Common Community Physics Package (CCPP) Overview

Grant Firl[1,2], Dom Heinzeller[1,3,4], Ligia Bernardet[1,3], Laurie Carson[1,2], Man Zhang[1,3,4], Julie Schramm[1,2]

[1]DTC
[2]NCAR/RAL/JNT
[3]NOAA GSL
[4]CIRES

UFS Training — September 23, 2021

Outline

- What is the CCPP?
- How does the CCPP fit within a modeling system?
- How are CCPP physics suites defined?
- What makes a piece of code CCPP-compliant?
- How does a host model use the CCPP?
- Further resources

CCPP is the infrastructure for physics development

… facilitate the improvement of physical parameterizations and their transition from research to operations by enabling the community to participate in the development and testing …

Common Community Physics Package (CCPP)

https://github.com/NCAR/ccpp-framework
https://github.com/NCAR/ccpp-physics
https://github.com/NCAR/ccpp-doc

Consolidated: Single library of operational and developmental parameterizations and suites for all applications

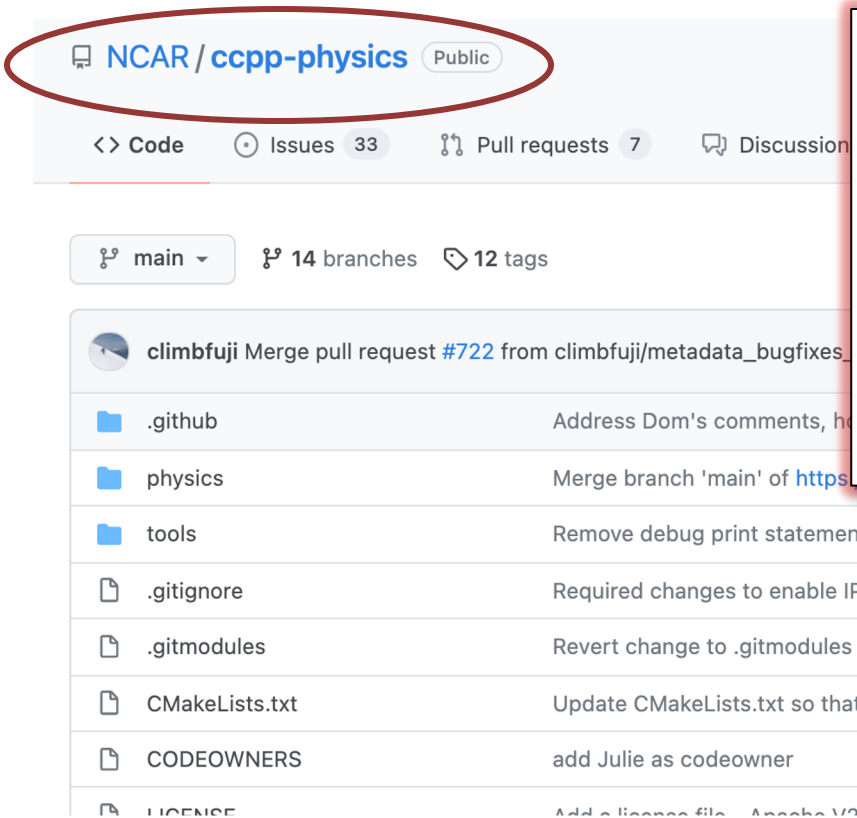
Supported: Well-supported community code

Open: Have accessible development practices (GitHub)

Clear interfaces: Well documented and defined interfaces to facilitate using/enhancing existing parameterizations and adding new parameterizations

