Contents

Preface

This User Manual is a part of the NanoSim delivery and is intended to be read by students and researchers developing software with/or based on Porto. As Porto is build on the SOFT5 framework it is essential to also become familiar with SOFT5. This manual will therefore cover some introduction to the ideas and concepts in SOFT5.

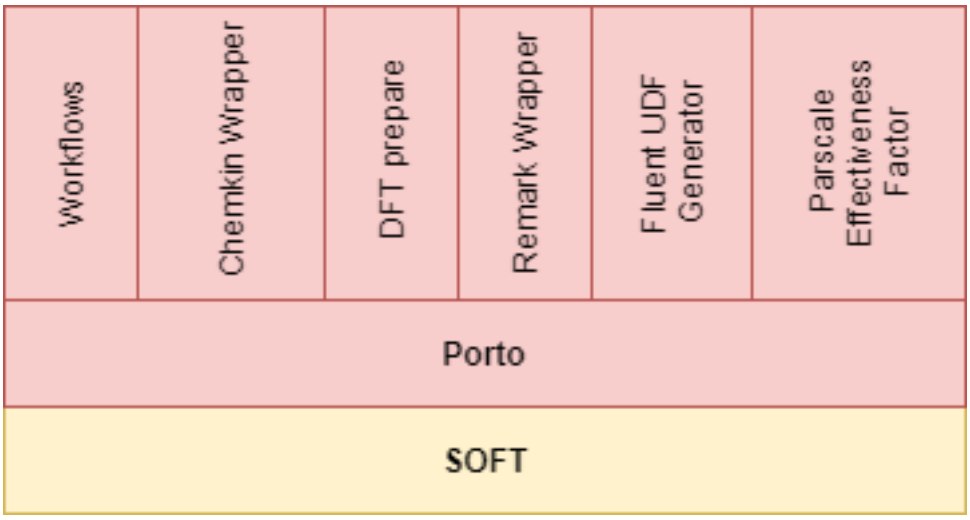

Figure 1: Porto Architecture

SOFT5 is a toolkit for building scientific software. SOFT5 allows for data to be formally described at a very low level. These descriptions are used to instruct computers how data should be interpreted or transformed. By sharing these formal descriptions, SOFT5 allows for the coupling of software by allowing one application to read data written by another, without having to care about how the data represenation such as file formats etc.

Many ideas of SOFT5 aligns with the concepts of Linked Open Data and the semantic web, however SOFT5 sacrifises the richness and dynamic flexibility of the Resource Description Framework (RDF) with a pragmatic and minimalistic approach that is suitable for representing data and "state" in scientific software, and how this relates to persistant storage (i.e. physical data storage on a harddisk in a given file format).

Porto extends SOFT5 to target different use cases in scope of the NanoSim project. The different software packages developed as part of the NanoSim project needs the ability to both publish (make available) its own data, as well as using results from other simulations.

Scope of this manual

This manual is intended for getting started with using Porto for connecting software platforms. As a fundament for developing with Porto, chapter two gives a brief introduction to the fundaments of SOFT5. In Chapter 3 Porto and NanoSim is presented. The last 2 chapters are dedicated to use cases in NanoSim where SOFT/Porto is used to handle interoperability between different simulation tools.

Porto/SOFT5

SOFT is an acronym for SINTEF Open Framework and Tools. SOFT5 is a set of libraries and tools to support scientific software development.

The development of SOFT5 was motivated by many years of experience with developing scientific software, where it was observed that a lot of efforts went into developing parts that had little to do with the domain. A significant part of the development process was spent on different software engineering tasks, such as code design, the handling of I/O, correct memory handling of the program state and writing import and export filters in order to use data from different sources. In addition comes the code maintenance with support of legacy formats and the introduction of new features and changes to internal data state in the scientific software. With SOFT5 it is possible to utilize reusable software components that handle all this, or develop new reusable software components that can be used by others in the same framework.

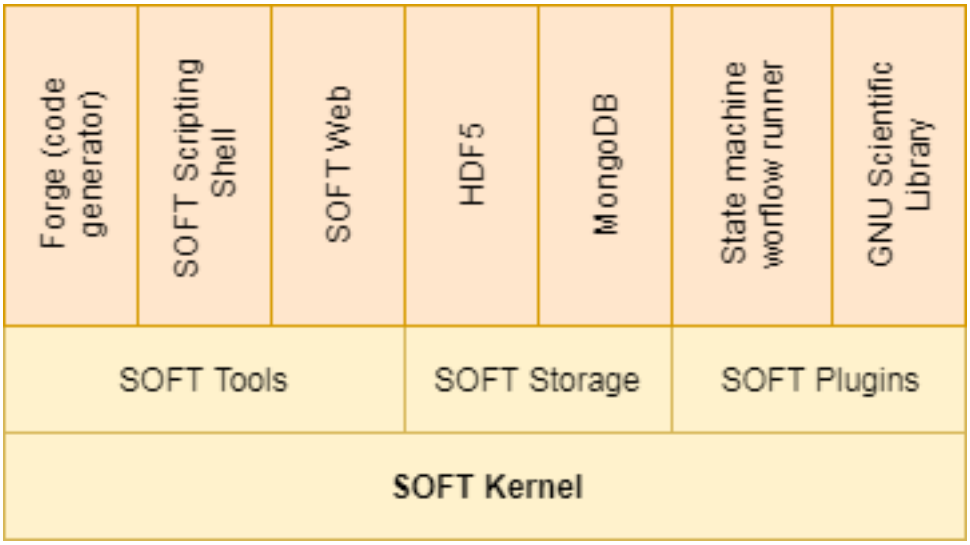

Figure 2: SOFT5 Architecture highlighting the key components also used by Porto

SOFT5 contains a core library with plugin support. The library also comes with set of interfaces (API) to create extensions and custom plugins. The core library is used to connect a software application with the framework.

There are currently two supported storage options for storing with SOFT5, namely HDF5 and MongoDB. Local data stored in HDF5 files is suitable for managing local data

The main approach to developing software with SOFT5 is to incrementally describe the domain of the software using entities (see below). The entities can represent different elements of the software, and be used in handling I/O as well as in code generation and documentation. Entities can also be used for annotating data and data sets. This might be useful in cases where for instance the origin of the data, license and ownership are of importance.

Since any complex software will have many entities and often multiple instances of the same entity, SOFT5 allows for creating collections of entities with defined relationships. These entity collections are called 'collections' (see below).

One idea of SOFT5 is that software may be written is such way that business logic is handled by the codebase, while I/O, file-formats, version handling, data import/export and interoperability can be handled by reusable components in the SOFT5-framework, thus reducing risk and development time.

The main components of SOFT5 is shown in Figure 2. The key modules are the tools, storage support and plugin framework. The key tools are the scripting utility and the code generator. The main Storage module used by Porto is MongoDB. Different plugins that extends the scripting shell includes a state machine based workflow runner.

Installation

Please see the file INSTALL.md for detailed instructions on how to configure and compile the software.

SOFT5 Features

Entities

An entity can be a single thing or object that represents something physical or nonphysical, concretely or abstract. The entity contains information about the data that constitutes the state of thing it describes. The entity does not contain the actual data, but describes what the different data fields are, in terms of name, data types, units, dimensionality etc. Information about data is often called meta data. Formal meta data enables for the correct interpretation of a set of data, which otherwise would be unreadable.

An example of an entity is 'Atom', which can be defined as something that has a position, an atomic number (which characterizes the chemical element), mass, charge, etc. Another example of a completely different kind of entity can be a data reference-entity with properties such as name, description, license, access-url, media-type, format, etc). The first entity is suitable as an object in a simulation code, while the latter is more suitable for a data catalog distribution description (see dcat:Distribution). Entities allows for describing many aspects of the domain. While each entity describes a single unit of information, a collection of entities can describe the complete domain. See collections below.

Uniqueness

Each published entity needs to be uniquely identified in order to avoid confusion. The entity identifier has therefore 3 separate elements: a name, a namespace and a version number. An entity named 'Particle' is unlikely to have the same meaning and the set of parameters across all domains. In particle physics, the entity 'Particle' would constitute matter and radiation, while in other fields the term 'Particle' can be a general term to describe something small. For this reason the SOFT5 entities have namespaces, similar to how vocabularies are defined in OWL. The version number is a pragmatic solution to handle how properties of an Entity might evolve during the development process. In order to handle different versions of a software, the entity version number can be used to identify the necessary transformation between two data sets.

