  pass
93 
94          return map_outputs, map_inputs
95 
96 
97      def minMax(self):
98          len = 1 + self.inputs_max_argPos()
99          minValues = [-9e99] * len
100         maxValues = [9e99] * len
101         for k, v in self.inputs.iteritems():
102             minValues[self.inputs_argPos(k)] = v.min
103             maxValues[self.inputs_argPos(k)] = v.max
104 
105         return minValues, maxValues
106 
107 
108     def __getattribute__(self, name):
109         if name.startswith( '___' ):
110             return object.__getattribute__(self, name)
111         else:
112             return super(DynamicDocument, self).__getattribute__(name)
113 
114 
115     def __setattribute__(self, name, value):
116         if name.startswith( '___' ):
117             object.__setattribute__(self, name, value)
```

```
118             else:
119                 super(DynamicDocument, self).__setattribute__(name, value)
120 
121 
122     @classmethod
123     def exceptionLoad(self, surrogateModelId):
124         collection = self._get_collection()
125         collection.update(
126             { '_id': surrogateModelId },
127             { '_id': surrogateModelId },
128             upsert=True
129         )
130         return 201
131 
132 
133     def exceptionOutOfBounds(self, oPoint):
134         oPointDict = {
135             k: oPoint[v.argPos]
136             for k, v in self.inputs.iteritems()
137         }
138         self.outsidePoint = EmbDoc(**oPointDict)
139         self.save()
140         return 200
141 
142 
143     def callModel(self, inputs):
144         # Instantiate the surrogate model
145         cModel = modena.libmodena.modena_model_t(model=self)
146 
147         in_i = list()
148         i = [0] * (1 + self.inputs_max_argPos())
149 
150         for k, v in self.inputs.iteritems(): # Set inputs
151             i[self.inputs_argPos(k)] = inputs[k]
152 
153         out = cModel.call(in_i, i) # Call the surrogate model
154 
155         outputs = {
156             k: out[v.argPos]
157             for k, v in self.surrogateFunction.outputs.iteritems()
158         };
159 
160         return outputs
161 
162 
163     @classmethod
164     def load(self, surrogateModelId):
165         return self.objects.get(_id=surrogateModelId)
166 
167 
168     @classmethod
169     def loadFailing(self):
170         return self.objects( __raw__={'outsidePoint': { '$exists': True}}
171         ).first()
172 
173 
174     @classmethod
175     def loadFromModule(self):
176         collection = self._get_collection()
177         doc = collection.find_one({ '_cls': { '$exists': False}})
178         modelId = doc['_id']
179         mod = __import__(modelId)
180         return mod.m
181 
182 
```

```
183     @classmethod
184     def get_instances(self):
185         for inst_ref in self.___refs___:
186             inst = inst_ref()
187             if inst is not None:
188                 yield inst
```

3 Low level framework

The low level framework allows the surrogate models to be used in C and Fortran, but are initiated in Python; hence it can be described as providing extension types specific to the MoDeNa framework. It is compiled and linked as a shared library, which means that it can in principle can be linked against existing projects implemented in C and Fortran, as well as being imported into high-level level scripting languages, such as Matlab.

3.1 Modena Function

The definition of the surrogate function "type", a type-declaration for the low-level C-functions, is shown in Listings 5. The declaration defines a struct, a complex data type that bundles a list of variables under a common name in memory, containing pointers to various members.

`PyObject_HEAD` is a macro which is expanded to two pointers respectively containing the reference count and a pointer to a "Type object". The rest of the structure defines three members: a pointer, `pFunction`, to a python object, a dynamic library `handle`, and `function` pointer.

```
1  typedef struct modena_function_t
2  {
3      PyObject_HEAD
4      PyObject *pFunction;
5      lt_dlhandle handle;
6      void (*function)
7      (
8          const double* p,
9          const double* in_i,
10         const double* i,
11         double *o
12     );
13
14 } modena_function_t;
```

Listing 5: The code block shows the low-level type-definition for the surrogate function.

The low-level implementation in itself deserves more space than what can be include into this report. So we constrain ourselves to the document the main issues: The important detail to note is that the implementation uses the python header file, which provides access to the internal python API; thus allowing the framework to be embedded into the Python interpreter. Moreover, the modena function type is mostly used as by the modena model in 6 in order to load a surrogate function from the database and parse the index set.

3.2 Modena Model

The code block in Listing 6 shows the type definition for a surrogate model. Similar to the modena function in section 3.1 the modena model is also a python extension. However, unlike the modena

function it is the modena model that actually calls the surrogate function, setting the input and fetching the output.

```
typedef struct modena_model_t
{
    PyObject_HEAD
    PyObject *pModel;
    size_t outputs_size;
    size_t inputs_size;
    size_t inputs_minMax_size;
    double *inputs_min;
    double *inputs_max;
    bool *argPos_used;
    size_t inherited_inputs_size;
    size_t parameters_size;
    double *parameters;
    struct modena_function_t *mf;

    void (*function)
    (
        const double* p,
        const double* in_i,
        const double* i,
        double *o
    );

    size_t substituteModels_size;
    modena_substitute_model_t *substituteModels;

} modena_model_t;
```

Listing 6: The code block shows the low-level type-definition for the surrogate function.