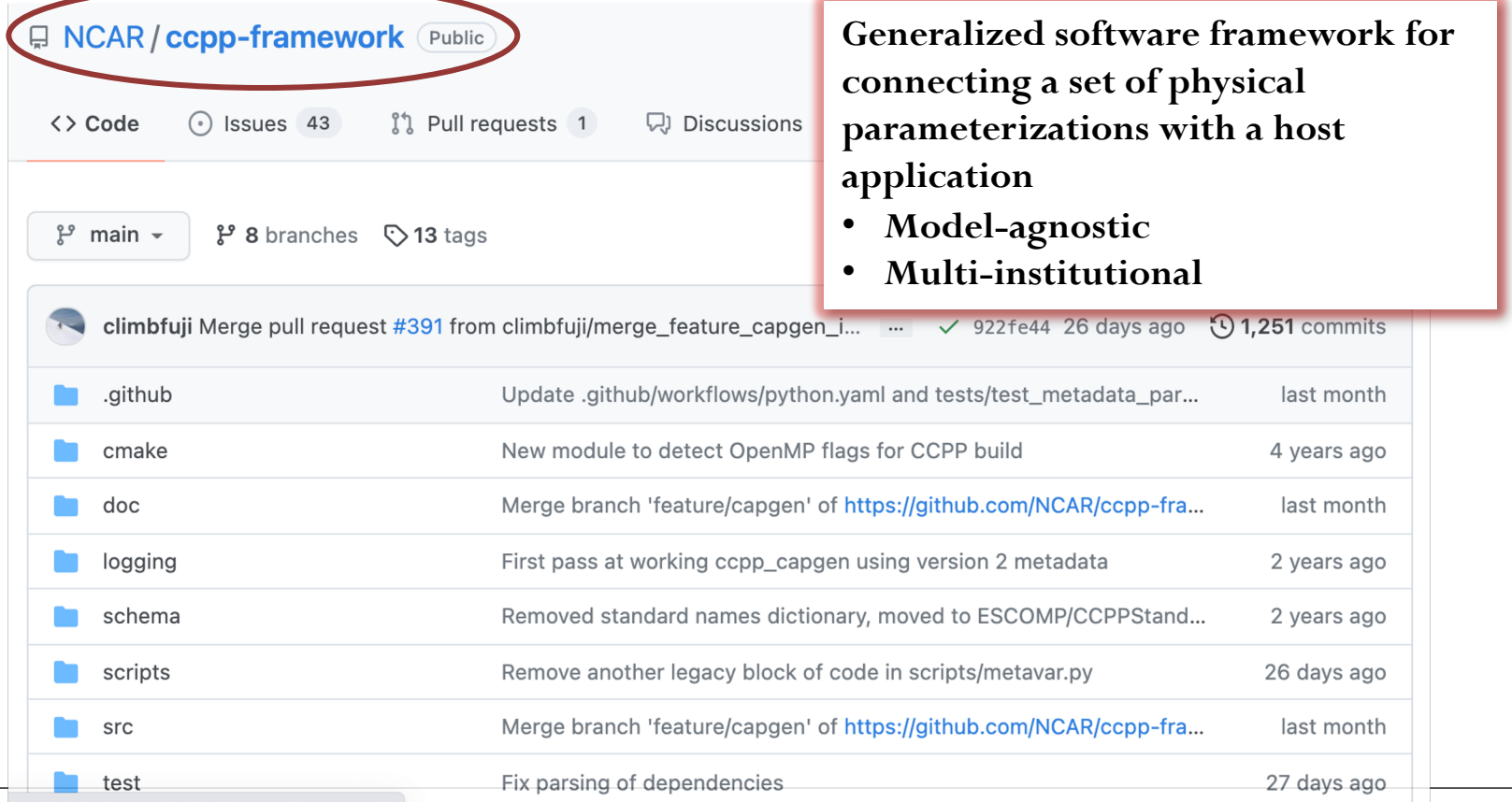
Interoperable: usable with other dycores/hosts to increase scientific exchange

What is the CCPP? (1 of 3)



- Library of physical parameterizations
- Authoritative fork contains:
  - Operational
  - Candidates for upcoming implementations
- Third-party forks can be used to contain compliant schemes used/developed in other institutions