Collections

A collection is defined as a set of entities and relationships between them. Collections are themselves defined as entities, and can this contain other collections as well. This is useful to represent the knowledge of the domain where data exists, in order to find data that relates to other data, but also to uniquely identify a complete data set with a single identifier.

Script Engine

SOFT5 supports natively a JavaScript (ECMA Script) engine and scripting shell. This is mainly useful for building and running workflows, code generation and for direct manipulation of entities and collections. The scripting shell can be run for the command line. In this "Hello World!" example we start the shell by typing 'softshell'. In this shell any legal javascript expression will be evaluated, in addition to the build in SOFT5 extensions.

```
user@mycomputer:~$ softshell
SOFT v5.1.5 (GNU LESSER GENERAL PUBLIC LICENSE (v 2.1, February 1999))
```

```
For help, type :help

> print ("Hello World!");
Hello World!
undefined
>
```

When we type print ("Hello World!"); the message "Hello World!" will be printed, followed by the return value from the expression (print), which in this case is 'undefined'. If we want to just examine the value of an expression, we can type it like this:

```
> 2+3
5
>
```

For further instructions on the JavaScript language, please refer to other literature.

Concepts in SOFT5

In later chapters, the usage of SOFT5 and Porto is explained using 3 use cases. In this chapter a brief introduction to using SOFT5 is presented.

Working with metadata

This section will give an example of how we might design a software tool from modeling the domain, to implementing the logic, without the regard for the syntax or file formats of the data that is the input or output. In our example, we want to calculate the average life expectancy per person living anywhere. We know statistical data exists somewhere, but we do not need to worry about the format and syntax of the data. Our main goal is express exactly how the data is to be represented in *our* system. We do this be declaring information about the data we want to use (metadata). In SOFT5, this formal metadata is described using the schema of SOFT5 Entities written in JSON notation.

We start the work by defining an entity with the fields we want to use:

```
{
    "name": "LifeExpectancy",
    "version": "1.0",
    "namespace": "http://www.sintef.no/owl/ontologies/statistics#",
    "dimensions": [
        {
            "name": "N",
            "description": "Number of samples"
        }
    ],
    "properties": [
        {
            "name": "country",
            "type": "string",
            "dims": ["N"]
        },
        {
            "name": "population",
            "type": "integer",
```

```
            "dims": ["N"],
        },
        {
            "name": "age",
            "type": "float",
            "dims": ["N"],
            "unit": "year"
        }
    ]
}
```

The first 3 fields (name, version and namespace) is uniquely identifying our Entity. The dimensions field defines labels for array sizes used in properties. The property field defines the different elements of the data structure, with data types, units, dimensionality etc. The main reason for declaring this as part of the metadata and not as part of the source code is the ability to re-use this information. One thing is that we document clearly what the data should be, but it can also instruct a file-reader/writer about how to read the different properties from a binary or text file of any format. In addition, a model of data allows for mapping from one set of metadata to another, this introducing a way to translate data from one domain to another. This is also useful for version handling.

Now we can write our algorithm:

```
var lifeExpenctancyEntity = new LifeExpenctancy(),
    totalAvgLifeExpectancyPerPopulation = 0.0,
    totalPopulation = 0;
for (var i = 0; i < lifeExpenctancyEntity.N; i++) {
    totalPopulation += lifeExpenctancyEntity.population[i];
   weightedLifeExpectency += lifeExpenctancyEntity.age[i] * lifeExpenctancyEntity.population[i];
}
var avgLifeExpectency = weightedLifeExpectency / totalPopulation;
```

'LifeExpenctancy' is a an instance of an object type derived from the metadata defined above. We know the type to contain 3 properties, namely country (not used here), population and age. We also know the properties are arrays of rank 1 with dimensionality N, where N is the number of country-population-age tuples. What is currently missing is the actual reading of the data. This will be taken care of by the framework, however some boilerplate code is needed. We need to tell which entity the LifeExpectancy is meant to represent.

```
var entity = require('porto.entity');
LifeExpectancy = entity.using('LifeExpectancy', 'http://www.sintef.no/owl/ontologies/statistics#', '1.0')
```

Now the LifeExpectancy type is defined. We also need to use a storage driver to read the actual data. While we rely on the framework to handle the IO, we do need to know which driver to actually use.

```
var storage = new soft.Storage(driver, uri, options);
...
storage.load(lifeExpenctancyEntity);
```

We will fill in the driver, uri and options arguments when we know where we get the data from. Note that this can also be input to the program itself, meaning that our code can be completely agnostic to where the data comes from.

The data source for our problem can be in any file format supported by the framework. In case it is not supported, a new plugin can be implemented.

```
Country, Population, Life Expectancy,
Monaco, 38499, 89.52
Japan,  127000000, 84.74
```

```
Singapore, 5607000000   84.68
Macau, 612167,  84.51
San Marino, 33203,  83.24
Iceland, 334252,    82.97
Hong Kong, 7347000000,  82.86
```

NanoSim/Porto Overview

Porto is build on top of SOFT5 and is the "glue" between the different simulators that needs to publish and/or exchange data. Porto uses all the major elements from SOFT5 such as the MongoDB storage module, the scripting tools, code generation framework and APIs to extend with custom data readers/writers.

Modularity and reusability

The Porto framework encourages a data driven approach to application development. Instead of encapsulation and data hiding (as is the key concept in object oriented programming), the data is public and explicit. An effect of this is that individual models or offline simulation codes that are developed can be very specialized (modular) and need only to relate to the data and not (necessarily) to other computational codes.

A key feature of Porto is to enable offline-coupling of physical processes occurring on different scales.

Key Features of the Porto platform.

The structure of the NanoSim platform is shown in Figure 3, where also the components of the two software platforms are visible. The Porto Core is a system that includes generic tools such as a code generator that allow for rapid development of interfaces, wrappers, and automatic programming of filtered models etc.

The metadata module is important, as all data that will be shared between different processes and applications needs to be well defined. The data storage module is the central database storage system and the existing tool; MongoDB (http://www.mongodb.org/), will employed for this purpose. MongoDB is a data management and data aggregation system and offers important features such as a free text query system that is fast and scalable. MongoDB supports automatic data partitioning for scalability and full index support.

A programming interface (API) for developing data format adaptors is included in the user interface system of Porto to support 3rd party data formats. When using the data interface module of Porto, the data stored in separate formats are searchable and available through the same interfaces as the data stored in the central data storage system. To further simplify the usage of these tools, the user interface of Porto includes a scripting shell that can access all the functionalities of the Porto framework. The scripting shell is the central interface of the NanoSim platform. In addition, users and developers are able to interface with specific tools on the NanoSim platform in ways they are familiar with (e.g., existing graphical user interfaces for computational fluid dynamics software, etc.).

Demonstrated Use Cases using Porto

In order to demonstrate how Porto is used, three use cases are in this document demonstrated. Connections between different scales are in practice making simulation software able to pass information to each other.

The three use cases are:

1. Coupling from atmomistic modeling (REMARC) to CFD modeling (Ansys FLUENT)
2. Coupling from atmomistic modeling (REMARC) to particle modeling (parScale)

