CCPP Technical Overview

Grant Firl
Cooperative Institute for Research in the Atmosphere, NOAA Global Systems Laboratory, Developmental Testbed Center

August 15, 2023 – CCPP Visioning Workshop

Developmental Testbed Center

Outline

1. CCPP Within the Model System
2. “CCPP-compliant” Physics
3. Assembling Physics Suites
4. Host-side Coding and the CCPP API
5. Framework Scripts and Building
6. Miscellanea

CCPP Within the Model System

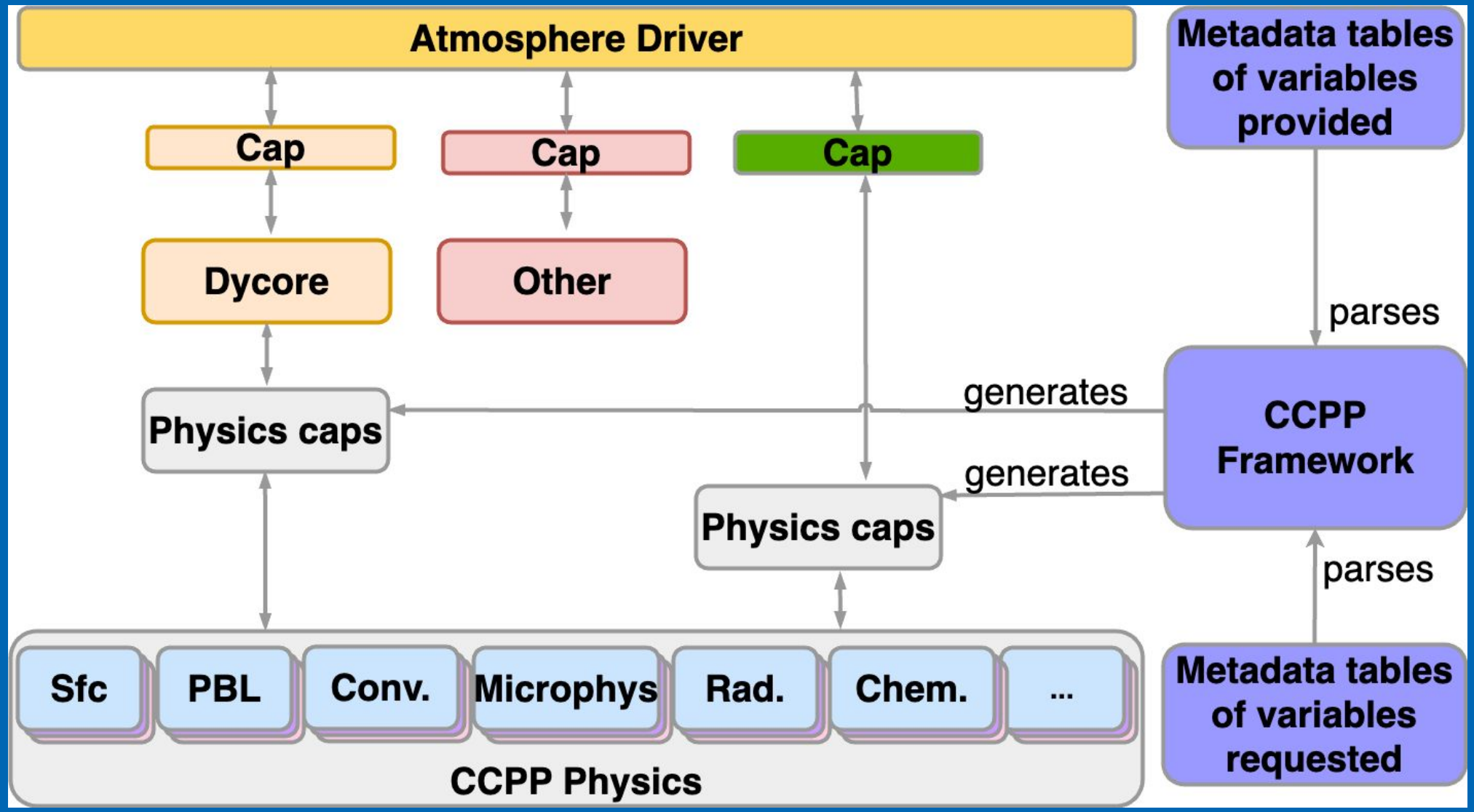
Model **without** CCPP

- Physics Schemes
  - Host-specific interface
- Physics Driver
  - Manually coded and host-specific
- Suite Control
  - NML and IF statements
- Memory Allocation
  - Spread between host, driver, schemes as necessary
- Documentation
  - Variable?
- Build System
  - “Normal”