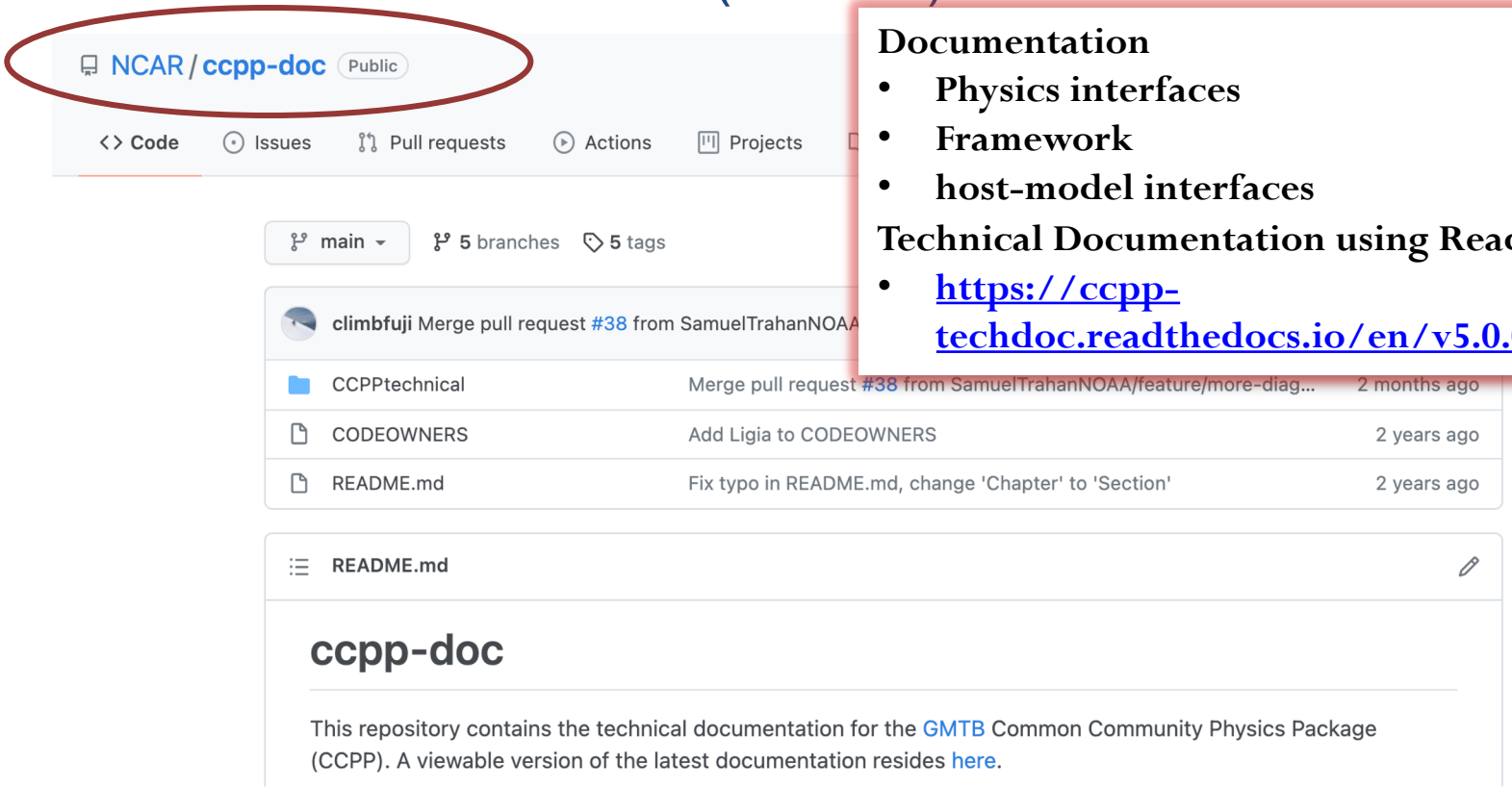
What is the CCPP? (2 of 3)



Generalized software framework for connecting a set of physical parameterizations with a host application

- **Model-agnostic**
- **Multi-institutional**

What is the CCPP? (3 of 3)



Documentation

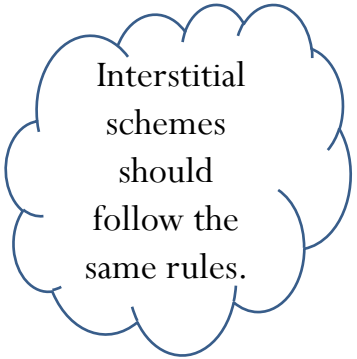
- **Physics interfaces**
- **Framework**
- **host-model interfaces**

Technical Documentation using ReadTheDocs

- **https://ccpp-techdoc.readthedocs.io/en/v5.0.0/**

What constitutes a CCPP “scheme”?