4 Data containers

A database can be defined as an organized collection of data which enables us to handle large quantities of information by inputting, storing, retrieving and managing them. A relational database is defined as a database in which the data is organized based on the relational model of data. The purpose of this model is to provide a declarative method for data and query specification. Relational databases mostly use structured query language (SQL) which can also be used as a term to describe a relational database.

Non-relational databases, dubbed as NoSQL (Not Only SQL), provide a mechanism for storage and retrieval of data which is modeled in a way different than in a relational database. NoSQL databases are of interest in MoDeNa project since they allow database schemas which adapt to the users needs in a seemless manner. MongoDB is a auch a NoSQL document store database written in CPP and developed in an open-source project by the company 10gen Inc. The motivation behind its development is to close the gap between the fast and highly scalable key-value-stores and feature-rich traditional relational database management systems.

Some fundamental features of MongoDB are listed below to provide a more concrete insight and detailed information about the terminology and concepts:

Document database A record in MongoDB is a document, which is a data structure composed of pairs (field and value), or (key and value). The values of fields can also include arrays of other documents. MongoDB documents are in BSON ('Binary JSON') data format. Essentially, it is

Table 1: Terminology and concepts in SQL and MongoDB.

SQL	MongoDB
Database	Database
Table	Collection
Row	Document or BSON Document
Column	Field
Index	Index
Table joins	Embedded documents and linking
Primary key (specify any unique column or column combinations as primary key)	Primary key (the primary key is automatically set to the _id field in MongoDB)
Aggregation (e.g. by group)	Aggregation pipeline

a binary form for representing objects or documents. Using documents is advantageous since in many programming languages they correspond to native data types, embedded documents (subdocuments) and arrays reduce need for expensive joins, and dynamic schema supports fluent polymorphism. In the documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. MongoDB stores all documents in collections. A collection is a group of related documents that have a set of shared common indexes. Collections are analogous to a table in relational databases.

High performance MongoDB provides high performance data persistence. In particular, support for embedded data models reduces I/O activity on database system, and indexes support faster queries and can include keys from embedded documents and arrays.

High availability To provide high availability, MongoDB's replication facility, called replica sets, provide automatic failover and data redundancy. A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

Automatic scaling MongoDB provides horizontal scalability as part of its core functionality. Automatic sharding distributes data across a cluster of machines. Replica sets can provide eventually-consistent reads for low-latency high throughput deployment.

To provide insight in MongoDB in a comparative manner with more widely used relational database (SQL) approach, terminology and concepts of these two are summarized and shown in Table 1. MongoDB implements the basic functions of data storage, initially defined for SQL: (create, read, update, delete) CRUD. In MongoDB, these are represented as query which corresponds to read operations while data modification stands for create, update and delete operations. In MongoDB, read operation is a query which targets a specific collection of documents. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a projection that specifies the fields from the matching documents to return.

MongoDB queries exhibit the following behaviour:

- All queries in MongoDB address a single collection.
- Queries can be modified to impose limits, skips, and sort orders.
- The order of documents returned by a query is not defined unless a `sort()` method is used.

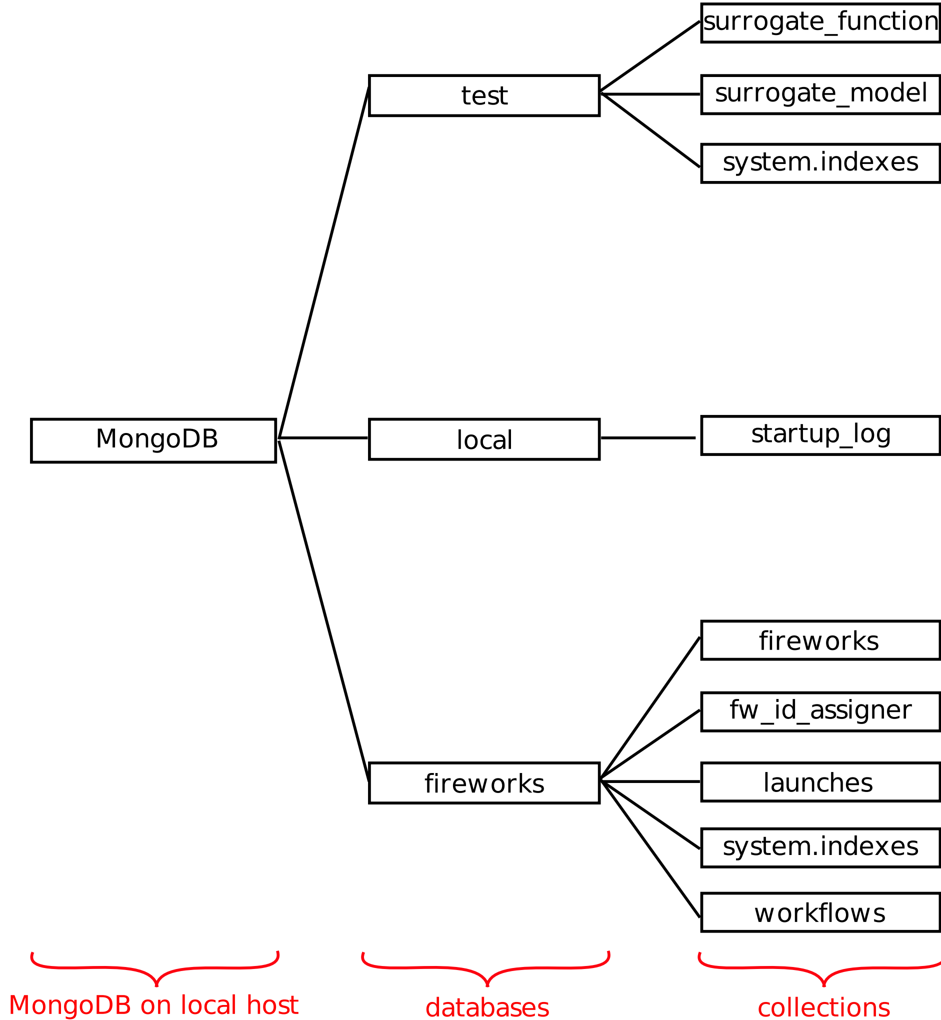

Figure 1: Illustration of database organization in MongoDB on a local host.

- Operations that modify existing documents use the same query syntax as queries to select documents to update.
- In aggregation pipeline, the $ match pipeline stage provides access to MongoDB queries.