Model **with** CCPP

- Physics Schemes
  - Standardized interface
    - 5 phases possible
    - Arguments described and documented
- Physics Driver
  - Autogenerated for given suites
- Suite Control
  - External XML files
- Memory Allocation
  - Host-only
- Documentation
  - Standardized, in-line
- Build System
  - Includes “hand-shake” step and code autogeneration

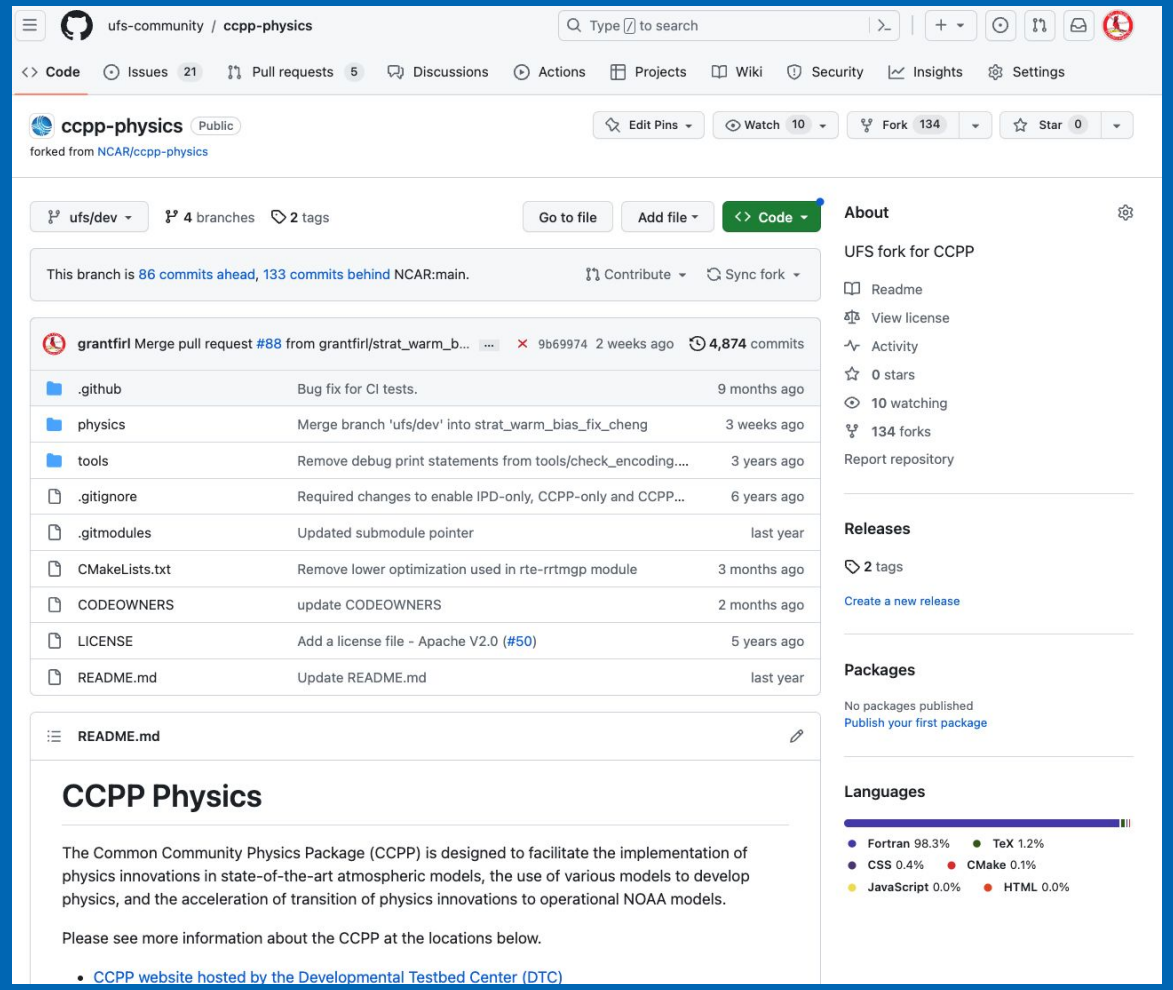
CCPP Within the Model System



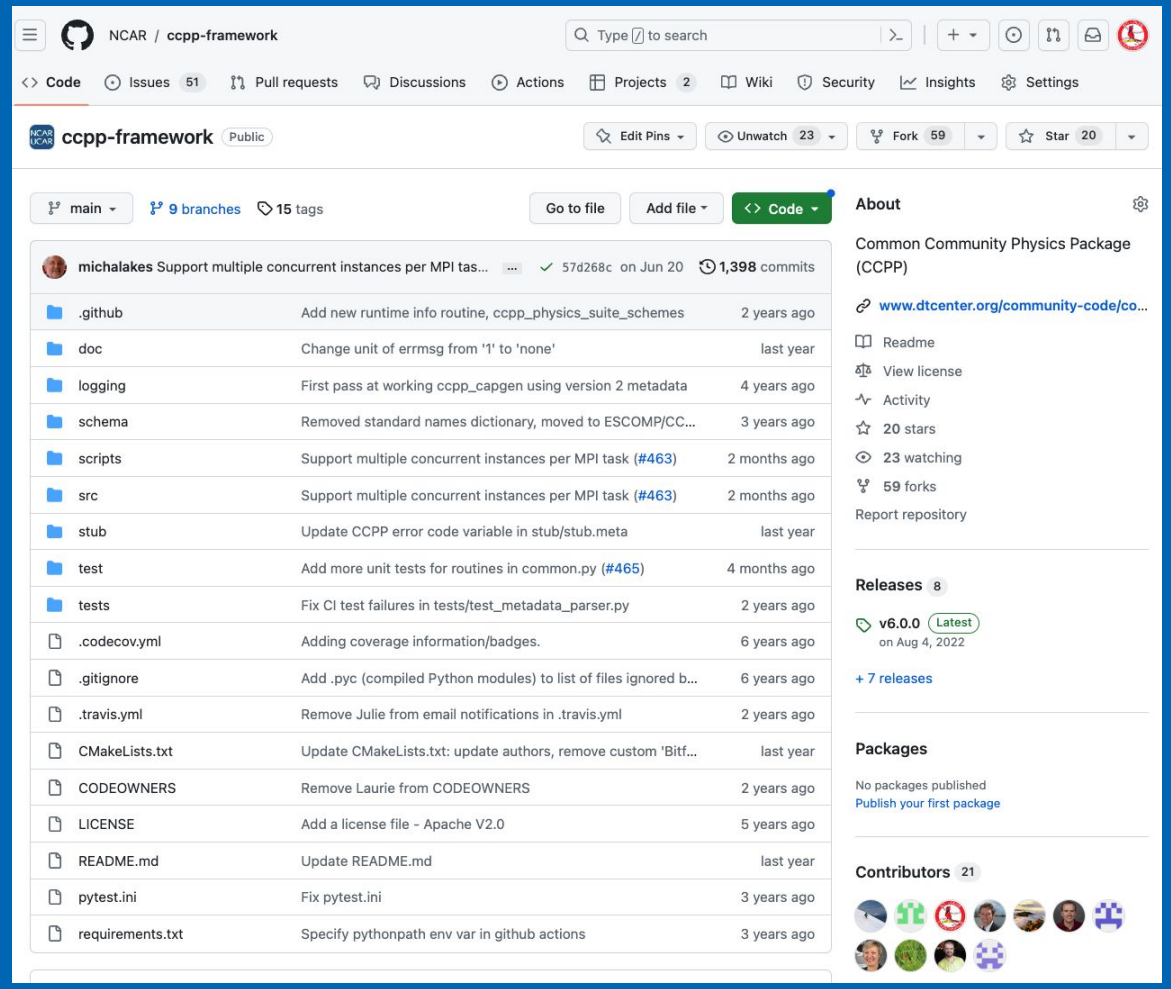
CCPP Within the Model System

- Host repository
  - …
    - ccpp
      - **ccpp-physics**
      - ccpp-framework

- Contains all physics code
  - “entry points”
  - Dependencies
  - Metadata



CCPP Within the Model System



- Host repository
  - …
    - ccpp
      - ccpp-physics
      - **ccpp-framework**

- Contains Python scripts that autogenerates physics drivers
  - Needs configuration file (stored in host repository) that points to information about host and physics schemes
  - Autogenerates information needed by host's build system to compile physics and autogenerated drivers

“CCPP-compliant” Physics

Structure (File 1 of 2)

- FORTRAN module (F90+)
- Filename same as module
- Contains at least 1 subroutine corresponding to 1 of 5 phases
  1. `*_init`
  2. `*_timestep_init`
  3. `*_run`
  4. `*_timestep_finalize`
  5. `*_finalize`
  - `*_init and *_finalize` must have the same result when called more than once
    - Can make use of `is_initialized` flag
- Contains Doxygen “hook” prior to subroutine definition
  -
```
!> \section arg_table_X_run Argument Table
!! \htmlinclude X_run.html
!!
```