- Any piece of code with a CCPP-compliant interface:
  - Code must be wrapped within a Fortran module
  - Must contain init, run, and finalize subroutines
  - Must contain CCPP-readable metadata describing argument variables for all subroutines (init/run/finalize)
  - Use CCPP error-tracking variables rather than printing/stopping
  - Have formatted scientific/technical documentation
  - Conform to modern coding standards
- Scheme independence: smallest functional unit possible
  - if scheme functions will always be called together, OK to keep as one
  - if scheme functions will operate independently, separate the schemes



What makes a scheme interoperable?

- Well-defined and documented entry points
- Readability of the code (faciliate debugging)
- Expose constants and tuning parameters
- Avoid use of derived or compounded data types
- Standardized error handling, communication

metadata

variable intents

explicit import statements

private/public declarations

use constants and tuning parameters from host model

no assumption on data storage model

no interference with host model's logging/error handling strategy

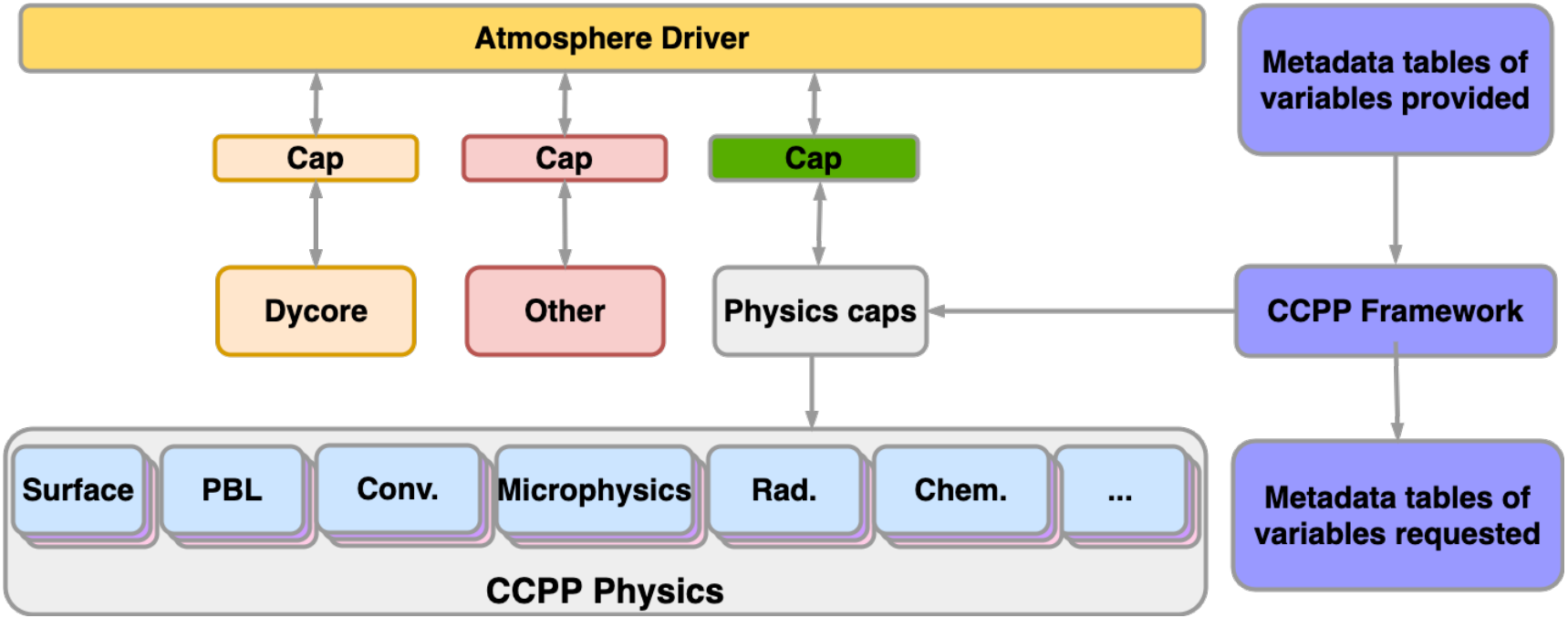
no assumptions about parallelization strategy

test and develop with different compilers and optimization flags

Primary vs “Interstitial” Schemes

- **Primary Scheme**: a parameterization, such as PBL, microphysics, convection, and radiation, that fits the traditionally-accepted definition.
- **Interstitial Scheme**: a modularized piece of code to perform data preparation, diagnostics, or other “glue” functions that allows primary schemes to work together as a suite.
  - AKA: the code in a traditional physics “driver” between physics scheme calls