The MoDeNa project uses MongoDB for data storage. As shown in Figure 1, the database has a crucial role as it resides in the core of the flow of information. In MongoDB, when there is data insertion, default database named as test is automatically populated unless specified otherwise. The test database in MoDeNa project contains information regarding *work-flow*, *work-flow state*, *models*, *experiments*, *metadata*, *simulation context*. Moreover, the software, Fireworks used in the project also employs MongoDB for *work-flow* (FireTasks) storage in a database called as fireworks. In addition to test and fireworks databases, there is another database (local) that is automatically created by MongoDB for the purpose of saving log information. Figure 1 shows an illustration of the MongoDB databases and corresponding collections. As can be seen in Figure 1, there are three MongoDB databases (test, local, fireworks) on local host and their corresponding collections are also shown. In MongoDB, the system generates the name by concatenating the index key field and value with an underscore, e.g. cat_1., if

the user does not specify an index name, and they are stored in a collection named as system.indexes which can be observed in the figure both for the test and fireworks databases. Each collection has at least one document which are not shown in the figure.

Since the local database is generated by MongoDB and fireworks database structure is defined by the software itself, the main focus will be the test database which is developed within the scope of the MoDeNa project. For twoTank example in the project, data is stored in test database as well which has 3 collections: surrogate_function, surrogate_model and system.indexes. Each collection stores documents which consist of key and value pairs.

The collection, surrogate_function contains one single document with the keys: id of the function _id; class of the function, _cls; the global bounds for the function, inputs which are p0, rho0, p1Byp0 and D, outputs which is flowRate and parameters which are param1 and param2 with global minimum value min, global maximum value max and argument position argPos; name of the function libraryName; compiled model in the library libraryName; C-code of the function for model execution and parameter estimation Ccode. All the mentioned keys and their value, and the hierarchy between them in terms of subdocuments constitute the database schema. The database schema is specified with corresponding keys and values in the initialization script except libraryName whose value is compiled in surrogate model script. To illustrate the data structure, data schema for surrogate_function collection is drawn with keys and it is shown in Figure 2.

As can be seen in Figure 2, surrogate_function collection has one document which has 8 keys (_id, _cls, inputs, libraryName, outputs, libraryName, parameters, Ccode). Three of the keys store documents as values e.g. subdocuments (p0, rho0, p1Byp0, D, flowRate, param1, param2), and subdocuments also store documents (min, max and argPos). Thus, there are three levels of hierarchy in the data structure. The collection, surrogate_model contains also one single document with the keys: class of the model, _cls; id of the model _id; Firework spec exactTask which has a name of _fw_name; fitting data fitData which are p0, rho0, p1Byp0 and D; inherited inputs inherited_inputs; initialization strategy initializationStrategy which has initialPoints of p0, rho0, p1Byp0 and D, and a name of _fw_name; the global bounds for the function, inputs which are p0, rho0, p1Byp0 and D with global minimum value min, global maximum value max; number of sample points nSamples; determination of points outside of the bounds outOfBoundsStrategy and corresponding number of points nNewPoints and a name of _fw_name; output of the calculation outputs which is flowRate with global minimum value min, global maximum value max; parameter fitting strategy ParameterFittingStrategy with maximum error limit of maxerror, sub-strategy for improvement improveErrorStrategy and corresponding number of points nNewPoints and a name of _fw_name, maximum number of iterations maxIterations, name of _fw_name and testDataPercentage; parameters, corresponding function of the model surrogateFunction and calculated numbers outside of bounds outsidePoint. To illustrate the data structure, data schema for surrogate_model collection is drawn with keys and it is shown in Figure 3.