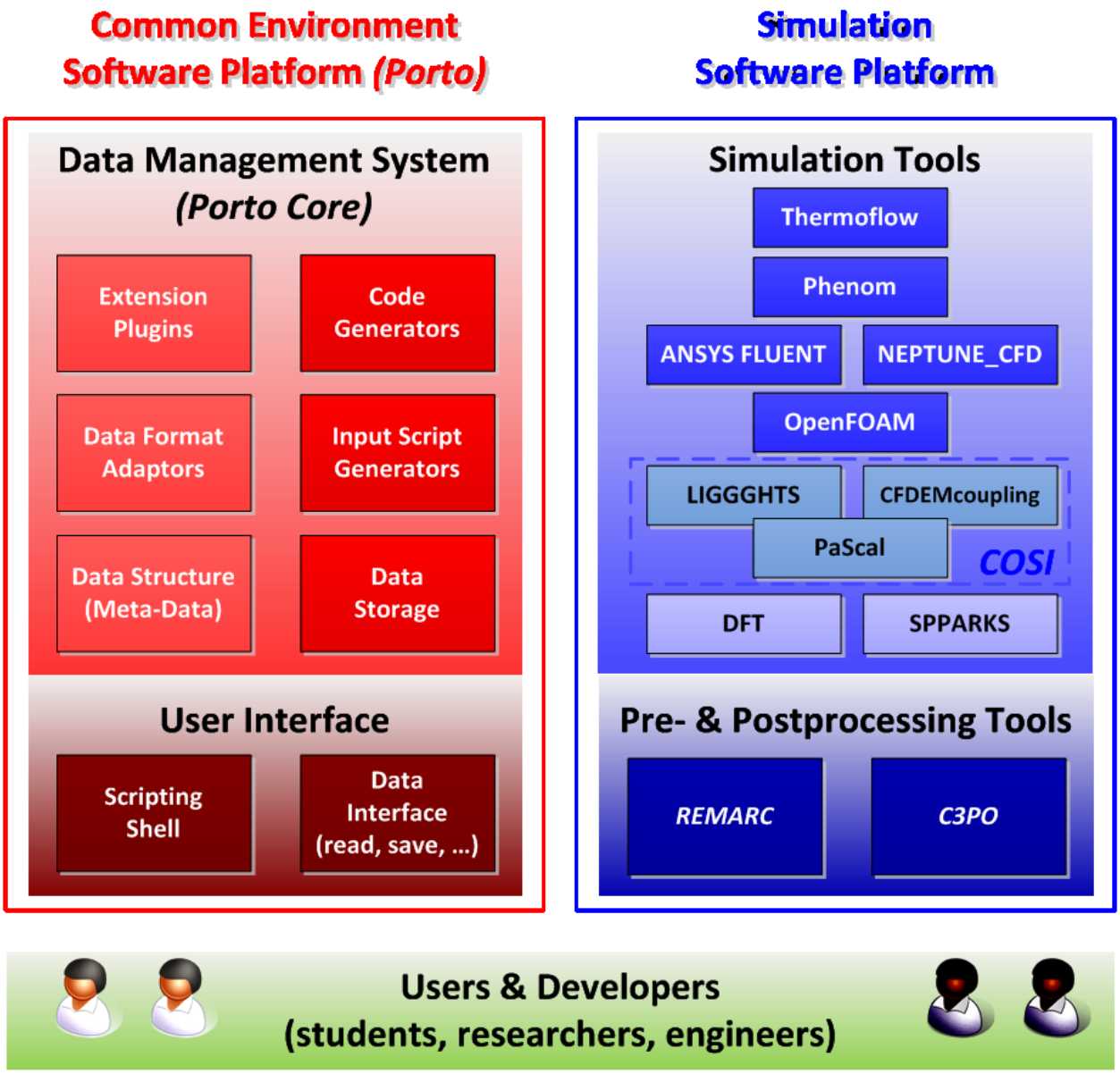

Figure 3: NanoSim Platform

3. Coupling from atmomistic modeling (REMARC) to particle modeling (parScale), then the particle modeling to CFD modeling (Ansys FLUENT)

Note: Use case 3 is not yet complete and will be completed at a later stage.

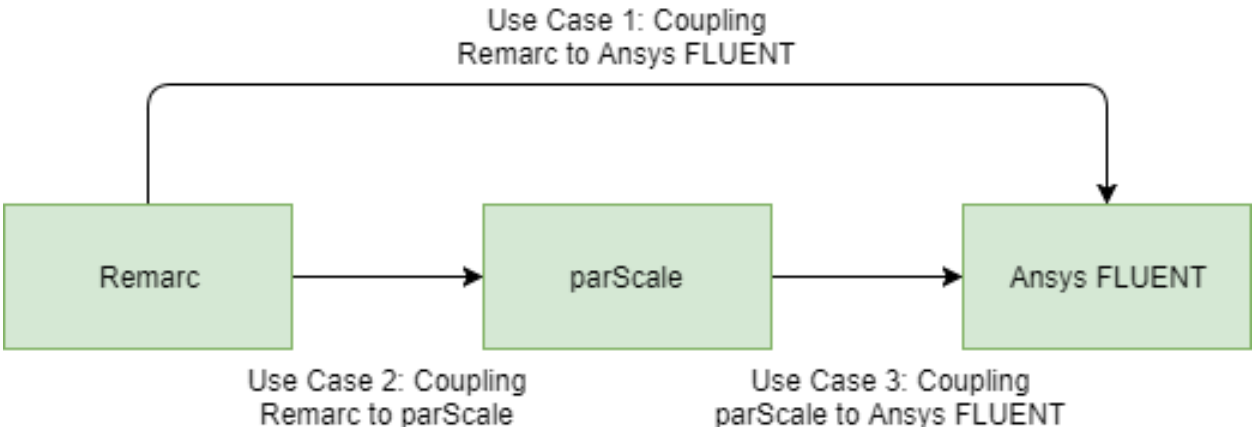

Figure 4: Use cases overview

For all practical purposes, the coupling from REMARC to parScale are identical between use case 2 and 3.

By introducing Porto as the framework for coupling, this problem is transformed. Instead of attempting to make each different software communicate with each other, each software should communicate with Porto. The data communicated in stored in a MongoDB back-end.

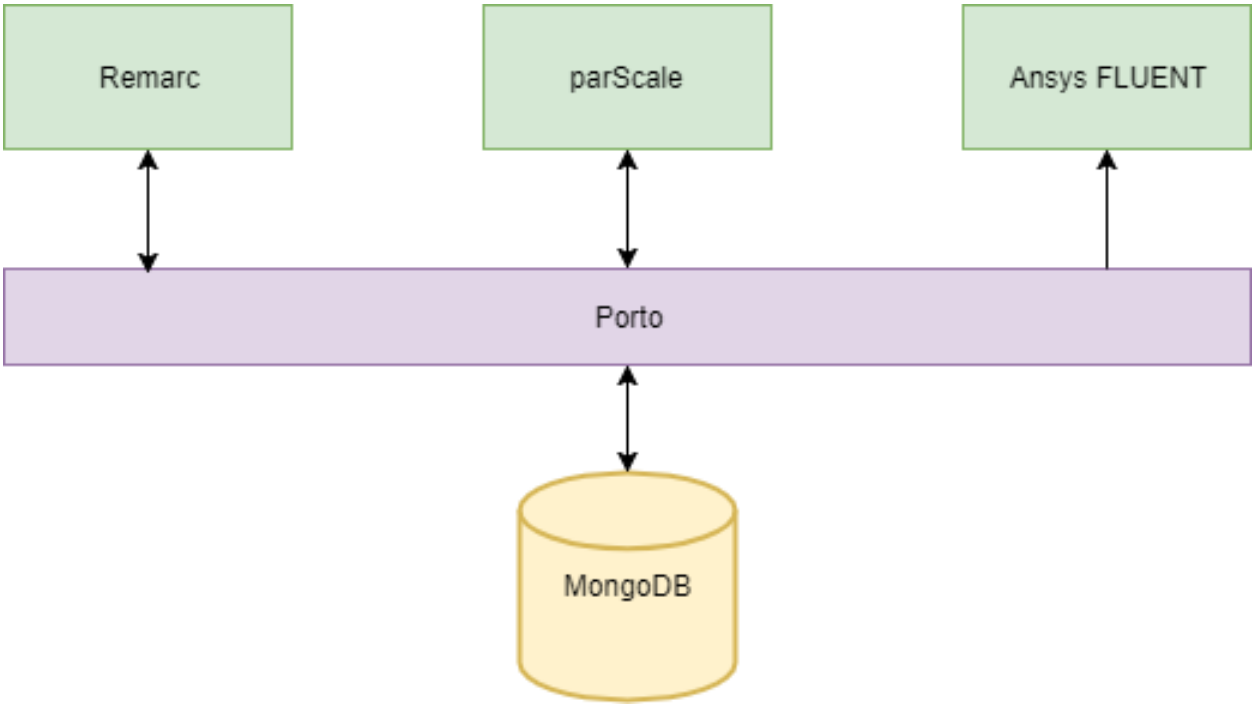

Figure 5: Use cases connected through Porto

These use cases demonstrate different techniques supported by Porto and should be regarded as tutorials on how to build coupling between any software.

