Excalibur-Neptune report
2047356-TN-14-01

Task 4.3
DSL and code design pattern for NEPTUNE

Ben Dudson, Peter Hill, Edward Higgins, David Dickinson, and Steven Wright

University of York

David Moxey

University of Exeter

March 31, 2022

Contents

1 Executive summary **2**

2 Approaches to Multi-Species Simulations **2**

3 DSL Adoption **5**

4 Conclusions **7**

5 References **8**

1 Executive summary

In this report we discuss approaches to code design and the adoption of domain specific languages (DSLs) for multi-species simulations in the BOUT++ [1] and Nektar++ [2] solvers. Specifically this report covers:

1. Approaches to code design in Bout++ that allow for single-species plasma simulations to be extended to multiple species.
2. How these multi-species models are expressed in a high-level DSL.
3. An overview of the DSLs that exist already in the Bout++ and Nektar++ frameworks and how a common UFL-like DSL might drive both Bout++ and Nektar++ solvers.

2 Approaches to Multi-Species Simulations

Our previous report (2047356-TN-05) focuses on the implementation of two multi-fluid simulations within the BOUT++ framework.

When developing such a simulation, if only a small number of species are required, such as in Hermes-2 [3] and STORM [4], then code can be copied and modified for each individual species relatively easily. However, as the number of species grows, so too does the length and complexity of the code. This makes the code increasingly error prone, difficult to test and difficult to maintain.

Instead, our previous report outlines two approaches to extending from single species simulations to multi species simulations.

Those two approaches demonstrated in Bout++ are:

- **Subclassing**, where the class hierarchy is used as a base from which to adapt and specialise the behaviour of each species.
- **State-Command Pattern**, where the states of multiple species are stored in a dictionary structure, and the evolution of those species is defined by a collection of model components.

These two approaches are implemented in the BOUT++ framework in SD1D [5] and Hermes-3 [6], respectively.

In SD1D, the model is constructed from numerous `Species` classes, which represent particular atomic or ion species, and `Reaction` classes which represent the interactions between different species. These interactions are orchestrated by a central object representing the whole system, containing a list of participating species and reaction objects. Figure 1 shows this simplistic design.

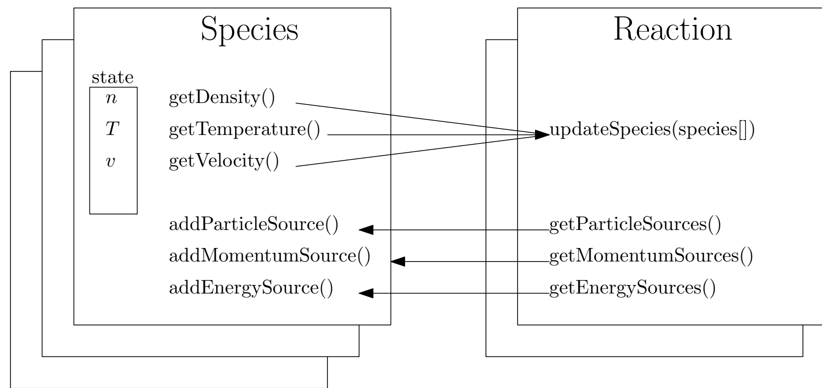

Figure 1: Subclassing architecture, in which `Species` objects describe plasma species, and `Reactions` represent interactions between them. All interactions are carried out through exchange of sources.

The main time look for SD1D is them implemented as follows:

```python
for species in species_list:
  species.evolve()

for reaction in reaction_list:
  # Calculate reaction rates using all species
  reaction.updateSpecies(species_list)

  for species_name, source in reaction.densitySources():
    # Check that the species is in species_list
    species_list[species_name].addDensity(source)

  for species_name, source in reaction.momentumSources():
    species_list[species_name].addMomentum(source)

  for species_name, source in reaction.energySources():
    species_list[species_name].addEnergy(source)
```

Figure 2: Psuedo-code for the main time loop in SD1D, showing how polymorphism is used to evolve multiple species.

In Hermes-3, a state-command pattern is used, where instead each individual species is specified in a nested dictionary-like structure, based on a simple schema, e.g., `state["species"]["h+"]["density"]` for the number density of hydrogen ions. A collection of composable model components are then defined that evolve the simulation state, performing calculations on single species or multiple species at once (e.g. collisions, etc). Figure 3 shows this design pattern.

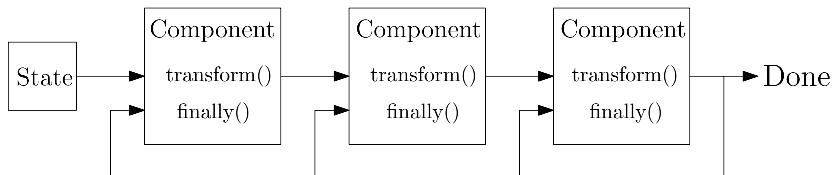

Figure 3: State-Command pattern: The `State` of the simulation system is passed through a sequence of `Components` in two passes. In the first pass (`transform()` calls), components can modify the state; in the second pass (`finally()` calls) the state cannot be modified, but is used to update component internal states.