As can be seen in Figure 3, surrogate_model collection has one document which has 14 keys (_id, fitData, inherited_inputs, initializationStrategy, exactTask, _cls, inputs, nSamples, surrogateFunction, outputs, outOfBoundsStrategy, parameters, ParameterFittingStrategy, outsidePoint). As described in Figure 2, keys store documents e.g. subdocuments. Thus, there are three levels of hierarchy in the data structure.

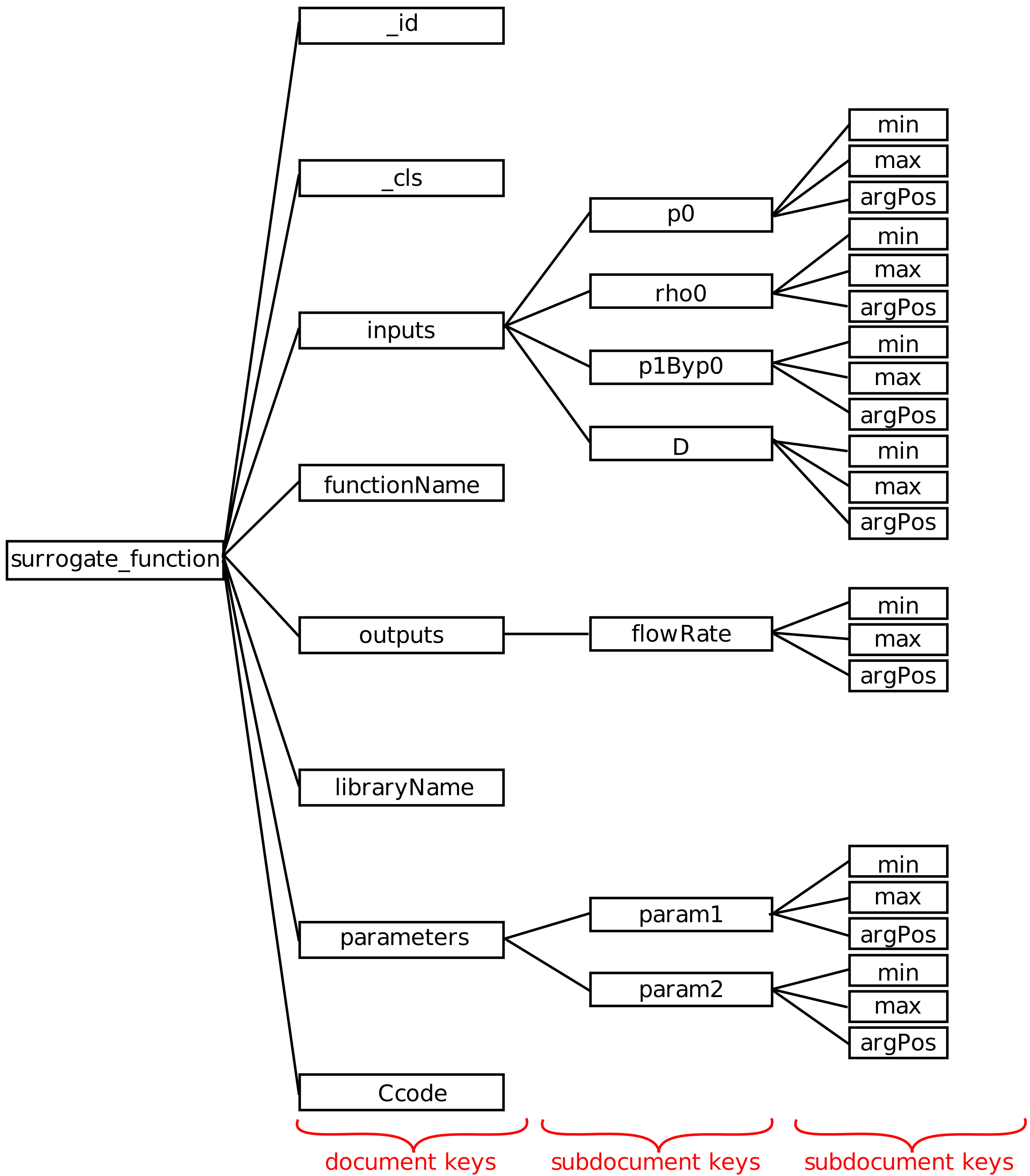

Figure 2: Data schema of `surrogate_function` collection.