Use Case 1. Coupling of REMARC and Ansys FLUENT

In this usecase we demonstrate that data from atomic scale can be used in the generation of a FLUENT UDF utilizing offline coupling with Porto. This simple demo uses some important fundamental features of Porto:

1. The ability to work with entities and metadata independent of storage mechanisms.
2. Make use of the Porto internal storage for storing both metadata and data.
3. Work with Collections for building Domain Specific Relationships.
4. Utilize custom external plugins for retrieving data from proprietary formats.
5. Seamless mixing of compiled and scripted languages (C++/JavaScript).

Prerequisite

This section contains prerequisites that needs to be resolved before running the examples.

Run a local installation of MongoDB

In the case where mongodb is installed, but not currently running it is possible to run the service from the shell:

```
sudo mongod --dbpath=/<path-to-data>/data --smallfiles
```

The `--smallfiles` option allows for mongodb to run on a computer with limited resources, as each journal file is reduced from 1GB to 128MB.

DFT data preparations

In order to run this workflow, it is assumed that there already exists DFT data coming from VASP simulations. This includes

- Gas Phase Species (large static folder of data)
- Thermo Data (needed as input for REMARC later)

We will demonstrate the use of a 'Reference' entity to simply store the location of the Gas Phase Species Data. For the accompanying thermo-properties we want to store the entire file in our database for later use.

Register entities in the database

The scripts in this UseCase utilizes the automatic runtime generation of types and objects in JavaScript. For this to work we need to register the entities that we will use. In this first Use Case we are only using the chemkinReaction. Register the entity by issuing the following command

```
soft-register-entity <soft-source-directory>/porto/src/entities/chemkinreaction.json
```

Python dependencies

Remarc uses the Python library YAML. This module can be installed with pip

```
pip install pyyaml
```

Or system-wide using apt/yum

```
sudo apt-get install python-yaml # For Debian-based systems
sudo yum install python-yaml # For RedHat-based systems
```

Create an initial collection with DFT-data

The dft-prepare is a tool written in C++ that takes the Gas Phase Species directory and a thermo-file as argument. It will first instantiate a 'Collection' and then populate this with the 'Reference' and 'File' entities. Then the collection is stored in the MongoDB backend database.

1. Create a collection
2. Create a File-entity and populate it with the contents of thermo-data
3. Create a Reference-entity and populate it with the gas phase folder info
4. Attach the File and Reference entity in the collection
5. Store the collection using an internal storage (MongoDB)
6. Return the identity (UUID) of the collection for further use.

```
$ cd porto/examples
$ dft-prepare dft/Fe2O3/ dft/thermo.dat
cc3bc435-159c-4e96-b53f-1b97a526d5ce
```

dft-prepare is a utility that can be found in the folder "porto/src/dft-prepare". It is a small C++ program that imports the auto generated C++ files "reference" and "file" created from entities with the same name. (The Porto entities resides in porto/src/entities). The classes "soft::Reference" and "soft::File" instantiated and initialized.

```
soft::Reference reference;
reference.uri          = dftPathInfo.absoluteFilePath().toStdString();
reference.created      = dftPathInfo.created().toString("dd-mm-yyyy").toStdString();
reference.owner        = dftPathInfo.owner().toStdString();
reference.lastModified = dftPathInfo.lastModified().toString("dd-mm-yyyy").toStdString();
...

soft::File file;
file.filename          = dftBoundInfo.fileName().toStdString();
file.suffix            = dftBoundInfo.suffix().toStdString();
file.size              = dftBoundInfo.size();
file.data              = dataFromFile(dftBoundInfo.absoluteFilePath());
...
```

A new soft::Collection is then instantiated and attached with reference and file.

```
soft::Collection collection;
collection.attachEntity("dftPath", &reference);
collection.attachEntity("dftBoundayFile", &file);
...
```

Run the REMARC simulation

REMARC consist of a set of Python-scripts. REMARC requires an interactive session that has been automated for this demo. We can show that this interaction can be replaced with using Porto scripting and an entity for REMARC Setup.

The REMARC-wrapper consist of three distinct parts.

1. Porto Input
2. REMARC Simulation
3. Porto Output

The wrapper assumes that there exist a Collection with correctly registered entities. (From dft-prepare). The "Porto Input" fetches the File and Reference entities from the collection.

The REMARC Simulation starts the Python scripts for a shell-process with the path to the Gas Species directory as argument. When the process has finished, a new Reference Entity is created - pointing to the SurfaceChemkin.inp file - for later reference. Additional metadata about the simulation can also be attached (version of the simulation software, ++) for later use.

The Porto Output will prepare the CHEMKIN-II data format produced by the REMARC-simulation. Here, the external storage backend 'chemkin' is used to read CHEMKIN-II data into a set of Chemkin-Entities that will be stored in the internal storage (mongodb). This data can be further used to reproduce the original CHEMKIN-II data files if needed.

```
$ # note that the uuid "cc3bc435-159c-4e96-b53f-1b97a526d5ce" will be different
$ # for each time dft-prepare is run. Replace the statement below with the the
$ # correct uuid.
$ remarc-wrapper remarc/ cc3bc435-159c-4e96-b53f-1b97a526d5ce
bin size: == 1
started
Extracting VASP data from: /home/user/nanosim-demo/dft/Fe2O3
...
bin size: == 6845
Checking the format of the chem.inp file.
chem.inp file format check PASSED.
Parsing NASA thermo file: /HOME/USER/NANOSIM-DEMO/REMARC/THERMO.DAT
End of Parsing NASA thermo file: /HOME/USER/NANOSIM-DEMO/REMARC/THERMO.DAT
Global Units are NO GLOBAL UNITS
Data output to speciesParsed and reactionsParsed.

Parsing NASA thermo file: /HOME/USER/NANOSIM-DEMO/REMARC/THERMO.DAT
End of Parsing NASA thermo file: /HOME/USER/NANOSIM-DEMO/REMARC/THERMO.DAT
```

Run the ANSYS Fluent UDF Code Generator

The code generator is a Porto-script that utilizes the 'soft.mvc' to create a text based on a template-file and a data source. In our case the template is based on an existing UDF. We will show that we can generate (a set of) UDFs from reaction data stored in a database.

The utility genudf.js is found in (porto/examples/udfgen) and uses the code generation facility of SOFT5. The entity eu.nanosim.vasp/ChemkinReaction/0.1 is used to create ChemkinReaction objects. In this example, we search the collection for a relationship:

```
var reactionDataIds = collection.findRelations("reactiondata", "has-id");
```

This will return a list of ids for a set of ChemkinReaction datapoints. We can then instantiate each one and populate it with content from the database:

```
reactionDataIds.forEach(function(reactionId){
            var reaction = new porto.ChemkinReaction(reactionId);
            reaction.read(storage);
...
```

In the SOFT5 MVC (Model-View-Controller) framework, the model constitutes an object with properties that is passed to a view-template.

```
var controller = require('soft.mvc').create({
    model: {
        min_cutoff: 1e-6,
        dp : 0.00001,
        k0 : 2.5,
```

```
        Temp: 1173.,
        EA: reaction.Ea, /* Read value from coming for the Porto database */
        Gas_Const: 8.314,
        S0Eq: "6.0 * 0.2066 * vol_frac_solid / dp",
        kEq: "k0 * exp(-EA/(Gas_Const*Temp))",
        RrateEq: "k * S0 * pow((C_R(c,gas_thread) * X_CH4 / MW_CH4 * 1000.0 ),0.6)"
    },
    view: "./template/udf.cjs"
});
```

The template is a text file (udf.cjs) which contains markups where expressions will be replaced by the evaluated value by a javascript engine. In this way we can start with a final document as we want it to be generated, and the replace all parameters with input data from the model. (The same technique is also used for generating source code).

```
#include "udf.h"

real min_cutoff = @{soft.model.min_cutoff}; /* Adjust to ensure stability */
real dp = @{soft.model.dp};                 /* Grain diameter */
real Temp = @{soft.model.Temp};             /* Adjust to experimental run */
real Gas_Const = @{soft.model.Gas_Const};
real k0 = @{soft.model.k0};                 /* Pre-exponential factor */
real EA = @{soft.model.EA};                 /* Activation energy */
```

Here we see that the activation energy (soft.model.EA) comes from the model.EA property defined earlier. This value comes from the previous calculations.

