ESCAPE 2

Deliverable D2.5:

Implementation of comprehensive domain specific language toolchain

Dissemination Level: public

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 800987

Funded by the European Union

Co-ordinated by ECMWF

Energy-efficient Scalable Algorithms for Weather and Climate Prediction at Exascale

Author: C. Müller, M. Röthlin, C. Osuna
Date: August 31, 2021

Research and Innovation Action
H2020-FETHPC-2017

Project Coordinator: Dr. Peter Bauer (ECMWF)
Project Start Date: 01/10/2018
Project Duration: 36 month
Published by the ESCAPE-2 Consortium

Version: 1.1
Contractual Delivery Date: 30.05.2021
Work Package/ Task: WP2
Document Owner: MSWISS
Contributors: MSWISS, DKRZ
Status: Final

1 Executive Summary

This document describes the work performed for the deliverable of a comprehensive domain specific language (DSL) toolchain. The DSL toolchain is a central pillar of the work performed in WP2 that allows to implement the ESCAPE-2 weather and climate dwarfs [2] using a high-level descriptive DSL language for code portability and performance portability. The use of the DSL for the ESCAPE-2 dwarfs allows to abstract away hardware specific implementation and advanced optimizations for different computing architectures. The deliverable is the set of software components, released as open source, that composes the entire DSL toolchain compiler. The toolchain is used to parse the weather and climate dwarfs and generate efficient kernels that are later compiled by vendor compilers. This document describes the architectural modular design adopted for the DSL developed within the ESCAPE-2 project, where the software is located and how to build and use the compiler. Section 3.2 describes the different components of the toolchain and how they interact. Section 3.3 provides a working example and finally Section 3.4 gives an overview of how the generated kernels of the DSL can interoperate with existing Fortran code and be inserted in model implementations.

2 Introduction

2.1 Backgroud

One of the aims of WP2 of ESCAPE-2 is to define, develop and apply a domain-specific language (DSL) toolchain applicable to a comprehensive list of algorithmic motifs (dwarfs) in weather and climate prediction. Domain specific languages are powerful tools that provide programming environments that allow to write numerical scientific algorithms in a concise and high level language. The weather and climate domain is characterized by very specific algorithmic motifs derived from the discretization of the numerical methods employed in the mathematical models, the specific aspect ratio of horizontal to vertical grids in regional and global models, and the use of sub-gridscale parametrization characterized by different algorithmic patterns. This motivates the development of a description suitable for these specific domain characteristics, using a highly concise and readable language. Details such as explicit loops, ordering of the loop nest, data layout or optimizations such as tiling are hardware specific optimizations that are abstracted away from such a high-level language. Among other things, the DSL language is abstracting away all the details of an efficient parallel implementation and the hardware dependent programming models and optimizations. There are several examples of DSLs being developed and applied to production weather and climate models, like COSMO GridTools[5],

the PSyclone for the LFRic model[1] or the CLAW DSL for column based parameterizations[3]. In contrast to the existing approaches, that are normally specifically developed for a particular model, the ESCAPE-2 DSL aims at developing a modular toolchain, that supports a wide range of models, numerical methods and grids, by adopting a modular design where domain specific frontends or optimizers can be easily incorporated into the toolchain. Additionally, most of the existing approaches provide a prescriptive language, where the user still has to provide information crucial for parallelization of the algorithm and to obtain good performance. Instead the goal of this document is to provide a high-level descriptive language where the algorithms are described in a sequential manner. The parallelization and optimization implementations are derived by the set of optimizers incorporated in the toolchain.

A diagram of such a toolchain is provided in Figure 1.

2.2 Scope of the Deliverable

2.2.1 Objectives of the Deliverable

The objective of this deliverable is to develop a comprehensive and modular DSL toolchain for the ESCAPE-2 dwarfs. The DSL toolchain will contain the following:

- High-level frontend that allows to express in a descriptive manner the ESCAPE-2 dwarfs used as a demonstration (deliverable D2.4).
- An implementation of the intermediate representation (HIR) as defined in D2.3 that is used to communicate the frontend with the compiler toolchain.
- A DSL compiler that incorporates safe checks and optimization passes which transform the user sequential code into efficient parallel computations.
- Code generators that generate efficient kernels that can be compiled by industry compiler for different architectures.