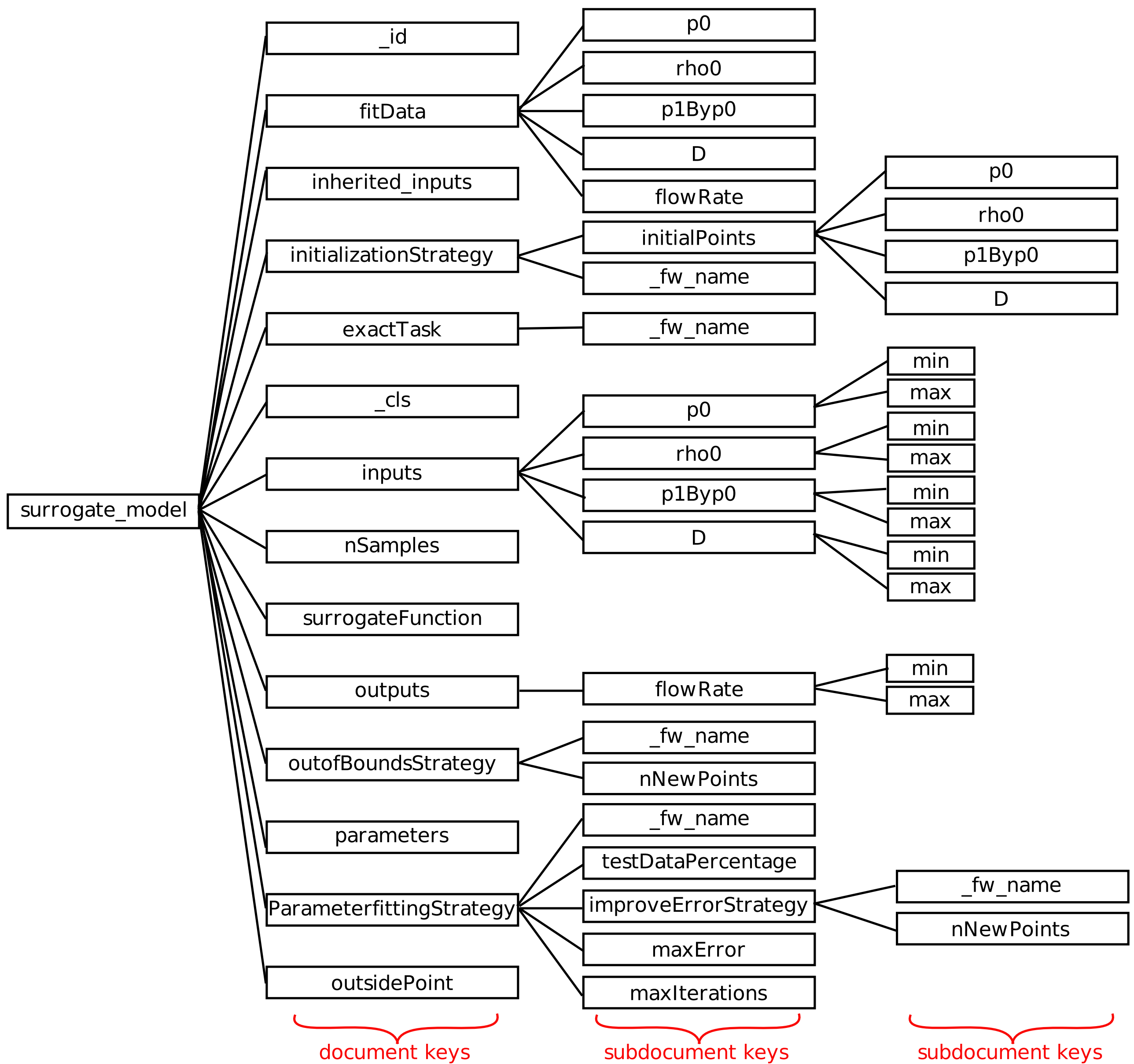

Figure 3: Data schema of `surrogate_model` collection.

5 Abstract model representation

The representation of models for the purpose of being used in coded form is a key issue in computational engineering. A simple set of rules, a simple structure that provides the complete model in all details would make it possible to transfer and inherit models in a myriad of application. This approach brings the model into the centre of computational activities and provides consistency between the different applications and utilisations of the model. The original idea that was reflected in the MoDeNa proposal and the follow up Description of Work, describes the plan of building the model systematically using a layered approach. The work done on the subject verified that indeed this approach yields the desired result, but it also showed that one had yet to take a step backwards from the position one started. The decision was taken to enter the design of a layered ontology for the representation of mathematical models that use a state space description for a network with nodes being characterised by their respective state and the arcs representing interactions between the nodes affecting the states of the two connected nodes.

This has lead to a more fundamental description, which is now able to contain mathematical models of not only physical systems but any system that satisfies those basic properties. Below we will describe the current state and the effect the development had on the implementation of model containers in MoDeNa.

5.1 The basic structural elements of the ontology

The ontologies are built on a basic core structure consisting of three parts and identified by three keywords namely structure, behaviour, typing. The first section, identified with structure contains the structural components, namely the structure of the model, in our case a graph consisting of node and arc. The two latter terms are used in singular. The fact that they are going to be lists or sets of nodes and arcs is reflected by adding a decorator. So we use a * to indicate vector/list/set properties, a . to indicate a scalar and ! to indicated the ability to transform. Later we use for representing reactions and other similarly motivated transformations.