Alternative approaches are also outlined in our previous report including the use of an Entity-Component System (ECS) and a task graph approach. The ECS approach is not dissimilar to our state-command approach detailed above, while a task graph approach can be taken together with either of our stated approaches. Task graphs are particularly of interest since they can be mapped directly to the SYCL/DPC++ programming model [7].

3 DSL Adoption

While our previous report covered the code structures that might be required to process multi-fluid simulations in Bout++, it did not elaborate how this might be exposed to a developer through a domain specific language (DSL).

In the T/NA086/20 [DSL and code generation] work package, a number of DSLs are outlined that exist at different levels of abstraction. Report 2057699-TN-02 details some of these DSLs and embedded DSLs (eDSLs) that are relevant to Project NEPTUNE.

At the lowest level are DSLs like RAJA, Kokkos and SYCL that provide mechanisms to specify regions of code for parallel execution. At a higher level, there are DSLs such as OP2 and OPS, where a developer are exposed to particular structures such as meshes, and they can write kernels that execute over elements of these structures (e.g. iterate over edges, nodes, etc.). At the highest level are DSLs such as UFL (used by Firedrake and FEniCS) and Bout++, where PDEs can be represented directly in code.

It is possible (and perhaps desirable) that multiple levels of DSL are used within a single application. This approach is already taken by Bout++. At its core, Bout++ is implemented in C++ with MPI; there are ongoing efforts to merge the low-level RAJA DSL with some of the core code, in order to provide heterogeneous loop-level parallelism. Bout++ then exposes a high-level eDSL to developers, allowing PDEs to be expressed relatively naturally in code. This approach is very similar to UFL. The C++ core is heavily templated, allowing the eDSL code to be translated at compile time to efficient code for execution.

The psuedo-code in Figure 2 provides an example of how to use Bout++'s primary DSL to create the SD1D code (the implementation can be seen in `sd1d.cxx`).

Bout++ also contains a secondary DSL (at a higher-level still) in its input files. This is used in the Hermes-3 simulation detailed above. Within the input DSL, we can specify multiple species and define their parameters and reactions. Figure 4 demonstrates this DSL.

Version 5.0 of the Nektar++ framework introduces a python interface for users that are less familiar with spectral elements and the Nektar++ framework itself [8]. This interface exists at a similar level to that of the UFL and primary Bout++ DSL.

Given the broad similarity in the three DSLs exposed by FEniCS, Bout++ and Nektar++, it is possible that common features could be extracted from the languages to form a common DSL that can drive both Bout++ and Nektar++ frameworks. However, defining a common DSL and building solvers that implement such a DSL will be a significant undertaking. Extending this DSL to multi-fluid reactions will also require that additional functionality is added to the Nektar++ framework.

```
[hermes]
components = d+, d, t+, t, e, collisions, sheath_boundary, recycling,
    reactions

# properties of each component
[d+]
type = evolve_density, evolve_momentum, evolve_pressure,
    anomalous_diffusion
AA = 2   # Atomic mass
charge = 1

[t]
...

[reactions]
type = (
        d + e -> d+ + 2e,   # Deuterium ionisation
        t + e -> t+ + 2e,   # Tritium ionisation
       )
```

Figure 4: An example of the Bout++ Ini input file describing a simple simulation with deuterium and tritium ions and atoms. The reactions are described in the input file.

4 Conclusions

In this report, we have outlined how Bout++ has been extended to support multi-fluid simulations. As part of this, the Bout++ DSLs have required extension to allow us to represent the complex reactions present in a multi-species simulation. In order to extend Bout++ to multi-fluid simulations, the embedded input DSL has also required extension, allowing run-time specification of component reactions.

While there are significant similarities in the interfaces exposed by Bout++ and Nektar++ (and indeed to UFL), there are areas of significant divergence in approach, meaning that it is likely to be challenging to drive both frameworks from a common DSL.

5 References

[1] BOUT++ contributors. BOUT++ manual. `https://bout-dev.readthedocs.io/`.

[2] C.D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R.M. Kirby, and S.J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[3] Ben Dudson, Jarrod Leddy, Hasan Muhammed. Hermes-2 hot ion drift-reduced model. `https://github.com/bendudson/hermes-2`.

[4] L. Easy, F. Militello, T. Nicholas, J. Omotani, F. Riva, N. Walkden, UKAEA. Hermes-2 hot ion drift-reduced model. `https://github.com/boutproject/STORM`.

[5] Ben Dudson et al. SD1D SOL and Divertor model in 1D. `https://github.com/boutproject/SD1D`.

[6] Ben Dudson. Hermes-3. `https://github.com/bendudson/hermes-3`.

[7] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. An experimental study of sycl task graph parallelism for large-scale machine learning workloads. Technical report, 2021.

[8] David Moxey, Chris D. Cantwell, Yan Bao, Andrea Cassinelli, Giacomo Castiglioni, Sehun Chun, Emilia Juda, Ehsan Kazemi, Kilian Lackhove, Julian Marcon, Gianmarco Mengaldo, Douglas Serson, Michael Turner, Hui Xu, Joaquim Peiró, Robert M. Kirby, and Spencer J. Sherwin. Nektar++: Enhancing the capability and application of high-fidelity spectral/hp element methods. *Computer Physics Communications*, 249:107110, 2020, doi:https://doi.org/10.1016/j.cpc.2019.107110.