2.2.2 Work Performed on this Deliverable

DKRZ developed the CDSL (Community Domain Specific Language) frontend [2], which constitutes the application entry point for the ESCAPE2 toolchain. It provides the user a high-level language that allows the user to express a broad range of algorithms of the climate and weather domain. The frontend is embedded in C++ and uses the clang frontend of the LLVM compiler. The complete

toolchain including the frontend supports dwarfs on two types of grids: Cartesian as well as unstructured grids (like the icosahedral grid), as defined in [8].

The CDSL processes the user computations described using the DSL language and generates the corresponding HIR instance. For that, the DSL toolchain provides a library with an API in python to easily create HIR instances. The HIR software was developed within the dawn [6] project.

MeteoSwiss developed an entire toolchain compiler, named dawn [6], that contains the HIR parser, DSL compiler and different code generators. Furthermore, MeteoSwiss also developed a custom front end for dawn, called dusk, that is purpose built for the computational patterns contained in the (dry) ICON dycore. The HIR provides a language independent implementation of the HIR specification [8]. It provides support to create HIR instances using json or by using the python API. The dawn DSL compiler takes as input the HIR instance as a sequential description of the computations by CDSL. A set of compiler optimizers organize the computations for a parallel execution model, resolves data dependencies by splitting parallel regions when required, fused computations according to data locality, etc. All the optimizers are integrated within the dawn [6] compiler and are shared between the Cartesian grid and the unstructure grid modes of dawn. Finally dawn also incorporates a set of code generators that generate efficient code for the dwarfs. There are specific code generators for Cartesian grids and the unstructured grids, since the structure of the kernels differ considerably and even the API of the code generation changes (due to the need of lookup tables that describe horizontal connectivity by the unstructure grid mode). For each of the grid modes, dawn provides an efficient CUDA backend code generator as well as naive C++ code generator (CPU) used for verification and debugging purposes.

2.2.3 Deviations and counter measures

The final work to put all the components of the toolchain together caused a small delay in the final deliverable (due by June 2021). In particular the final work for a smooth integration between frontend (developed by DKRZ) and the toolchain compiler (developed by MSWISS). There was also additional work in order to ensure consistency between the SIR and HIR definitions and to make the compilation of the entire software more portable (as reflected in the document) and compilable with the software stack of general linux distributions like Ubuntu.

This delay affected the final preparation of the software product, but did not have an impact on other deliverables, since the software toolchain was already in use for selected ESCAPE-2 dwarfs ported to the DSL.

3 Toolchain

In this section installation instructions for the CDSL+dawn and the dusk+dawn toolchains are provided first. Then there is a general discussion of the toolchain followed by a concrete example. Finally, the FORTRAN interoperability is briefly illustrated.

3.1 Toolchain Installation Instructions

3.1.1 Using dawn with the CDSL frontend

In the following a general list of requirements and installation instructions for the CDSL+dawn toolchain for any Linux distribution are provided with more concrete instructions for Ubuntu 20.04.

In general the following packages are required:

```
git
make

cmake > 3.13
gcc/g++/gfortran > 8.3.0
clang/llvm >= 10.0
python >= 3.8
boost >= 1.58
ecbuild >= 3.1.0
eckit >= 1.4.0
netcdf-fortran >= 4.5.2
```

On Ubuntu 20.04 one can satisfy these requirements by running:

```
sudo apt install git make g++ gfortran libeckit-dev \
llvm clang llvm-dev libclang-dev libclang-cpp10-dev \
python3-dev python3-venv python3-clang libboost-dev \
libnetcdff-dev nvidia-cuda-toolkit
sudo snap install cmake --classic
```

Next Atlas [4] is built following the instructions on the Atlas Github:

```
git clone https://github.com/ecmwf/ecbuild.git
git clone https://github.com/ecmwf/atlas.git

pushd atlas
```

```
    # Switch to supported commit
    git checkout 66e40483fa0b2882f1e412e7

    # Environment --- Edit as needed
    ATLAS_SRC=$(pwd)
    ATLAS_BUILD=build
    ATLAS_INSTALL=$(pwd)/install

    # 1. Create the build directory:
    mkdir $ATLAS_BUILD
    pushd $ATLAS_BUILD

        # 2. Run CMake
        ~/ecbuild/bin/ecbuild --prefix=$ATLAS_INSTALL \
        -- $ATLAS_SRC

        # 3. Compile
        make -j 8
        make install

        # 4. Check installation
        $ATLAS_INSTALL/bin/atlas --info
    popd
popd
```