“CCPP-compliant” Physics

Structure (File 1 of 2)

scheme_template.F90

```fortran
module scheme_template

  contains

!> \section arg_table_scheme_template_run Argument Table
!! \htmlinclude scheme_template_run.html
!!
  subroutine scheme_template_run (errmsg, errflg)

    implicit none

    !--- arguments
    ! add your arguments here
    character(len=*), intent(out)   :: errmsg
    integer,          intent(out)   :: errflg

    !--- local variables
    ! add your local variables here

    continue

    !--- initialize CCPP error handling variables
    errmsg = ''
    errflg = 0

    !--- initialize intent(out) variables
    ! initialize all intent(out) variables here

    !--- actual code
    ! add your code here

    ! in case of errors, set errflg to a value != 0,
    ! assign a meaningful message to errmsg and return

    return

  end subroutine scheme_template_run

end module scheme_template
```

“CCPP-compliant” Physics

Structure (File 2 of 2)

- Needs accompanying `X.meta` file describing the scheme
- Required contents
  - `[ccpp-table-properties]`

    Applies to entire scheme

```
[ccpp-table-properties]
  name = X
  type = scheme
  dependencies = X_dependency1.F90, X_dependency2.F90
```

  - `[ccpp-arg-table]`

    Applies to one subroutine; can be more than one

```
[ccpp-arg-table]
  name = X_run
  type = scheme
 [errmsg]
  standard_name = ccpp_error_message
  long_name = error message for error handling in CCPP
  units = none
  dimensions = ()
  type = character
  kind = len=*
  intent = out
```

“CCPP-compliant” Physics

Scheme Coding Rules/Concepts (1)

- Pass **all** data needed by the scheme through the argument list
  - Don’t put `use external_module` to pass data
- Use assumed-shape array declarations for argument variables

```
real(kind=kind_phys), dimension(:,:), intent(inout) :: foo
real(kind=kind_phys), dimension(its:,kts:), intent(inout) :: foo
```

Not

```
real(kind=kind_phys), dimension(ni,nk), intent(inout) :: foo
```

- This allows the compiler to perform bounds checking and detect errors that otherwise may go unnoticed.
- This also avoids segmentation faults for variables that may be conditionally-allocated in the host.

“CCPP-compliant” Physics

Scheme Coding Rules/Concepts (2)

- Pass physical constants through the argument list using either of these methods:
  1. Direct: pass in physical constants via the argument list and propagate them down to any subroutines that need them
  2. Scheme-level module:
     1. Pass the physical constants once through the argument list for the top-level `*_init` subroutine for the scheme. This top-level `_init` subroutine also imports scheme-specific constants from the scheme-level module.
     2. Set the scheme-level module constants from the those passed in from the host model via the argument list.
     3. Import constants where they are needed in the scheme from the scheme-level module.

- Use of the *physcons* module (ccpp-physics/physics/physcons.F90) is **not recommended**, since it is specific to FV3 and will be removed in the future.

“CCPP-compliant” Physics

Scheme Coding Rules/Concepts (3)

- Labeled end statements should be used for modules, subroutines, functions, and type definitions;
  e.g. `module scheme_template → end module scheme_template`
- Variables that contain domain-dependent data cannot be kept using the `save` attribute
- Schemes are not allowed to abort/stop execution
- All intent(out) variables must be set inside the subroutine
- The `implicit none` statement is mandatory and is preferable at the module-level so that it applies to all the subroutines in the module.
- Schemes are not allowed to perform I/O operations except for reading lookup tables or other information needed to initialize the scheme, including `stdout` and `stderr`. Diagnostic messages are tolerated, but should be minimal.

“CCPP-compliant” Physics

Scheme Coding Rules/Concepts (4)

- Errors are handled by the host model using the two mandatory arguments `errmsg` and `errflg`. In the event of an error, a meaningful error message should be assigned to `errmsg` and `errflg` set to a value other than 0. For example:

```
errmsg = 'Logic error in scheme xyz: ...'
errflg = 1
return
```

- Code must comply to modern Fortran standards (Fortran 90 or newer), where possible.
- Uppercase file endings (`.F`, `.F90`) are preferred to enable preprocessing by default.
- The use of `goto` statements is discouraged.
- `common` blocks are not allowed.

“CCPP-compliant” Physics

Metadata (1)

Recall that there are will be (at least) 2 sections of each metadata file.