Running the genudf utility gives the following:

```
$ cd udfgen && ./genudf.js cc3bc435-159c-4e96-b53f-1b97a526d5ce
```

This produces the two UDFs

```
/* Fluent UDF using id{cc3bc435-159c-4e96-b53f-1b97a526d5ce}*/

#include "udf.h"

real min_cutoff = 0.000001;         /* Adjust to ensure stability */
real dp = 0.00001;                  /* Grain diameter */
real Temp = 1173;                   /* Adjust to experimental run */
real Gas_Const = 8.314;
real k0 = 2.5;                      /* Pre-exponential factor */
real EA = 23.999;                   /* Activation energy */

DEFINE_HET_RXN_RATE(NiO_CH4,c,t,r,mw,yi,rr,rr_t)
{
    real Rrate, k, S0, X_CH4, MW_CH4, vol_frac_solid;
    Thread *gas_thread = THREAD_SUB_THREAD(t,0);
    Thread *solid_thread = THREAD_SUB_THREAD(t,1);

    vol_frac_solid = C_VOF(c,solid_thread);
    X_CH4 = yi[0][2];    /*   Check second index if order of species in gas is altered */
    MW_CH4 = mw[0][2];   /*   Check second index if order of species in gas is altered */

    S0 = 6.0 * 0.2066 * vol_frac_solid / dp;
    k = k0 * exp(-EA/(Gas_Const*Temp));
    Rrate = k * S0 * pow((C_R(c,gas_thread) * X_CH4 / MW_CH4 * 1000.0 ),0.6);
```

```
    if (X_CH4 < min_cutoff) Rrate = 0;

    if (vol_frac_solid < min_cutoff) Rrate = 0;

    *rr = Rrate / 1000.0;
}

/* Fluent UDF using id{688c0ccd-acc4-41f8-832f-d4c8112df049}*/

#include "udf.h"

real min_cutoff = 0.000001;         /* Adjust to ensure stability */
real dp = 0.00001;                  /* Grain diameter */
real Temp = 1173;                   /* Adjust to experimental run */
real Gas_Const = 8.314;
real k0 = 2.5;                      /* Pre-exponential factor */
real EA = 0;                        /* Activation energy */

DEFINE_HET_RXN_RATE(NiO_CH4,c,t,r,mw,yi,rr,rr_t)
{
    real Rrate, k, S0, X_CH4, MW_CH4, vol_frac_solid;
    Thread *gas_thread = THREAD_SUB_THREAD(t,0);
    Thread *solid_thread = THREAD_SUB_THREAD(t,1);

    vol_frac_solid = C_VOF(c,solid_thread);
    X_CH4 = yi[0][2];    /*   Check second index if order of species in gas is altered */
    MW_CH4 = mw[0][2];   /*   Check second index if order of species in gas is altered */

    S0 = 6.0 * 0.2066 * vol_frac_solid / dp;
    k = k0 * exp(-EA/(Gas_Const*Temp));
    Rrate = k * S0 * pow((C_R(c,gas_thread) * X_CH4 / MW_CH4 * 1000.0 ),0.6);

    if (X_CH4 < min_cutoff) Rrate = 0;

    if (vol_frac_solid < min_cutoff) Rrate = 0;

    *rr = Rrate / 1000.0;
}
```

Summary

In this use case we saw how a Collection was instanciated from a C++ code and how instances of other entities was attached to it. The REMARC-wrapper used the generated collection as input ran the REMARC (Python) code as an embedded process. The generated ChemkinII files was then read and its contents written into the database and attached to the collection. In the final step we saw how JavaScript could be employed to search for registered entities within a collection and use the code generator to generate source code.

Use Case 2. Coupling of REMARC to parScale

In this example we demonstrate how to use data from an atomic scale simulation in **REMARC** as input to the particle scale model **parScale**, using **Porto**.

Walkthrough

We assume that a simulation in REMARC has already been run and the simulation results have been stored in a *Porto Collection* with a unique identifier (*uuid*). This is described in the section *Run the REMARC simulation*. The uuid allows us to access the contents of the collection across different software and simulation tools.

Before we do the actual connection, we run the tool `inspectChemkinReactionEntity.js` which takes an uuid of a collection as argument. This allows us to inspect the contents of the Collection. In this example walkthrough we do this visual inspection in order to verify that the data is in place and that we are using the correct uuid. The following output is produced:

```
$ cd porto/examples/chemkin_generator
$ ./inspectChemkinReactionEntity.js a6a71841-139a-4310-a9e6-ef7a6f161a6f
Collection (uuid = a6a71841-139a-4310-a9e6-ef7a6f161a6f)
|    Reactants => Products, A, b, Ea
+--- CH4(S1) => CH4, 615900000000000000000, -2.54, 23.999
+--- CH4 => CH4(S1), 7553000000, -0.5, 0
```

We note that, for this example, the Collection contains two reactions, including reactants and its products alongside key information about these reactions (A, b and Ea). This is the *output* of the REMARC simulation, and will be *input* to the parScale simulation.

After we have inspected this collection and are satisfied with its contents, we can run the script that does the actual connection: `chemkinReactionEntity-to-chemkinFile.js`. Running this script with the same uuid as input gives us the following output:

```
$ ./chemkinReactionEntity-to-chemkinFile.js a6a71841-139a-4310-a9e6-ef7a6f161a6f > Chemkin.inp
$ cat Chemkin.inp
ELEMENTS H C
END
SPECIES
CH4(S1) CH4
END

REACTIONS KJOULES/MOLE MOLES
CH4(S1) => CH4 615900000000000000000 -2.54 23.999
CH4 => CH4(S1) 7553000000 -0.5 0
END
```

We here see that the CHEMKIN II file has been populated with the reactions and their parameters, as inspected in the collection. This is now a file that contains the input needed to run parScale.

Details

The source for these scripts can be found in the Porto repository under `porto/examples`. These scripts are meant to serve as examples of a range of techniques for connecting different software and can easilly be tailored towards other software, file types, data sets, entities and collections.

`inspectChemkinReactionEntity.js`

The following code snippet describes how to use Porto to access and use data stored in collections. For this particular case, the reactions are printed to console.

...

```
// Access the mongoDB database
var storage = new porto.Storage("mongo2", "mongodb://localhost", "db=porto;coll=demo");

// Load the collection given by the uuid on the command line, then retrieve all
// reaction entities in this collection
var uuid = args[1];
collection = new porto.Collection(uuid);
storage.load(collection);

print("Collection (uuid = " + uuid + ")");

// Find all the reactions attached to this collection
var reactionDataIds = collection.findRelations("reactiondata", "has-id");
print("|    Reactants => Products, A, b, Ea");

reactionDataIds.forEach(function (reactionId) {
   // Read each ChemkinReaction entity in the collection
   var reaction = new porto.ChemkinReaction(reactionId);
   reaction.read(storage);
   // Print out the details
   print("+--- " + reaction["reactants"] + " => "
    + reaction["products"] + ", " + reaction["A"]
    + ", " + reaction["b"] + ", " + reaction["Ea"]);
});

...
```

chemkinReactionEntity-to-chemkinFile.js

`chemkinReactionEntity-to-chemkinFile.js` automates the following steps:

1. Access the Porto collection, referenced by the *uuid*.
2. Retrieve all reactions stored in the collection. Each reaction has the following information:

- A list of reactants
- A list of products
- Pre-exponential factor A
- Temperature exponent b
- Activation energy, Ea

3. From the list of reactants and products, generate the list of *elements* and *species* involved in the reactions.
4. Use the reaction along with the generated list of elements and species to populate a *CHEMKIN II* template.
5. Store the filled template to a file.