Then a Python virtual environment is set up.

```
python3 -m venv dawn-dusk-venv

source dawn-dusk-venv/bin/activate

pip install --upgrade pip setuptools wheel
```

Next dawn is cloned, configured, compiled and installed in the virtual environment:

```
git clone https://github.com/MeteoSwiss-APN/dawn.git

pushd dawn
    mkdir build

    cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug \
    -DBUILD_TESTING=ON \
    -Datlas_DIR=~/atlas/install/lib/cmake/atlas \
    -DDAWN_REQUIRE_UNSTRUCTURED_TESTING=ON
```

```
    pushd build
        make -j 8

        ctest # all tests should pass!
    popd
popd

pip install -e dawn/dawn
```

Depending on if the device has a CUDA-enabled graphics cards, either all tests will pass or some tests, which depend on CUDA, will fail. To check if only CUDA-related tests failed one can run `ctest --rerun-failed --verbose`. All failing tests should display the "no CUDA-capable device is detected" error.

Now CDSL can be set up in the following way:

```
git clone \
https://gitlab.dkrz.de/escape2/cpp-dsl-front-end.git

pushd cpp-dsl-front-end
    # copy config file used for build
    cp ./conf/user_config-example-ubuntu.sh \
    ./user_config.sh
    # configure, build and run tests
    ./cmake_build.sh 1 && ./cmake_build.sh 2 \
    && ./cmake_build.sh 3
popd
```

The CDSL binary is available as `cpp-dsl-front-end/build/bin/cdsl` and has the following usage instructions:

```
usage: cdsl [-h] [-I dir] [-o file] [-s stage] [-json]
[-D macro] [-nopath] [-b backend] file

positional arguments:
  file         DSL source code file

optional arguments:
  -h, --help   show this help message and exit
  -I dir       include directory
  -o file      output filename
  -s stage     max processing stage: one of clang_filter,
               parser, AST, SA, HIR, SIR, CPP, CU
  -json        json output
  -D macro     cpp macro definition
  -nopath      do not encode the full filepath
  -b backend   one of dawn backends (CXXNaive,
```

```
CXXNaiveIco or CUDAIco)
```

3.1.2 Using dawn with the dusk frontend

Installing the alternative frontend to dawn called dusk is a simple matter given the setup from the last section. One has to activate the `dawn-dusk-venv`, clone dusk and install it in the environment:

```
source dawn-dusk-venv/bin/activate

git clone https://github.com/dawn-ico/dusk.git

pip install -e dusk
```

Now the `dusk-front` binary is available in the activated virtual environment. Together with the `dawn-opt` and `dawn-codegen` binaries in `dawn/build/dawn/bin/` the toolchain is complete.

3.2 Toolchain Program Description

The toolchain was designed in modular fashion. Roughly, it can be divided into three stages: the front end, the compiler stage, and finally the compilation into binaries using a host compiler. A diagram of the architecture is provided in figure 1.

The frontends, that is dusk and the community dsl (cdsl), respectively, take user code and transform said user code into a representation called the Stencil Intermediate Representation (SIR). The SIR representation is a specific version of the HIR described in [8][1]. Its purpose is to describe stencil like computations prevalent in climate and numerical weather prediction.

This SIR is then fed into the compiler stage, which is again subdivided into two logical units: the optimization stage `dawn-opt` and `dawn-codegen`. The first job of the optimization stage is to transform the user code into valid parallel code. This, for example, includes field versioning. After these transformations various optimization passes ensure efficient execution of the emitted code. A rich set of validation passes (e.g. type checking) is run after the initial lowering of the representation to parallel code, as well as after each optimization pass.

[1] SIR is the current version of the HIR specification supported by the toolchain. Its functionality supports fully the description of all the ESCAPE-2 dwarfs targeted by the DSL. The main difference with respect to the design of the HIR [8] is the generalization of some of the concepts of the HIR for a broader set computational patterns (beyond the ESCAPE-2 dwarfs) like the ability to define any number of arbitrary dimensions of a field.