The CCPP Within the Model System



CCPP Physics Suite Definition

- Individual CCPP-compliant physics parameterizations are assembled and controlled via an XML file called a

  "**S**uite **D**efinition **F**ile" (**SDF**)

- The SDF XML schema has the following hierarchy:
  - Suite

    Top-level element; defines the suite name and XML schema version

    - Group

      Schemes under one group always get called together in-sequence; non-physics code can be executed between physics groups

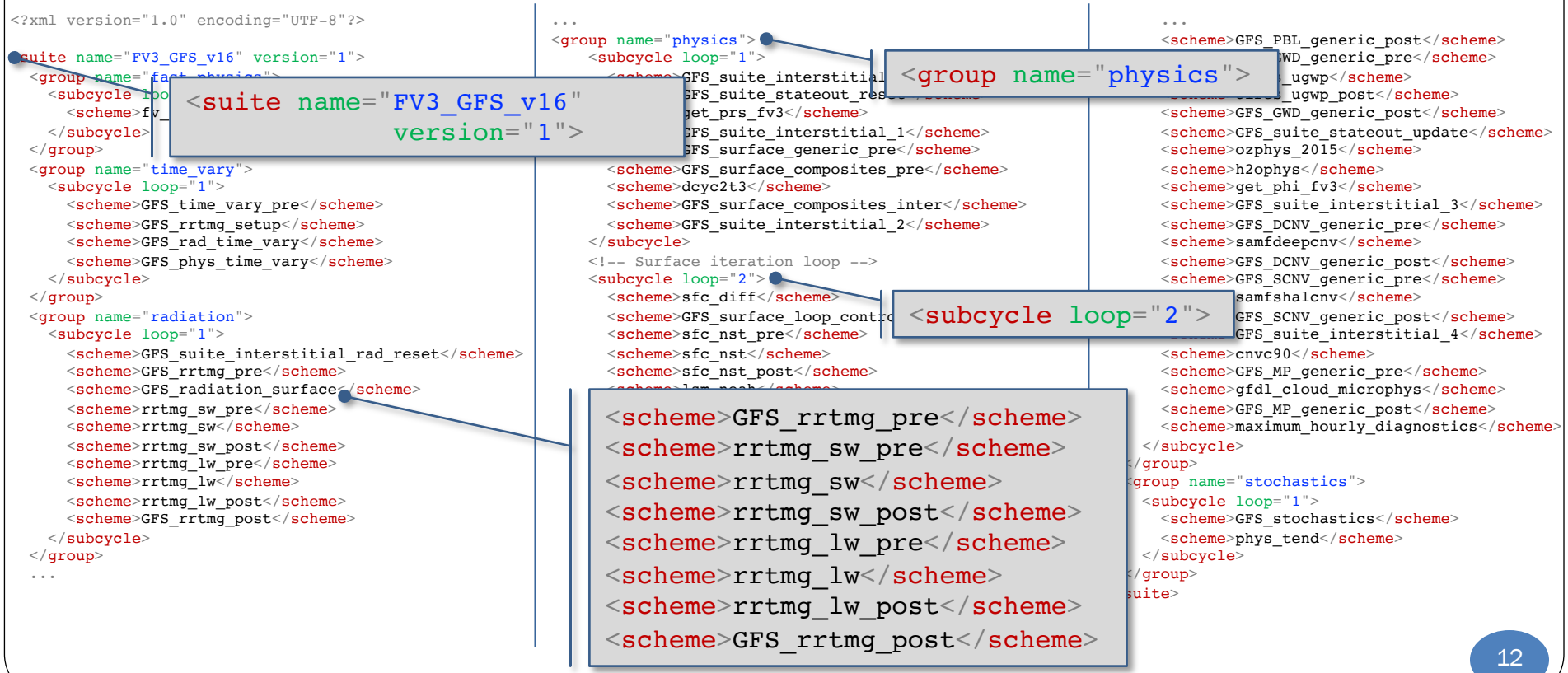
      - Subcycle

        Schemes within a subcycle element are executed N times according to the element's "loop" variable

        - Scheme

          Each scheme element contains the name of the scheme to run.