The way this is implemented in `chemkinReactionEntity-to-chemkinFile.js` is by first accessing the collection and retrieve its information:

```
// Attempt to talk to the local mongodb
var storage = new porto.Storage("mongo2", "mongodb://localhost", "db=porto;coll=demo");

// Load the collection given by the uuid on the command line, then retrieve all
// reaction entities in this collection
var uuid = args[1];
collection = new porto.Collection(uuid);
```

```
storage.load(collection);

var reactionDataIds = collection.findRelations("reactiondata", "has-id");
```

The list of reactants and products, species and elements are then generated:

```
// Collects all species, for example H2O, FeO3, CH4, etc.
var species = [];
// Collects all reaction lines, on the form
// <reactants> => <product> <pre.exp.factor> <b> <activation energy>
var all_reactions = []
reactionDataIds.forEach(function (reactionId) {
   var reaction = new porto.ChemkinReaction(reactionId);
   reaction.read(storage);

   var reactants = reaction.reactants;
   var products = reaction.products;

   // Collect a list of all reactants and product species
   for (var r in reactants) {
      species.push(reactants[r]);
   }
   for (var p in products) {
      species.push(products[p]);
   }

   // Construct the reactions
   if (products.length > 0 && reactants.length > 0) {
      all_reactions.push(reactants.join(" + ") + " => "
          + products.join(" + ") + " " + reaction.A + " "
          + reaction.b + " " + reaction.Ea);
   }
});

// All elements we want to check for, ideally this could be the entire periodic table
var all_elements = ["Fe", "H", "C", "O"];
var elements = [];

unique(species).forEach(function(s) {
   all_elements.forEach(function(e) {
      // For each unique species, loop over each element, then ...
      var ss = s;
      // ... check if the element is in the species.
      if (find(e, ss)) {
         // We found an element, push it to the total list of elements and
         // strip it away from the string we are searching in.
         elements.push(e);
         ss = ss.replace(e, "");
      }
   });
});
```

Using this information, the CHEMKIN II file template can be populated:

```
var controller = require('soft.mvc').create({
   model: {
```

```
      elements: unique(elements).join(" "),
      species: unique(species).join(" "),
      reactions: all_reactions.join("\n")
   },
   view: "./template/chemkin.cjs"
});
```

This template used here, `chemkin.cjs`, contains tags that corresponds to the names used in the above script which will substitute them for the actual values. For example, `@{soft.model.elements}` will here be replaced by a list of elements created above:

```
ELEMENTS @{soft.model.elements}
END
SPECIES
@{soft.model.species}
END

REACTIONS KJOULES/MOLE MOLES
@{soft.model.reactions}
END
```

CHEMKIN II file format

For this scenario, we use a CHEMKIN II file with following simple structure:

```
ELEMENTS C H FE O
END
SPECIES
CH4 H2 CO2 H2O FE2O3 FE3O4
END

REACTIONS KJOULES/MOLE MOLES
   CH4 + 3FE2O3 => CO + 2H2 + 2FE3O4    1.0e+14 0.0 100.0
   CH4 + 12FE2O3 => CO2 + 2H2O + 8FE3O4 1.0e+18 0.0 180.0
END
```

The input file consists of three sections, each starting with a name and ending with the keyword `END`.

- `ELEMENTS` contains a list of elements noted by their periodic table names
- `SPECIES` contains a list of all species used in the reaction equations
- `REACTIONS` lists the possible reactions, followed by the pre exponential factor A, temperature exponent b and activation energy Ea.

Use Case 3. Coupling of parScale to Ansys FLUENT

This use case demonstrates that effectiveness factors and effective reaction parameters calculated by the particle scale model **parScale** can be used in **Ansys FLUENT** by generating a UDF using **Porto**.

Walkthrough

This walkthrough is a continuation from the previous Use Case and assumes that it has already been run, followed by running parScale. ParScale will in this case have produced the following output:

- An `effectivenessFactor.json` file containing a description of the functional form of the effectiveness factor.
- An `effectiveReactionparameters.json` file containing a description of the functional form of the effective reaction parameters.

By "functional form" we here mean that each of these properties are described by the choice of a function and arguments to this function. For example, one simple functional form with one argument could be a *constant effectiveness factor*, written in C code as

```
double constantEffectivenessFactor(double c) {
  return c;
}
```

Since the choice of functional form that best represents the different parameters (effectiveness factor and effective reaction parameters) are expressed by parScale, we can use this information to generate an Ansys FLUENT UDF which takes this into account. Generation of the UDF is similar to the UDF generation done in Use Case 1.

To simplify this use case, we are from now on considering only the case of the constant effectiveness factor.

The output from ParScale in the form of `json` files can be read and stored in Porto. To do this we need to define the entities describing this data and registering these entities with Porto:

```
$ soft-register-entity ~/source/soft5/porto/src/entities/effectivenessfactor.json
$ soft-register-entity ~/source/soft5/porto/src/entities/effectivereactionparameters.json
```

Note that these two `json` files describe the structure of the entities and are not the same `json` files that ParScale produces.

In order to store the output from ParScale to Porto, we run the following tool:

```
$ cd ~/build/soft5/porto/src/parscale-effectiveness-factor
$ ./parscale-effectiveness-factor ~/source/soft5/porto/examples/parscale/effectivenessFactor_example.js
```

This will read the effectiveness factors and effective reaction parameters and store them in Porto under the collection referred to by the uuid. Note that the uuid used here is the same as in Use Case 2.

Following this, the UDFs can be generated by running

```
./generate-effectiveness-udf.js a6a71841-139a-4310-a9e6-ef7a6f161a6f > udf.c
```

which will generate the C code defining the UDF. This can then be compiled and used with Ansys FLUENT.

Details

`effectivenessFactor.json`

The `.json` file generated by parScale has the following example structure:

```
{
  "name": "effectivenessFactor",
  "nparameters": 1,
  "functionalForm": "constant",
  "parameters": [
    0.1234
  ],
  "argumentCount": 0
}
```

effectiveReactionparameters.json

The `.json` file generated by parScale has the following example structure:

```
{
  "name": "effectiveReactionparameters",
  "nparameters": 2,
  "multiplyByParticleVolumeFraction": true,
  "functionalForm": "OneMinusConversionTimesConcentration",
  "parameters": [
    0.6667,
    1
  ],
  "argumentCount": 3
}
```

generate-effectiveness-udf.js

The script used to generate the UDFs.

```
#!/usr/bin/env softshell
var entity = require('porto.entity');
porto.EffectivenessFactor = entity.using('effectivenessfactor', 'eu.nanosim.parscale', '0.2');
porto.EffectiveReactionparameters = entity.using('effectivereactionparameters', 'eu.nanosim.parscale', '0
porto.ChemkinReaction = entity.using('chemkinReaction', 'eu.nanosim.vasp', '0.1');

__main__ = function (args) {
    try {
        // Check that the user provides sufficient arguments to the program
        if (args.length < 2) {
            print("Usage " + args[0] + " <uuid>");
            return;
        }

        var uuid = args[1];

        // Attempt to talk to the local mongodb
     var storage = new porto.Storage("mongo2", "mongodb://localhost", "db=porto;coll=demo");

        // Load the collection given by the uuid on the command line, then retrieve all
        // reaction entities in this collection
        collection = new porto.Collection(uuid);
        storage.load(collection);

       var effectivenessfactorID = collection.findRelations("Effectivenessfactor", "id");
     var effectivereactionparametersID = collection.findRelations("Effectivereactionparameters", "id");

        var reactionDataIds = collection.findRelations("reactiondata", "has-id");

        var ef = new porto.EffectivenessFactor(effectivenessfactorID);
        ef.read(storage);

     if (ef.arguments.length != 0 || ef.parameters.length != 1 || ef.functionalForm != "constant") {
        throw "I do not understand any other functional forms for the effectiveness factor than the constant
```

```
        }

        var erp = new porto.EffectiveReactionparameters(effectivereactionparametersID);
        erp.read(storage);

        reactionDataIds.forEach(function (reactionId) {
            var reaction = new porto.ChemkinReaction(reactionId);
            reaction.read(storage);

        console.raw("/* ---[ GENERATED UDF ]----------------------------------------------------- */\n")
            console.raw("/* Collection uuid: " + collection.id() + " */ \n");
            console.raw("/* effectivenessfactorID: " + effectivenessfactorID + " */\n");
        console.raw("/* effectivereactionparametersID: " + effectivereactionparametersID + " */\n");
            console.raw("/* ChemkinReactionID: " + reactionId + "*/\n");

        // Pass the aggregated information from the entity to the code generator (soft.mvc). The
        // variables here (elements, species, reactions) can be found in the template specified
        // below. This will substitute the template entrires such as @{elements} with the contents
            // specified below.
            var controller = require('soft.mvc').create({
                model: {
                    k0: reaction.A / 10e6,
                    EA: reaction.Ea * 10e3,
                    eff: ef.parameters[0]
                },
                view: "./template/effectiveness-udf.cjs"
            });

            // Output the generated code directly to the console.
            console.raw(controller());
        });

    } catch (err) {
        // Any error caught during execution is logged to the console.
        console.raw("ERROR: Failed generating code.\nReason: " + err + "\n");
    }

};
```