1. `[ccpp-table-properties]`
   - type, e.g. (scheme, module, ddt)
   - name
     - If type is module or ddt, name must match single associated ccpp-arg-table name
     - Otherwise, use the “root” scheme name
   - dependencies
     - Comma-separated list of files that the scheme depends on (to be compiled first)
     - Full relative path from scheme’s location
   - relative_path
     - Can be used in conjunction with the dependencies list – relative_path gets prepended to all files

```
[ccpp-table-properties]
      name = X
      type = scheme
      dependencies = X_dependency1.F90, …

[ccpp-arg-table]
      name = X_run
      type = scheme
    [errmsg]
      standard_name = ccpp_error_message
      long_name = error message for …
      units = none
      dimensions = ()
      type = character
      kind = len=*
      intent = out
    [errflg]
      …
```

“CCPP-compliant” Physics

Metadata (2)

Recall that there are will be (at least) 2 sections of each metadata file.

2. `[ccpp-arg-table]`
   - type, e.g. (scheme, module, ddt)
   - name
     - If type is module or ddt, name must match ccpp-table-properties
     - Otherwise, use the subroutine name (e.g. X_init, X_run, …)
   - every argument is listed with the following attributes:
   - [local_name]
     - standard_name
     - long_name
     - units
     - dimensions
     - type
     - kind
     - intent

```
[ccpp-table-properties]
      name = X
      type = scheme
      dependencies = X_dependency1.F90, …

[ccpp-arg-table]
      name = X_run
      type = scheme
    [errmsg]
      standard_name = ccpp_error_message
      long_name = error message for …
      units = none
      dimensions = ()
      type = character
      kind = len=*
      intent = out
    [errflg]
      …
```

“CCPP-compliant” Physics

Metadata (3)

- [local_name]
  - What the variable is called in the code
  - Doesn’t have to match across schemes/hosts
- standard_name
  - Used as a variable’s “key”
  - Uniquely identifies a variable for all CCPP-compliant hosts and schemes
  - Extension of the CF Standard Names
  - There is a repository of standard names:
    - https://github.com/ESCOMP/CCPPStandardNames
    - Contains list of names and **rules for generating new names**
    - Currently, there is no checking between the standard name repo and what is used in ccpp-physics or host models
  - Reducing name ambiguity is more important than name length
  - Needs:
    - Search tool
    - Cross-checking
    - Consolidation

```
[ccpp-table-properties]
      name = X
      type = scheme
      dependencies = X_dependency1.F90, …

[ccpp-arg-table]
      name = X_run
      type = scheme
    [errmsg]
      standard_name = ccpp_error_message
      long_name = error message for …
      units = none
      dimensions = ()
      type = character
      kind = len=*
      intent = out
    [errflg]
      …
```

“CCPP-compliant” Physics

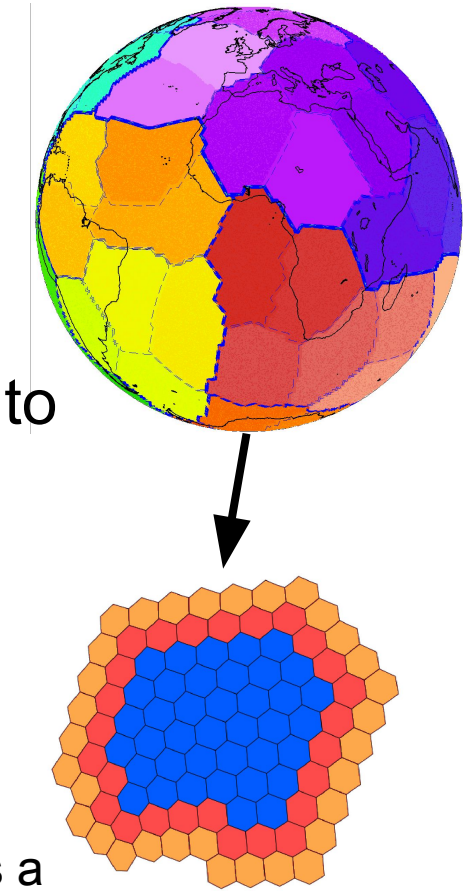
Metadata (4)

- long_name
  - more descriptive name (if necessary)
- units
  - aX bY-Z format (e.g. m2 s-2)
  - Automatic conversion possible
- dimensions
  - () = scalar
  - (ccpp_constant_one:horizontal_loop_extent)
  - (standard_name_of_dimension)
  - (standard_dim1, standard_dim2)
  - Implied 1 if no dimension start supplied
- type
  - e.g. integer, real, character, DDT
- kind
  - real precision or character length
- intent
  - in, in/out, out
  - scheme metadata only

```
[ccpp-table-properties]
      name = X
      type = scheme
      dependencies = X_dependency1.F90, …

[ccpp-arg-table]
      name = X_run
      type = scheme
    [errmsg]
      standard_name = ccpp_error_message
      long_name = error message for …
      units = none
      dimensions = ()
      type = character
      kind = len=*
      intent = out
    [errflg]
      …
```