```
1  [structure]   # building blocks
2    graph  =  ['*node', '*arc']
3    frame  =  ['.time']
4
5  [behaviour] # link to mathematical description
6    node  = ['state', 'constant']
7
8  [typing]  # specialisation of building block
9    graph  =  [ 'physical']
10   node   =  ['event','dynamic', 'constant']
11   arc    =  ['uni-directional', 'bi-directional']
```

Listing 7: The root ontology

The structure defines the graph and in the frame time. Thus we define here dynamic models. The behaviour defines that the nodes are described by state and constant. The section typing provides the refinement for the networks, in our example only one, namely physical. The nodes are placed into three subdomains in a given time scale interpretation and the arc may or may not be limited to positive flow. The arcs are defined as reference coordinates when defining the flows. Listing 7

The root ontology defines the first layer and is inherited by all the following ones. The forming of the next layer is initiated by the typing entry graph. The Listing 9 shows the next layer.

```
[structure] # extends structure with the relevant physical components
frame = ["*spatcoord"]
token = [".mass",".energy","*!species",".entropy","*information"]

[behaviour] # specialises to relevant physical variable types
node = ['secondary_state', 'transposition']
arc = ['transport']

[typing]
graph = ['liquid', 'solid','gas']  # hooks for further specialisation
node = ['0D', '1D', '2D','3D']  # qualifies nodes
```

Listing 8: The specialisation to physical networks

The structure provides the base description of a physical containment living in the frame of time and space. The token defines what goes into the containment, kind of colours. No interaction of different tokens are defined on this level. The behaviour is refined to reflect a small classes of variables here motivated by the block diagram we use for the representation of physical system (Figure 4).

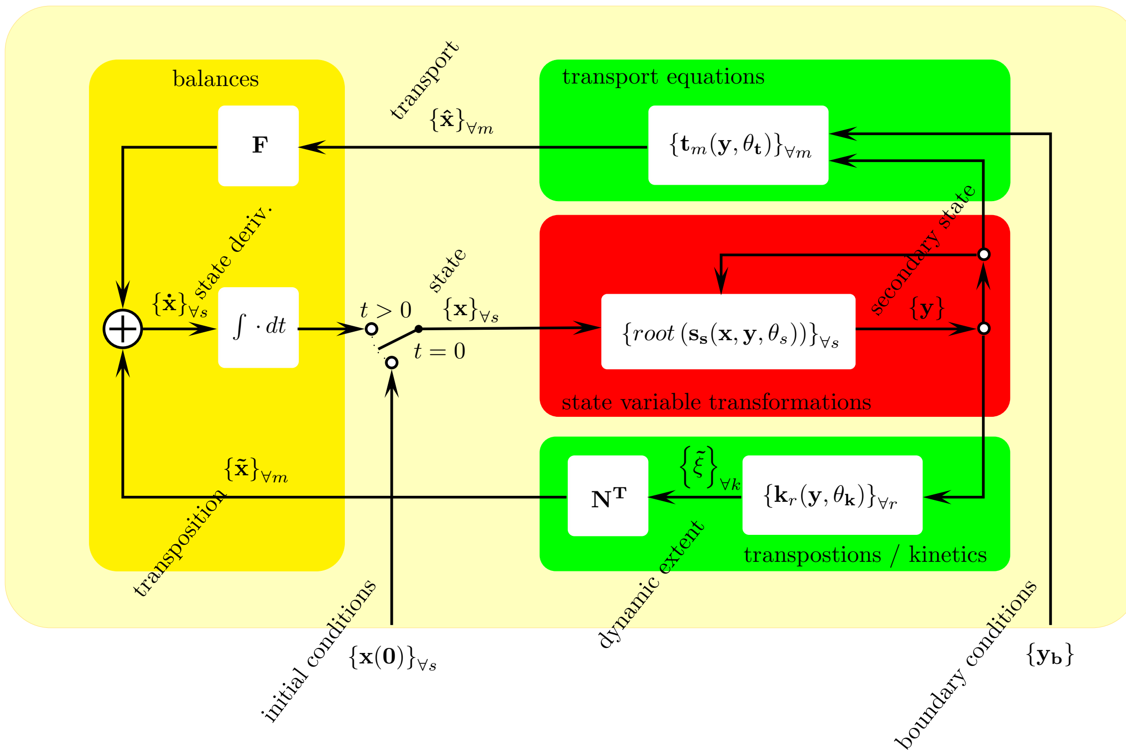

Figure 4: Block diagram of a network of physical systems with lumped elementary capacities

5.2 Mathematical section

We have established an ontology editor that builds on the above structure. Currently the root ontology and the refined ontologies are generated manually. Current focus is on generating the complex equation editor. It starts of requesting additional information for the root structural elements, specifically the state and the frame. Here we define the physical units manually. The index sets are generated automatically from the structure components and the tokens . For our case this results the index sets as shown in Listing 4.