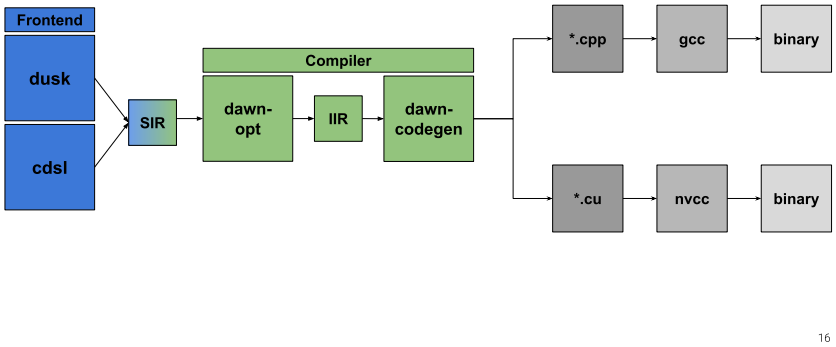

Figure 1: Schematic view of the toolchain program architecture

The purpose is two-fold: on one hand the user can be informed if there is a mistake in the program, on the other hand these validation passes serve as a post-condition contract for each optimization pass, enabling rigorous testing of the optimization facilities. The available passes in dawn are summarized in tables 1 and 2.

All of these passes operate on a representation called the Intermediary Intermediate Representation (IIR). It differs from the SIR mostly in two aspects: first the computations are hierarchically compartmentalized such that they are safe to be executed in parallel. Additionally decorations / meta information important to reason about possible optimizations are added.

The final stage of the toolchain is the code generation. Here, each of the IIR nodes is processed in recursive fashion and tokens in the target language are emitted. Two such backends are implemented: A C++ backend which focuses on legibility and is mostly meant for debugging purposes. It is also useful for a programmer wishing to learn using the toolchain, since it provides a way to easily exercise the front end code. The other backend is a raw CUDA codegenerator and is meant to ensure the highest possible performance on GPUs.

Both backends emit program code as text. This ensures that the programmer is free to inspect the emitted code, as well as chose the appropriate compiler for the host computer at hand.

Pass Name	Description
Grid Type Checker	Prevents that the user mixes Cartesian with unstructured code
Indirection Checker	Prevents vertically indirected writes
Integrity Checker	Ensure that AST is well formed at all times
Unstructured Dimensions Checker	Ensure that unstructured user code is consistent in location types (edge, cell, vertex)
Weight Checker	Ensure that weights passed into a reduction are of the correct type

Table 1: List of validation passes

Pass Name	Description
Stage Reordering	Rearrange computation stages to maximize merging potential
Stage Merger	Merge adjacent stages if possible
Pass Temporary Type	Demote temporary fields to scalars if possible
Interval Partitioning	Reorganizes vertically overlapping computations into a non overlapping ordered set of interval computations to maximize fusing potential
Temporary Merging	Reduces the number of emitted temporaries (e.g. due to field versioning) to a minimum.
Inlining	Avoid memory accesses by inlining computations saved to a temporary
Set Block Size	Determine optimal CUDA block size heuristically
Set Caches	Chose optimal caching strategy for CUDA backend

Table 2: List of validation passes, see also [7].

3.3 Worked Example

To illustrate the explanations above, a worked example is presented in this section. For this purpose, lets chose a stencil given in [10] and implement it in cdsl. The notation has been adapted for better legibility:

$$\zeta_q = \frac{1}{\hat{a}_q} \sum_{j}^{j \in \mathcal{E}(q} v_j^n \cdot \hat{l}_j \cdot f_j \qquad |\forall_q \in \mathcal{V}$$

This stencil computes the curl at a vertex q given the normal velocity v^n at the edges j around q. That is, the set $\mathcal{E}(q)$ denotes the edge neighbors of vertex q. $\hat{a}_q$ is the area of the dual cell at q, whereas $\hat{l}_j$ is the length of dual edge j and f_j is an orientation factor ± 1 to ensure that the Stokes theorem is implemented properly. $\mathcal{V}$ is simply the set of all vertices in the mesh. The situation is illustrated in figure 2.

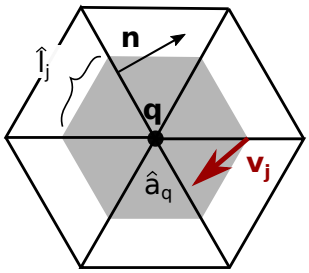

Figure 2: Geometrical representation of the stencil used in the worked example.