“CCPP-compliant” Physics

Domain Decomposition and Parallelization (1)



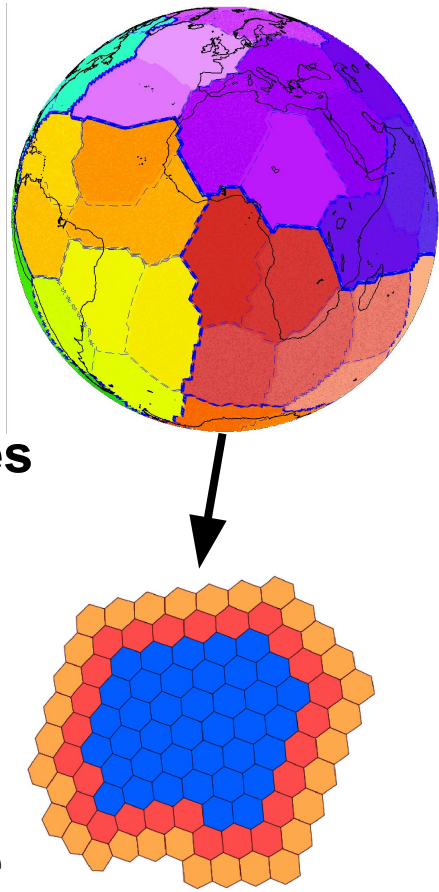
- Hosts with large horizontal domains may decompose the domain into smaller chunks for parallel processing, often in multiple different ways.

  1. Divide entire domain into smaller subdomains for each MPI process
  2. Call physics on some subset of the MPI subdomain at a time

- The horizontal dimension referring to the size of the current MPI process subdomain has a standard name of **`horizontal_dimension`**
  - The `init`, `timestep_init`, `timestep_finalize`, and `finalize` phases have access to the entire MPI subdomain so variables with a horizontal dimension should use this standard name during these phase.
- The horizontal dimension referring to the computational “block/chunk” size in the horizontal dimension has the standard name of **`horizontal_loop_extent`**.
  - The `run` phase only has access to this (potentially) smaller sub-subdomain so this standard name is used within.

“CCPP-compliant” Physics

Domain Decomposition and Parallelization (2)



- Most often, shared memory (OpenMP: Open Multi-Processing) and distributed memory (MPI: Message Passing Interface) communication is done **outside** the physics, in which case the **loops and arrays already take into account the sizes of the threaded tasks through their input indices and array dimensions.**
- Further parallelization within physics schemes must follow certain rules:
  1. See previous slide RE: which CCPP phases expect entire MPI subdomains.
  2. The `run` phase may be further threaded, making use of smaller horizontal blocks.
  3. openMP threading is allowed within schemes, but must use the passed-in number of threads.
  4. MPI communication is allowed in the *init*, *timestep_init*, *timestep_finalize*, and *finalize* phases for the purpose of computing, reading or writing scheme-specific data that is independent of the host model’s data decomposition.
  5. If MPI is used, it is restricted to global communications: barrier, broadcast, gather, scatter, reduction. Point-to-point communication is not allowed. Use MPI communicator provided by host model, not `MPI_COMM_WORLD`.
  6. Calls to MPI and OpenMP functions, and the import of the MPI and OpenMP libraries, must be guarded by C preprocessor directives.