udf.cjs

The template used to generate the UDFs.

```
#include "udf.h"

real min_cutoff = 1e-6;          /* Stops the reaction when reactants approach zero to ensure stability
real Gas_Const = 8.314;          /* Ideal gas constant [J/mol/K] */
real k0 = @{soft.model.k0};      /* Pre-exponential factor [m3/mol/s] from Chemkin files. Chemkin file
real EA = @{soft.model.EA};      /* Activation energy [J/mol] from Chemkin files. Chemkin file value [k

DEFINE_HET_RXN_RATE(Rrate,c,t,r,mw,yi,rr,rr_t)
{
   real Rrate, k, X_CH4, MW_CH4, X_Fe2O3, MW_Fe2O3, VOF, T, rho_g, rho_s, eff;
   Thread *tg = THREAD_SUB_THREAD(t,0);
```

```
    Thread *ts = THREAD_SUB_THREAD(t,1);

    VOF = C_VOF(c,ts);              /* Particle volume fraction [-]*/
    T = C_T(c,ts);                  /* Particle temperature [K]*/
    rho_g = C_R(c,tg);              /* Gas density [kg/m3] */
    rho_s = C_R(c,ts);              /* Particle density [kg/m3] */
    X_CH4 = yi[0][0];               /* Methane mass fraction (check second index if order of species in ga
    MW_CH4 = mw[0][0]/1000.;        /* Methane molar weight [kg/mol] (check second index if order of speci
    X_Fe2O3 = yi[1][1];             /* Hematite mass fraction (check second index if order of species in g
    MW_Fe2O3 = mw[1][1]/1000.;      /* Hematite molar weight [kg/mol] (check second index if order of spec

    k = k0 * exp(-EA/(Gas_Const*T)); /* Reaction rate constant calculated from Chemkin file inputs [m3/m

    eff = @{soft.model.eff};        /* Effectiveness factor from Parscale. Should probably be expressed as

    Rrate = eff * VOF * k * (rho_g*X_CH4/MW_CH4) * (rho_s*X_Fe2O3/MW_Fe2O3/6.);  /* Reaction rate [mol/m

    if (X_CH4 < min_cutoff) Rrate = 0;

    if (X_Fe2O3 < min_cutoff) Rrate = 0;

    if (VOF < min_cutoff) Rrate = 0;

    *rr = Rrate / 1000.0;           /* Return final reaction rate to FLUENT [kmol/m3/s] */
}
```

Use Case 4. Providing input to Phenom

This use case demonstrates how to use Porto to generate input to the MATLAB program Phenom. The code for this use case can be found in the `porto/src/phenom` folder of the Porto repository.

Phenom is a 1-D phenomenological model for fluidized bed reactors, which was developed to simulate the performance of second generation CO2 capture technologies with focus on chemical looping reforming (CLR) [1]. The model consists of a generic formulation based on an averaging probabilistic approach and can be used under bubbling, turbulent and fast fluidization regimes. The main purpose of using a 1-D phenomenological model instead of a more complex fundamental formulation is to provide valuable and sufficiently accurate information of industrial interest with less computational costs and within simulation times in the order of seconds/minutes. 1-D phenomenological models are effective tools for simulation, design and optimization of complex technologies while fundamental computational fluid dynamics (CFD) models are not so easily accessible by the industry. Phenom is implemented in Matlab.

1. Joana Francisco Morgado, Schalk Cloete, John Morud, Thomas Gurker, Shahriar Amini (2017). Modelling study of two chemical looping reforming reactor configurations: looping vs. switching. Powder Technology 316, 599–613

Walkthrough

Since MATLAB does not presently directly communicate with Porto, we choose to generate a MATLAB function that will return the full set of input parameters needed to run the Phenom code.

This is done by:

- Ensuring that a Porto entity instance described by `phenom-input.json` is populated with input data, and the entity is registered in the metadata database.

- Retrieving an `phenom-input` entity from the database by referring to its uuid.
- Populating a MATLAB function template with the contents of the `phenom-input` entity.
- Writing the MATLAB function to a `.m` file that can be evaluated.

This entire process can be performed by running the `define-phenom-input.js` and `make-phenom-input.js` script with the uuid of the `phenom-input` entity:

```
$ cd porto/src/phenom
$ soft-register-entity template/phenom-input.json # Register metadata. (Do this only once!)
$ ./define-phenom-input.js  # Populate an instance of the PhenomInput Entity and stores the data
a6a71841-139a-4310-a9e6-ef7a6f161a6f
$ ./make-phenom-input.js a6a71841-139a-4310-a9e6-ef7a6f161a6f > phenom_input.m
```

This creates the file `phenom_input.m`.

Call the main program of Phenom in Matlab using the input file as argument:

```
phenom_main_fvm('phenom_input');
```

This creates plots of the solution as well as an output file `output_cond.mat` that contains the reactor output streams (pressure, temperature, composition of the gas and solids phases).