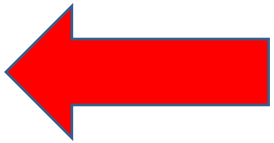
CCPP uses XML suite definition files at build time



Basic code structure

```
module myscheme

  implicit none

  contains

  subroutine myscheme_init ()
  end subroutine myscheme_init

!> \section arg_table_myscheme_run Argument Table
!! \htmlinclude myscheme_run.html
!!
  subroutine myscheme_run(ni, psfc, errmsg, errflg)
    integer,           intent(in)    :: ni
    real,              intent(inout) :: psfc(:)
    character(len=*),  intent(out)   :: errmsg
    integer,           intent(out)   :: errflg
    ...
  end subroutine myscheme_run

  subroutine myscheme_finalize()
  end subroutine myscheme_finalize

end module myscheme
```

myscheme.F90



"Hook" for CCPP metadata

CCPP scheme metadata

```
[ccpp-table-properties]
  name = myscheme
  type = scheme
  dependencies = other_file.F90

[ccpp-arg-table]
  name = myscheme_run
  type = scheme
[stress]
  standard_name = surface_wind_stress
  long_name = surface wind stress
  units = m2 s-2
  dimensions = (horizontal_loop_extent)
  type = real
  kind = kind_phys
  intent = in
  optional = F
...
```

myscheme.meta

Start of new metadata "table"

name of attached subroutine/module

type = [**scheme**, module, DDT, host]

CCPP scheme metadata

```
[ccpp-table-properties]
  name = myscheme
  type = scheme
  dependencies = other_file.F90

[ccpp-arg-table]
  name = myscheme_run
  type = scheme
[stress]
  standard_name = surface_wind_stress
  long_name = surface wind stress
  units = m2 s-2
  dimensions = (horizontal_loop_extent)
  type = real
  kind = kind_phys
  intent = in
  optional = F
...
```

myscheme.meta

name of variable in subroutine

the key by which this data is known in the CCPP

more descriptive name if standard name is not sufficient

note the format; possibility of automatic unit conversion among schemes and between host

CCPP scheme metadata

```
[ccpp-table-properties]
  name = myscheme
  type = scheme
  dependencies = other_file.F90

[ccpp-arg-table]
  name = myscheme_run
  type = scheme
[stress]
  standard_name = surface_wind_stress
  long_name = surface wind stress
  units = m2 s-2
  dimensions = (horizontal_loop_extent)
  type = real
  kind = kind_phys
  intent = in
  optional = F
...
```

myscheme.meta

standard names of array dimensions;
() for scalar;
can specify start:end for dimension
(default is 1)

FORTRAN intrinsic type or
DDT name

precision or character length

FORTRAN argument intent

FORTRAN optional argument

CCPP scheme metadata

```
[ccpp-table-properties]
  name = myscheme
  type = scheme
  dependencies = other_file.F90

[ccpp-arg-table]
  name = myscheme_run
  type = scheme
[stress]
  standard_name = surface_wind_stress
  long_name = surface wind stress
  units = m2 s-2
  dimensions = (horizontal_loop_extent)
  type = real
  kind = kind_phys
  intent = in
  optional = F
...
```

myscheme.meta

Applies to entire scheme; dependencies attribute allows compiling only those files that are necessary for a given list of suites