“CCPP-compliant” Physics

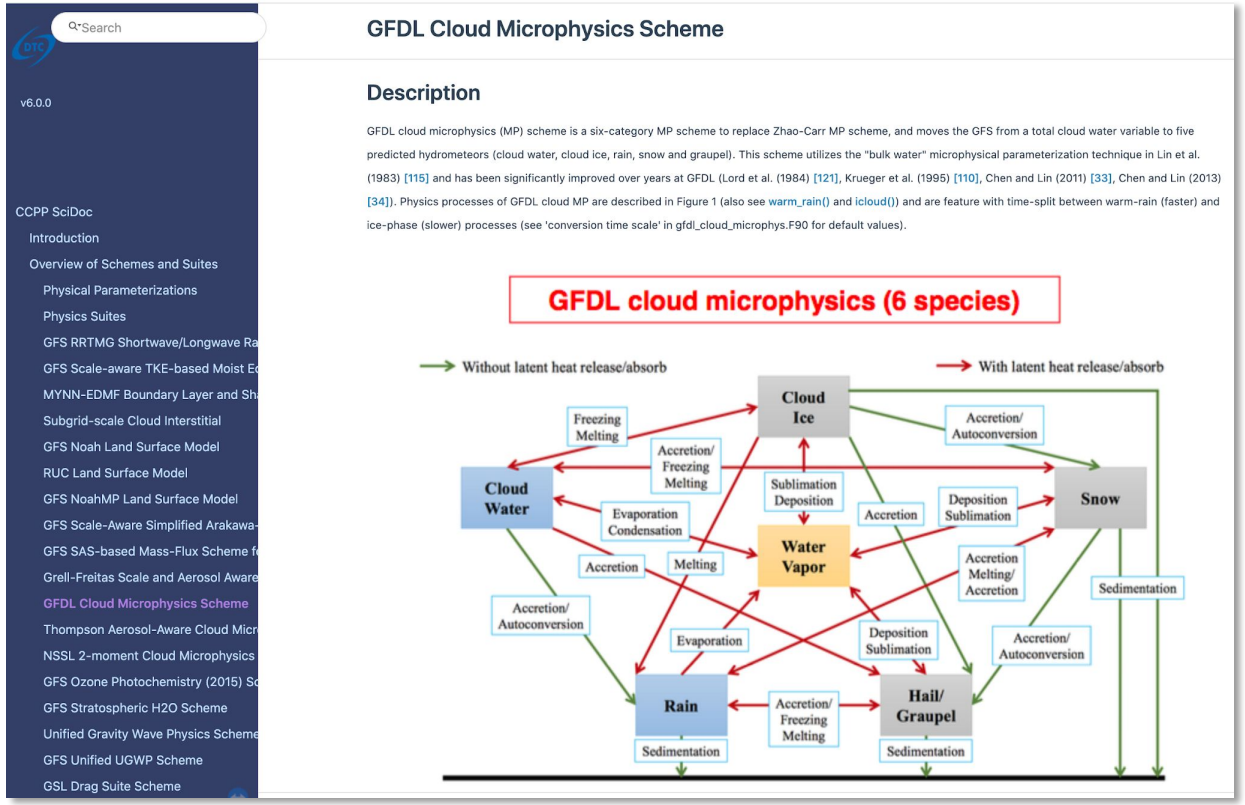
Repository Acceptance

- Compliance to the “mechanical” rules is the first step for a new scheme to be accepted into the CCPP-physics authoritative repository
  - Once any new variables are added to the host, this should allow a scheme to “function”, but considerable work must be done to verify that the scheme behaves as expected within an entire suite of physics.
- The CCPP Physics Management Committee, comprising individuals from multiple institutions should weigh in on the inclusion of new schemes.
- Non-authoritative forks of ccpp-physics may have their own rules for new scheme inclusion.
- CODEOWNERS

“CCPP-compliant” Physics

Scientific Documentation Considerations

- CCPP schemes use in-line Doxygen comments to generate the scientific documentation that is posted on the web.
- See https://ccpp-techdoc.readthedocs.io/en/latest/CompliantPhysicsParams.html#scientific-documentation-rules for details for documentation generation.
- Documentation should *ideally* be updated and pushed to the authoritative repository whenever changes are made to scientific algorithms.



Pause for Questions…

Assembling Physics Suites

Suite Definition File (SDF)

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
    - Group
      - Subcycle
        - Scheme