The & represents the product operator for the set. In algebra we will write more compactly. So we have:

- network components nodes :: $\mathcal{N}$ and arcs :: $\mathcal{A}$
- frame components spatial coordinates :: $\mathcal{C}$
- tokens species :: $\mathcal{S}$
- product set node & spatial coordinates :: $\mathcal{N}\mathcal{C}$
- product set node & species :: $\mathcal{N}\mathcal{S}$
- product set arc & species :: $\mathcal{A}\mathcal{S}$
- token conversion r_species :: $\mathcal{R}$

```
[indexsets]
  0 = arc
  1 = arc & information
  2 = arc & species
  3 = information
  4 = node
  5 = node & information
  6 = node & spatcoord
  7 = node & species
  8 = r_species
  9 = spatcoord
 10 = species
```

Listing 9: All automatically generated index sets

The index sets are also automatically allocated to the states and the frame variables that themselves are generated from the defined frame and behaviour definitions.

For the definition of the equations we have defined a small language, which implements a couple of operators that are handling the block structure of the network representation. The definition thus starts in the block diagram (4) at the integrator defining the states followed by the state variable transformation, the closure equations. The required constants are defined before they are used and equipped with appropriate units and index sets, where applicable. All operators check units and index sets according to given rules and generate them for the newly defined variable as they are defined.

The language defines, besides the common operations, some special ones, as mentioned:

operator	*operation*		*result*
expand – analogue to scaler times matrix – a mapping	$a_{\mathcal{N}} \cdot b_{\mathcal{N},\mathcal{A}}$	expands space to	$c_{\mathcal{N},\mathcal{A}}$
reduce – matrix operations using Einstein indexing	$a_{\mathcal{N}} \overset{\mathcal{N}}{\star} b_{\mathcal{N},\mathcal{A}}$	reduces space to	$c_{\mathcal{A}}$
block reduce – block-wise scalar product	$a_{\mathcal{S}} \overset{\mathcal{S}}{\sharp} b_{\mathcal{N}\mathcal{S}}$	reduce block in product set index	$c_{\mathcal{N}}$
Khatri Rao – block operations	$a_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N},\mathcal{A}}{\odot} b_{\mathcal{N}\mathcal{S},\mathcal{A}\mathcal{S}}$	keeps larger blocked space	$c_{\mathcal{N}\mathcal{S},\mathcal{A}\mathcal{S}}$
	$a_{\mathcal{N}} \overset{\mathcal{N}}{\odot} b_{\mathcal{N}\mathcal{S}}$	keeps larger blocked space	$c_{\mathcal{N}\mathcal{S}}$
	$a_{\mathcal{N}} \overset{\mathcal{N},\mathcal{A}}{\odot} b_{\mathcal{N}\mathcal{S},\mathcal{A}\mathcal{S}}$	keeps larger blocked space	$c_{\mathcal{N}\mathcal{S},\mathcal{A}\mathcal{S}}$
	$a_{\mathcal{A}} \overset{\mathcal{N},\mathcal{A}}{\odot} b_{\mathcal{N}\mathcal{S},\mathcal{A}\mathcal{S}}$	keeps larger blocked space	$c_{\mathcal{N}\mathcal{S},\mathcal{A}\mathcal{S}}$

The next page shows a sample model representing a mass transfer system.

	equations	*res*	*vars*	*given*
integrals	$n_{\mathcal{NS}} := \int_0^t \dot{n}_{\mathcal{NS}}\, dt + n^o_{\mathcal{NS}}$	$n_{\mathcal{NS}}$	$\dot{n}_{\mathcal{NS}}$	$n^o_{\mathcal{NS}}$
differential balances	$\dot{n}_{\mathcal{NS}} = {F^n}_{\mathcal{NS},\mathcal{AS}} \overset{\mathcal{AS}}{\star} \hat{n}_{\mathcal{AS}}$	$\dot{n}_{\mathcal{NS}}$	$\hat{n}_{\mathcal{AS}}, {F^n}_{\mathcal{NS},\mathcal{AS}}$	
constants	${F^n}_{\mathcal{NS},\mathcal{AS}} = {F^m}_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N},\mathcal{A}}{\odot} S_{\mathcal{NS},\mathcal{AS}}$	${F^n}_{\mathcal{NS},\mathcal{AS}}$		${F^m}_{\mathcal{N},\mathcal{A}}, S_{\mathcal{NS},\mathcal{AS}}$
flows	$\hat{n}_{\mathcal{AS}} := \hat{V}_{\mathcal{A}} \overset{\mathcal{A}}{\odot} c_{\mathcal{AS}}$	$\hat{n}_{\mathcal{AS}}$	$\hat{V}_{\mathcal{A}}, c_{\mathcal{AS}}$	
	$\hat{V}_{\mathcal{A}} := -{K^V}_{\mathcal{A}} \cdot {F^m}_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N}}{\star} p_{\mathcal{N}}$	$\hat{V}_{\mathcal{A}}$	$p_{\mathcal{N}}$	${K^V}_{\mathcal{A}}, {F^m}_{\mathcal{N},\mathcal{A}}$