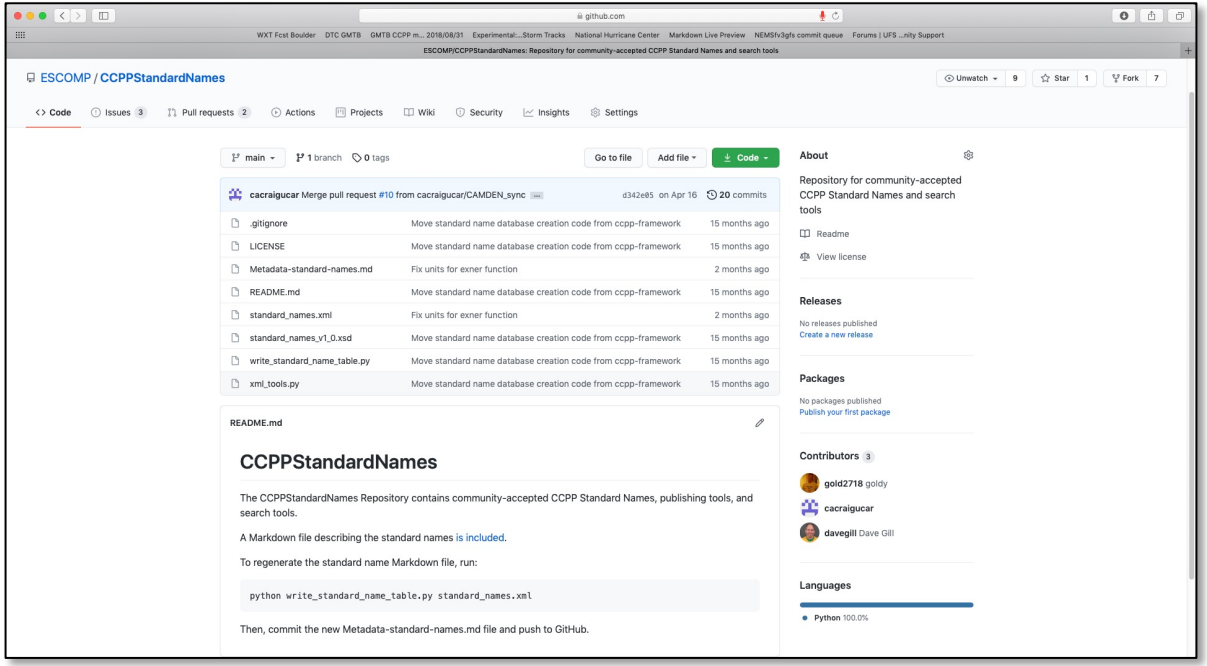
CCPP error handling

- Schemes should make use of CCPP error-handling variables and not stop/abort/print errors within
- `ccpp_error_flag` and `ccpp_error_message` must be arguments (intent OUT)
- In the event of an error, assign a meaningful error message to **errmsg** and set **errflg** to a value other than 0:

```
[errmsg]
  standard_name = ccpp_error_message
  long_name = error message for error
…
  units = none
  dimensions = ()
  type = character
  kind = len=*
  intent = out
  optional = F
[errflg]
  standard_name = ccpp_error_flag
  long_name = error flag for error …
  units = flag
  dimensions = ()
  type = integer
  intent = out
  optional = F
```

```
write (errmsg, '(*(a))') 'Logic error in scheme xyz: …'
errflg = 1
return
```

Progress towards a cross-lab standard name resource



- Active discussion on contents and ways to utilize the repository
- First set of updated standard names and rules for creating new names have been published

https://github.com/ESCOMP/CCPPStandardNames

CCPP inline scientific/technical documentation

- Uses Doxygen inline markup
- Additive to existing source code documentation
- Metadata table is parsed into HTML to be included on generated documentation website
- Includes information about scheme provenance, scientific papers, figures, code layout, and scheme algorithm

https://dtcenter.ucar.edu/GMTB/v5.0.0/sci_doc/index.html

https://ccpp-techdoc.readthedocs.io/en/v5.0.0/

CCPP coding miscellany

- All external information required by the scheme must be passed in via the argument list.
  - No 'use EXTERNAL_MODULE' for passing in data
  - Physical constants should go through the argument list
- Code must comply to modern Fortran standards (Fortran 90/95/2003/2008).
- Use labeled **end** statements for modules, subroutines and functions, example:
  - **module scheme_template → end module scheme_template**.
- Use **implicit none**.
- All **intent(out)** variables must be set inside the subroutine, including the mandatory variables **errflg** and **errmsg**. [Watch out for partially set **intent(out)** variables.]
- No permanent state of decomposition-dependent host model data inside the module, i.e. no variables that contain domain-dependent data using the **save** attribute.
- No **goto** statements.
- No **common** blocks.

Additional coding rules are listed under the *Coding Standards* section of the NOAA NGGPS Overarching System team document on Code, Data, and Documentation Management for NEMS Modeling Applications and Suites (available at https://docs.google.com/document/u/1/d/1bjnyJpJ7T3XeW3zCnhRLTL5a3m4_3XIAUeThUPWD9Tg/edit#heading=h.97v79689onyd).