Details

`phenom-input.json` entity description

The entity describing the input parameters to Phenom is structured in the following entity:

```
{
    "name": "phenom-input",
    "version": "1",
    "namespace": "eu.nanosim.phenom",
    "description": "Input parameters used to run the Phenom model",
    "dimensions": [
        {
            "name": "Nads",
            "description": "Number of adsorbing species."
        },
        {
            "name": "Ncomp",
            "description": "Number of gas species."
        },
        {
            "name": "Ncomp_T",
            "description": "Number of gas + solid species."
        },
        {
            "name": "Nrx_g",
            "description": "Number of reforming reactions."
        },
        {
            "name": "Ncomp_s",
            "description": "Number of het reactions."
        }
    ],
    "properties": [
```

```
{
    "name": "Ac",
    "type": "double",
    "description": "Cross sect bed area"
},
{
    "name": "ADENT",
    "type": "double",
    "dims": [
        "Nads"
    ],
    "description": "Enthalpy of adsorption CH4 CO H2 H2O."
},
{
    "name": "B",
    "type": "double",
    "dims": [
        "Ncomp"
    ],
    "description": "Viscosity coeffs CH4 CO CO2 H2 H2O N2."
},
{
    "name": "BET_a0",
    "type": "double",
    "description": "Spec surface area."
},
{
    "name": "CP",
    "type": "double",
    "dims": [
        4,
        "Ncomp"
    ],
    "description": "Heat capacity coeffs 4coeff x Ncomp species."
},
{
    "name": "CP_s",
    "type": "double",
    "description": "Solid heat transfer coeff."
},
{
    "name": "deltaHf",
    "type": "double",
    "dims": [
        "Ncomp"
    ],
    "description": "Heat of formation, gas (Ncomp species)."
},
{
    "name": "deltaHf_s",
    "type": "double",
    "dims": [
        "Ncomp_T - Ncomp"
    ],
```

```
    "description": "Heat of form Ni NiO MgAl2O4 ((Ncomp_T-Ncomp) species)."
},
{
    "name": "Dt",
    "type": "double",
    "description": "ReactorE diameter."
},
{
    "name": "E",
    "type": "double",
    "dims": [
        "Nads"
    ],
    "description": "Activation energies (Nads adsorption reactions)."
},
{
    "name": "Ea_s",
    "type": "double",
    "dims": [
        "Nrx_s"
    ],
    "description": "Act energy (Nrx_s solid rx)."
},
{
    "name": "ENT298",
    "type": "double",
    "dims": [
        "Nrx_g + 1"
    ],
    "description": "Reaction enthalpy 298 K ((Nrx_g+1) rx)."
},
{
    "name": "F_sol_w",
    "type": "double",
    "description": ""
},
{
    "name": "Flowin_w",
    "type": "double",
    "description": ""
},
{
    "name": "Fluxin_w",
    "type": "double",
    "description": ""
},
{
    "name": "FRACin",
    "type": "double",
    "dims": [
        "Ncomp"
    ],
    "description": "Mass fractions gas at inlet (Ncomp species)"
},
```

```
{
    "name": "FRACin_s",
    "type": "double",
    "dims": [
        "Ncomp_T - Ncomp"
    ],
    "description": "Mass fract at inlet solid   ((Ncomp_T-Ncomp) species)"
},
{
    "name": "Fw",
    "type": "double",
    "dims": [
        "Ncomp"
    ],
    "description": "Kg/s of each species  (Ncomp)"
},
{
    "name": "Gs0",
    "type": "double",
    "description": ""
},
{
    "name": "Heq",
    "type": "double",
    "dims": [
        "Nads"
    ],
    "description": "Equilibrium enthalpies (Nads rx)"
},
{
    "name": "K_0i",
    "type": "double",
    "dims": [
        "Nads"
    ],
    "description": "Pre-exponential factors, Nads adsorption reactions"
},
{
    "name": "k_0j",
    "type": "double",
    "dims": [
        "Nrx_g + 1"
    ],
    "description": "Preexp factors reforming (Nrx_g+1)"
},
{
    "name": "k0_s",
    "type": "double",
    "dims": [
        "Nrx_s"
    ],
    "description": "Preexp for heterogeneous kinetics (Nrx_s rx)"
},
{
```

```
    "name": "k0_s",
    "type": "double",
    "dims": [
        "Nrx_s"
    ],
    "description": "Preexp for heterogeneous kinetics (Nrx_s rx)"
},
{
    "name": "Keq_j",
    "type": "double",
    "dims": [
        "Nads"
    ],
    "description": "Equilibr constants, (Nads rx)"
},
{
    "name": "MW",
    "type": "double",
    "dims": [
        "Ncomp"
    ],
    "description": "Molar weights (Ncomp species)"
},
{
    "name": "MW_O2",
    "type": "double",
    "description": ""
},
{
    "name": "MW_s",
    "type": "double",
    "dims": [
        "Ncomp_T - Ncomp"
    ],
    "description": "Molar weight, (Ncomp_T-Ncomp) solid species"
},
{
    "name": "n_rx",
    "type": "double",
    "dims": [
        "Nrx_s"
    ],
    "description": "Reaction order heterogeneous rx  (Nrx_s)"
},
{
    "name": "Pin",
    "type": "double",
    "description": ""
},
{
    "name": "R0_NiAl2O4",
    "type": "double",
    "description": ""
},
```

```
{
    "name": "R0_NiO",
    "type": "double",
    "description": ""
},
{
    "name": "rhogm",
    "type": "double",
    "description": ""
},
{
    "name": "rhogmWin",
    "type": "double",
    "description": ""
},
{
    "name": "RHOin",
    "type": "double",
    "description": ""
},
{
    "name": "rhos0",
    "type": "double",
    "description": ""
},
{
    "name": "Smu",
    "type": "double",
    "dims": [
        "Ncomp"
    ],
    "description": "Viscosity parameter, (Ncomp) gases"
},
{
    "name": "Stoich",
    "type": "double",
    "dims": [
        "Ncomp",
        "Nrx_g + 1"
    ],
    "description": "Stoichiometric coeffs (Ncomp x (Nrx_g+1) rx)"
},
{
    "name": "Stoich_s",
    "type": "double",
    "dims": [
        "Ncomp_T",
        "Nrx_s + 1"
    ],
    "description": "Stoich for solids reactions (Ncomp_T x (Nrx_s+1) rx )"
},
{
    "name": "Tin",
    "type": "double",
```

```
        "description": ""
    },
    {
        "name": "Uin",
        "type": "double",
        "description": ""
    },
    {
        "name": "Yin",
        "type": "double",
        "dims": [
            "Ncomp"
        ],
        "description": "Mol fractions, Ncomp species"
    },
    {
        "name": "g",
        "type": "double",
        "description": ""
    },
    {
        "name": "R",
        "type": "double",
        "description": ""
    },
    {
        "name": "C1",
        "type": "double",
        "description": ""
    },
    {
        "name": "C2",
        "type": "double",
        "description": ""
    },
    {
        "name": "A0",
        "type": "double",
        "description": ""
    },
    {
        "name": "dg",
        "type": "double",
        "description": ""
    },
    {
        "name": "dp",
        "type": "double",
        "description": ""
    },
    {
        "name": "mug",
        "type": "double",
        "description": ""
```

```
        },
        {
            "name": "rhog",
            "type": "double",
            "description": ""
        },
        {
            "name": "P0",
            "type": "double",
            "description": ""
        },
        {
            "name": "Ea",
            "type": "double",
            "description": ""
        },
        {
            "name": "tm",
            "type": "double",
            "description": ""
        },
        {
            "name": "dm",
            "type": "double",
            "description": ""
        },
        {
            "name": "Nm",
            "type": "double",
            "description": ""
        }
    ]
}
```

phenom-input.m.js template file

The template used to generate the Matlab file is given below. This results in a Matlab file that defines one function. This function then populates an object with parameters and returns this.

```
function par = ReactionParams
    par.Nads = @{soft.model.dim.Nads};
    par.Ncomp = @{soft.model.dim.Ncomp};
    par.Ncomp_T = @{soft.model.dim.Ncomp_T};
    par.Nrx_g = @{soft.model.dim.Nrx_g};
    par.Nrx_s = @{soft.model.dim.Nrx_s};

    par.Ac = @{soft.model.Ac}
    par.ADENT = @{arraySpan(soft.model.ADENT)};
    par.B = @{arraySpan(soft.model.B)};
    par.BET_a0 = @{soft.model.BET_a0};
    par.CP = @{table(soft.model.CP, 4, 6)};
    par.CP_s = @{soft.model.CP_s};
    par.deltaHf = @{arraySpan(soft.model.deltaHf)};
    par.deltaHf_s = @{arraySpan(soft.model.deltaHf_s)};
```

```
par.Dt = @{soft.model.Dt};
par.E = @{arraySpan(soft.model.E)};
par.Ea_s = @{arraySpan(soft.model.Ea_s)};
par.ENT298 = @{arraySpan(soft.model.ENT298)};
par.F_sol_w = @{soft.model.F_sol_w};
par.Flowin_w = @{soft.model.Flowin_w};
par.Fluxin_w = @{soft.model.Fluxin_w};
par.FRACin = @{arraySpan(soft.model.FRACin)};
par.FRACin_s = @{arraySpan(soft.model.FRACin_s)};
par.Fw = @{arraySpan(soft.model.Fw)};
par.Gs0 = @{soft.model.Gs0};
par.Heq = @{arraySpan(soft.model.Heq)};
par.K_0i = @{arraySpan(soft.model.K_0i)};
par.k_0j = @{arraySpan(soft.model.k_0j)};
par.k0_s = @{arraySpan(soft.model.k0_s)};
par.Keq_j = @{arraySpan(soft.model.Keq_j)};
par.MW = @{arraySpan(soft.model.MW)};
par.MW_O2 = @{soft.model.MW_O2};
par.MW_s = @{arraySpan(soft.model.MW_s)};
par.n_rx = @{arraySpan(soft.model.n_rx)};

par.Pin = @{soft.model.Pin};
par.R0_NiAl2O4 = @{soft.model.R0_NiAl2O4};
par.R0_NiO = @{soft.model.R0_NiO};
par.rhogm = @{soft.model.rhogm};
par.rhogmWin = @{soft.model.rhogmWin};
par.RHOin = @{soft.model.RHOin};
par.rhos0 = @{soft.model.rhos0};
par.Smu = @{arraySpan(soft.model.Smu)};
par.Stoich = @{table(soft.model.Stoich, 6, 4)};
par.Stoich_s = @{table(soft.model.Stoich_s, 9, 4)};
par.Tin = @{soft.model.Tin};
par.Uin = @{soft.model.Uin};
par.Yin = @{arraySpan(soft.model.Yin)};

par.Param.g = @{soft.model.g};
par.Param.R = @{soft.model.R};
par.Param.C1 = @{soft.model.C1};
par.Param.C2 = @{soft.model.C2};
par.Param.A0 = @{soft.model.A0};
par.Param.dg = @{soft.model.dg};
par.Param.dp = @{soft.model.dp};
par.Param.mug = @{soft.model.mug};
par.Param.rhog = @{soft.model.rhog};

par.membrane.P0 = @{soft.model.P0};
par.membrane.Ea = @{soft.model.Ea};
par.membrane.tm = @{soft.model.tm};
par.membrane.dm = @{soft.model.dm};
par.membrane.Nm = @{soft.model.Nm};

par.Param.memb.P0 = @{soft.model.P0};
par.Param.memb.Ea = @{soft.model.Ea};
par.Param.memb.tm = @{soft.model.tm};
```

```
    par.Param.memb.dm = @{soft.model.dm};
    par.Param.memb.Nm = @{soft.model.Nm};
end
```