state variable transformations	$c_{\mathcal{AS}} := d_{\mathcal{A}} \overset{\mathcal{A}}{\odot} {F^n}_{\mathcal{NS},\mathcal{AS}} \overset{\mathcal{NS}}{\star} c_{\mathcal{NS}}$	$c_{\mathcal{AS}}$	$d_{\mathcal{A}}, {F^n}_{\mathcal{NS},\mathcal{AS}}, c_{\mathcal{NS}}$	
	$d_{\mathcal{A}} := \text{sign}\left({F^m}_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N}}{\star} p_{\mathcal{N}}\right)$	$c_{\mathcal{AS}}$	$p_{\mathcal{NS}}$	${F^m}_{\mathcal{N},\mathcal{A}}$
	$c_{\mathcal{NS}} := V_{\mathcal{N}}^{-1} \overset{\mathcal{N}}{\odot} n_{\mathcal{NS}}$	$c_{\mathcal{NS}}$	$V_{\mathcal{N}}, n_{\mathcal{NS}}$	
	$p_{\mathcal{N}} := \rho_{\mathcal{N}} \cdot g \cdot h_{\mathcal{N}}$	$p_{\mathcal{N}}$	$h_{\mathcal{N}}$	$\rho_{\mathcal{N}}, g$
	$h_{\mathcal{N}} := A_{\mathcal{N}}^{-1} \cdot V_{\mathcal{N}}$	$h_{\mathcal{N}}$	$V_{\mathcal{N}}$	$A_{\mathcal{N}}$
	$V_{\mathcal{N}} := \rho_{\mathcal{N}}^{-1} \cdot m_{\mathcal{N}}$	$V_{\mathcal{N}}$	$m_{\mathcal{N}}$	$\rho_{\mathcal{N}}$
	$m_{\mathcal{N}} := \boldsymbol{\lambda}_{\mathcal{S}} \overset{\mathcal{S}}{\sharp} n_{\mathcal{NS}}$	$m_{\mathcal{N}}$	$n_{\mathcal{NS}}$	$\boldsymbol{\lambda}_{\mathcal{S}}$

5.3 Multiple networks

The chosen structure of the ontology allows for the definition of sets of networks and connect them together. The extension to connected multiple networks is now open and allows for a multitude of combined modelling domains. The approach is to define automatically connection networks that inherit common tokens from both sides and are limited to event dynamic, with token transformation allowing for the transposition of tokens of one type into tokens of another type in the connected networks. For example a measurement will be of the represented by the token : information on the resulting information processing network side and may be temperature on the other, physical network side. Similarly we can approach different time scales in that one part of the connection network is dedicated to the lifting operation (from the macroscopic to the particle level) and homogenisation in the other direction, namely from the particle to the macroscopic/field level.

5.4 Conclusion

The decision of making a step backwards and to extent the ontology representation into multiple network representations has resulted in a clear canonical structure. At the same time the user is given quite some freedom to define her/his own details and thus the application domain has been greatly opened up. For materials modelling this implies that we have now the tools to capture not only multi-physics, but also different time scales in the same structures and combine them together.

The direct result was the introduction of indexing into the MoDeNa software, which now enables the handling of different species.

Nomenclature

$\hat{V}$ volume flow. 21

F^m mass-network matrix - the incidence matrix of the directed graph for the connected mass network. 21

F^n moalr mass-network matrix - the incidence matrix of the directed graph for the connected molar mass network. 21

S selection matrix. 21

$\hat{n}$ vector of molar mass flows. 21

$\boldsymbol{\lambda}$ vector of molecular masses. 21

$\dot{n}$ vector of accumulated molar mass. 21

c concentration vector. 21

n vector of molar mass in mol/s. 21

$\cdot$ product expanding to the minimal dimension (no duplication of dimension). 20, 21

$\odot$ Khatri Rao block product. 20, 21

ρ density. 21

$\sharp$ reduce block product product. 20, 21

$\star$ product reducing over the given dimension. 20, 21

A area. 21

$\mathcal{AS}$ product index set arcs & species. 19–21

$\mathcal{A}$ index set for arcs. 19–21

$\mathcal{C}$ index set for spatial coordinates. 19

d vector of directions for arcs. 21

g gravitation constant. 21

h height. 21

K^V diagonal matrix of conductivity for volume flow. 21

m mass in kg. 21

$\mathcal{NC}$ product index set nodes and spatial coordinates. 19

$\mathcal{NS}$ product index set nodes & species. 19–21

$\mathcal{N}$ index set for nodes. 19–21

p pressure. 21

$\mathcal{R}$ index set for reactions. 19

$\mathcal{S}$ index set for species. 19–21

V volume. 21