How can a host use the CCPP?

- See Chapter 6 in the CCPP Documentation:
  - https://ccpp-techdoc.readthedocs.io/en/v5.0.0/HostSideCoding.html
- Host metadata (which variables it can provide to physics)
- Calls within code
- Parallelism
- CCPP at build-time
  - Multi-suite compilation
  - What is produced?

CCPP Host metadata

- Most of the host metadata is in `FV3/ccpp/data/GFS_typedefs.meta`
- Other files also have metadata to help define DDTs or provide other variables to the physics (e.g. `machine.F`)
- Differences compared to scheme metadata:
  - Uses `type = DDT or module`
  - Optional and intent metadata attributes are not used
  - Variables can have `active` attribute:
    - `active = logical expression`
    - Since host models may conditionally allocate memory, the logical expression uses CCPP standard names and represents when the given variable is allocated for use in physics:
      - e.g., `active = (flag_diagnostics_3D)`

Parallelism using the CCPP

Overarching paradigms

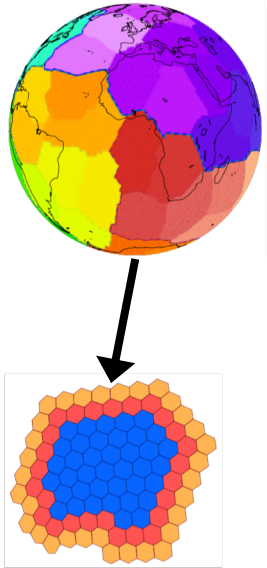
- Physics are column-based, no communication during time integration in physics
- Physics initialization/finalization are independent of threading strategy of the model

MPI

- MPI communication only allowed in the physics initialization/finalization
- Use MPI communicator provided by host model, not `MPI_COMM_WORLD`

OpenMP

- Time integration (but not init./final.) can be called by multiple threads
- Threading inside physics is allowed, use # OpenMP threads provided by host model



CCPP @ build time

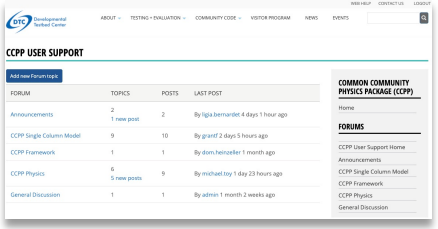
- A Python script is the "workhorse" of the CCPP framework and is called at build-time
- The script is given a set of SDFs representing the suites to be compiled and those available to use at run-time
  - Reads all scheme metadata for each given suite
  - Reads all host metadata
  - Matches **variables provided** with **variables requested**
  - Autogenerates suite and group caps
  - Autogenerates `ccpp_static_api.F90`
  - Autogenerates makefile information for compiling physics and caps within host's build system

Future Direction

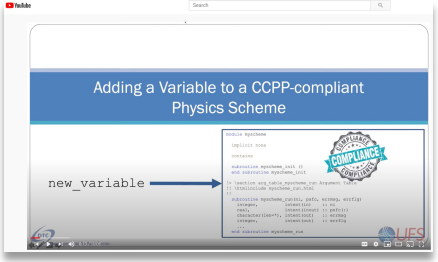
- Continue to expand contributions and partner with other organizations
- CCPP-physics
  - Continue adding and improving existing schemes to improve UFS applications (e.g. chemistry schemes from NOAA GSL)
- CCPP-framework
  - Transition to new cap generation software (capgen.py; in coordination with NCAR)
  - Usability improvements (e.g. in-suite variable tracking)
  - NUOPC interface for CCPP suites (unfunded)

Other CCPP support/training resources

- Forums
  - https://dtcenter.org/forum/ccpp-user-support
  - https://forums.ufscommunity.org/



- YouTube
  - Developmental Testbed Center Channel
  - CCPP playlist
  - https://www.youtube.com/watch?v=ut1mfK5K84w&list=PLFqIc1m9FLQxCpogp6x_KQMYvY0BBqY2c



- CCPP Technical Documentation
  - https://ccpp-techdoc.readthedocs.io/en/v5.0.0/
- CCPP Physics Scientific Docs
  - https://dtcenter.ucar.edu/GMTB/v5.0.0/sci_doc/index.html