This stencil is readily implemented using cdsl (abridged listing, consider the appendix 6.1 for the complete listing):

```
void curl(VK_Field zeta_q,
          EK_Field vn,
          EK_Field dualL,
          VEK_Field f,
          VK_Field dualA) {
        zeta_q = nreduce(edges, vn * dualL * f) / dualA
}
```

In the listing above, a few things are of note. All the fields have a type, indicating their location. In conjunction with the type annotation of the reduction `nreduce` this provides safety already at compile time, preventing the user from performing a reduction that is intended from cells to edges, say, to a vertex field. Types can also represent so called *sparse* fields. This is the case for the orientation

f, which is of type `VEK_Field`. This is because for each vertex, there is an orientation stored for each of that vertex' neighbors (e.g. 6 in the case of an icosahedral mesh). We can now use the cdsl frontend to produce the SIR.

```
cdsl -s SIR -json curl.cpp -o curl.sir
```

Then use `dawn-opt` to emit the IIR.

```
dawn-opt --default-opt curl.sir -o curl.iir
```

And finally use `dawn-codegen` to emit the code for the host compiler.

```
dawn-codegen --atlas-compatible curl.iir -b cuda-ico -o curl.cpp
```

We take care to emit code compatible with the Atlas meshing library [4]. This allows using atlas meshes in the driver code, allowing for easy testing of the complete stencil (i.e. an integration test). Figure 3 shows the stencil applied to a simple test function.

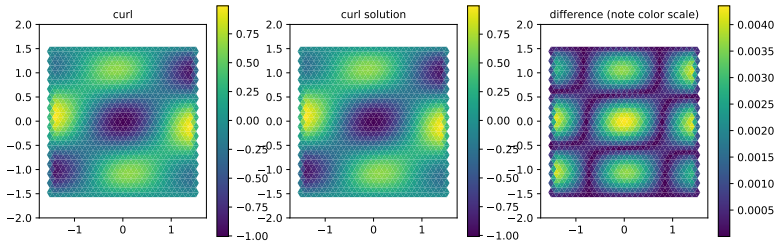

Figure 3: The curl operator implemented in this section applied to some simple spherical harmonics.

3.4 dawn to FORTRAN interoperability

The primary use case of the DSL approach in NWP and climate is not to develop a model from scratch using the DSL, but porting existing parts of a model. Most model code is in FORTRAN. The toolchain thus provides facilities to easily inter-operate with existing FORTRAN code. An interface is provided to either just launch a DSL kernel from FORTRAN, or additionally provide reference solutions from the FORTRAN codebase to efficiently verify the DSL code. This allows for incremental porting of existing model code to DSL on a per-stencil basis. The toolchain can be instructed to emit a FORTRAN interface using

```
dawn-codegen --atlas-compatible curl.iir -b cuda-ico
   --output-f90-interface curl.f90 -o curl.cpp
```

A block diagram of the FORTRAN interoperability is given in figure 4. An excerpt of a FORTRAN interface for the worked example from section 3.3 is given in appendix 6.2.

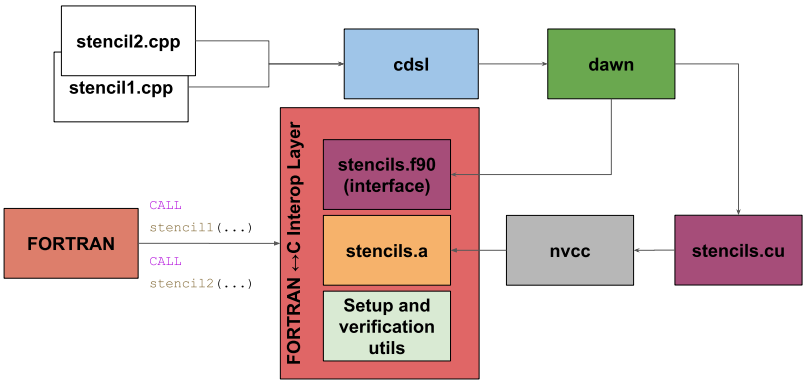

Figure 4: Block diagram of FORTARN interoperation

4 Conclusions

ESCAPE-2 has successfully developed an entire DSL toolchain that is capable of processing the target weather and climate dwarfs and generate efficient kernels that can be integrated into existing models. During early work of the project, in closed collaboration with developers from various models, a comprehensive definition of a high-level language for a DSL for weather and climate models was defined in D2.1 [8]. A formal definition of an intermediate representation, HIR, that captures the DSL language elements of D2.1 was then fully described in D2.3 [9]. Finally this deliverable describes the entire set of software components developed to implement this DSL language. The toolchain adopts a modular design, which allows to integrate various frontends as well as multiple computing architecture specific backends. The toolchain presented here is composed of the entry level frontend, CDSL, that parses the dwarfs computations implemented using the DSL language, and the compiler dawn, which incorporates an implementation of the HIR specification as well as various code generators. Finally, both the CDSL frontend as well as the dawn compiler support a wide variety of computational motifs as well as grids, including the Cartesian grid and unstructured (icosahedral) grid models.

5 References

[1] S. V. Adams et al. "LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models". In: *Journal of Parallel and Distributed Computing* 132 (2019), pp. 383–396. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.02.007. URL: https://www.sciencedirect.com/science/article/pii/S0743731518305306.

[2] J. Behrens et al. *ESCAPE2 D2.4: Demonstration of domain specific language toolchain for selected weather and climate dwarfs*. 2021.

[3] V. Clement et al. "The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '18. Basel, Switzerland: Association for Computing Machinery, 2018. ISBN: 9781450358910. DOI: 10.1145/3218176.3218226. URL: https://doi.org/10.1145/3218176.3218226.

[4] W. Deconinck et al. "Atlas : A library for numerical weather prediction and climate modelling". In: *Computer Physics Communications* 220 (2017), pp. 188–204. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2017.07.006. URL: http://www.sciencedirect.com/science/article/pii/S0010465517302138.

[5] T. Gysi et al. "STELLA: a domain-specific tool for structured grid methods in weather and climate models". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807627.

[6] C. Müller et al. *dawn*. URL: https://github.com/MeteoSwiss-APN/dawn/.

[7] C. Osuna et al. "Dawn: a high-level domain-specific language compiler toolchain for weather and climate applications". In: *Supercomputing Frontiers and Innovations* 7.2 (2020), pp. 79–97.

[8] C. Osuna et al. *ESCAPE2 D2.1: High-Level Domain Specific Language (DSL) specification*. 2019.

[9] C. Osuna et al. *High-level intermediate (HIR) representation specification*. 2020.

[10] G. Zängl et al. "The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core". In: *Quarterly Journal of the Royal Meteorological Society* 141.687 (2015), pp. 563–579.

6 Appendix

6.1 Complete cdsl file for the worked example

```
#include "dsl.hpp"

using namespace EDSL;

namespace edsl {

  Gridspace ek_space(edges,levels);
  define_field_type(EK_Field,ek_space);

  Gridspace vk_space(verts,levels);
  define_field_type(VK_Field,vk_space);

  Gridspace vek_space(verts,edges,levels);
  define_field_type(VEK_Field,vek_space);

  Gridspace ecek_space(edges,cells.edges,levels);
  define_field_type(ECEK_Field,ecek_space);

  void curl (VK_Field zeta_q,
             EK_Field vn,
             EK_Field dualL,
             VEK_Field f,
             VK_Field dualA) {
      zeta_q = nreduce ( edges , vn * dualL * f ) / dualA;
  }
}
```

6.2 FORTRAN interface for the worked example (excerpt)

```
module curl
use, intrinsic :: iso_c_binding
implicit none
  interface
    real(c_double) function &
    run_curl( &
    vn, &
    dualL, &
    dualA, &
    f, &
    zeta_q &
    ) bind(c)
```

```
      use, intrinsic :: iso_c_binding
      real(c_double), dimension(*), target :: vn
      real(c_double), dimension(*), target :: dualL
      real(c_double), dimension(*), target :: dualA
      real(c_double), dimension(*), target :: f
      real(c_double), dimension(*), target :: zeta_q
    end function
  end interface
end module
```

Document History

Version	Author(s)	Date	Changes
1.0	Christoph Müller and Matthias Röthlin	15.07.2021	first version
1.1	Christoph Müller and Matthias Röthlin	19.08.2021	final version with all review comments

Internal Review History

Version	Author(s)	Date	Changes
1.0	Kim Serradell	22.07.2021	implement review comments
1.0	Mike Gillard	23.07.2021	implement review comments

Effort Contributions per Partner

Partner	Efforts
MSISS	24 PM
DKRZ	4 PM
Total	28 PM

ECMWF Shinfield Park Reading RG2 9AX UK
Contact: peter.bauer@ecmwf.int