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

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

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

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

Assembling Physics Suites

Suite Definition File (SDF)

```xml
<?xml version="1.0" encoding="UTF-8"?>

<suite name="FV3_GFS_v16" version="1">
  <group name="fast_physics">
    <subcycle loop="1">
      <scheme>fv_sat_adj</scheme>
    </subcycle>
  </group>
  <group name="time_vary">
    <subcycle loop="1">
      <scheme>GFS_time_vary_pre</scheme>
      <scheme>GFS_rrtmg_setup</scheme>
      <scheme>GFS_rad_time_vary</scheme>
      <scheme>GFS_phys_time_vary</scheme>
    </subcycle>
  </group>
  <group name="radiation">
    <subcycle loop="1">
      <scheme>GFS_suite_interstitial_rad_reset</scheme>
      <scheme>GFS_rrtmg_pre</scheme>
      <scheme>GFS_radiation_surface</scheme>
      <scheme>rrtmg_sw_pre</scheme>
      <scheme>rrtmg_sw</scheme>
      <scheme>rrtmg_sw_post</scheme>
      <scheme>rrtmg_lw_pre</scheme>
      <scheme>rrtmg_lw</scheme>
      <scheme>rrtmg_lw_post</scheme>
      <scheme>GFS_rrtmg_post</scheme>
    </subcycle>
  </group>
  ...
```

```xml
<group name="physics">
    <subcycle loop="1">
      <scheme>GFS_suite_interstitial_phys_reset</scheme>
      <scheme>GFS_suite_stateout_reset</scheme>
      <scheme>get_prs_fv3</scheme>
      <scheme>GFS_suite_interstitial_1</scheme>
      <scheme>GFS_surface_generic_pre</scheme>
      <scheme>GFS_surface_composites_pre</scheme>
      <scheme>dcyc2t3</scheme>
      <scheme>GFS_surface_composites_inter</scheme>
      <scheme>GFS_suite_interstitial_2</scheme>
    </subcycle>
    <!-- Surface iteration loop -->
    <subcycle loop="2">
      <scheme>sfc_diff</scheme>
      <scheme>GFS_surface_loop_control_part1</scheme>
      <scheme>sfc_nst_pre</scheme>
      <scheme>sfc_nst</scheme>
      <scheme>sfc_nst_post</scheme>
      <scheme>lsm_noah</scheme>
      <scheme>sfc_sice</scheme>
      <scheme>GFS_surface_loop_control_part2</scheme>
    </subcycle>
    <!-- End of surface iteration loop -->
    <subcycle loop="1">
      <scheme>GFS_surface_composites_post</scheme>
      <scheme>sfc_diag</scheme>
      <scheme>sfc_diag_post</scheme>
      <scheme>GFS_surface_generic_post</scheme>
      <scheme>GFS_PBL_generic_pre</scheme>
      <scheme>satmedmfvdifq</scheme>
```

```xml
<scheme>GFS_PBL_generic_post</scheme>
      <scheme>GFS_GWD_generic_pre</scheme>
      <scheme>cires_ugwp</scheme>
      <scheme>cires_ugwp_post</scheme>
      <scheme>GFS_GWD_generic_post</scheme>
      <scheme>GFS_suite_stateout_update</scheme>
      <scheme>ozphys_2015</scheme>
      <scheme>h2ophys</scheme>
      <scheme>get_phi_fv3</scheme>
      <scheme>GFS_suite_interstitial_3</scheme>
      <scheme>GFS_DCNV_generic_pre</scheme>
      <scheme>samfdeepcnv</scheme>
      <scheme>GFS_DCNV_generic_post</scheme>
      <scheme>GFS_SCNV_generic_pre</scheme>
      <scheme>samfshalcnv</scheme>
      <scheme>GFS_SCNV_generic_post</scheme>
      <scheme>GFS_suite_interstitial_4</scheme>
      <scheme>cnvc90</scheme>
      <scheme>GFS_MP_generic_pre</scheme>
      <scheme>gfdl_cloud_microphys</scheme>
      <scheme>GFS_MP_generic_post</scheme>
      <scheme>maximum_hourly_diagnostics</scheme>
    </subcycle>
  </group>
  <group name="stochastics">
    <subcycle loop="1">
      <scheme>GFS_stochastics</scheme>
      <scheme>phys_tend</scheme>
    </subcycle>
  </group>
</suite>
```

Assembling Physics Suites

Suite Definition File (SDF)

- SDFs are part of the host model repository
- Control is still “shared” with physics namelists
  - Physics code often still relies on logicals that denote whether a scheme is active; there must be a consistency check
- CCPP phases follow order of SDF too
- SDF groups allow any computation to happen in between
  - E.g. externally-coupled process in the middle of a physics suite, advanced time-stepping schemes
- Order is easily changeable, but one needs to understand repercussions, both numerically and code-wise (will inputs have values?)
- Schemes can be called more often via SDF subcycles or internally (e.g. Thompson MP)

Assembling Physics Suites

Primary vs. Interstitial Schemes

Schemes in the CCPP are NOT required to be categorized. However, it is useful to make the following distinction.

- Primary Scheme
  - A parameterization, such as PBL, microphysics, convection, and radiation, that fits the traditionally-accepted definition.
  - These often change the state variables in some way.
- 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.
  - This code is typically found in physics drivers in non-CCPP models, but it needs to exist as a “scheme” in the CCPP.

Assembling Physics Suites

Interstitial Scheme Organization

Original organizing principle (may not be valid for all hosts):

1. Scheme-specific: for code that is only needed for one specific scheme, but doesn't belong in the scheme itself (e.g. mp_thompson_pre)

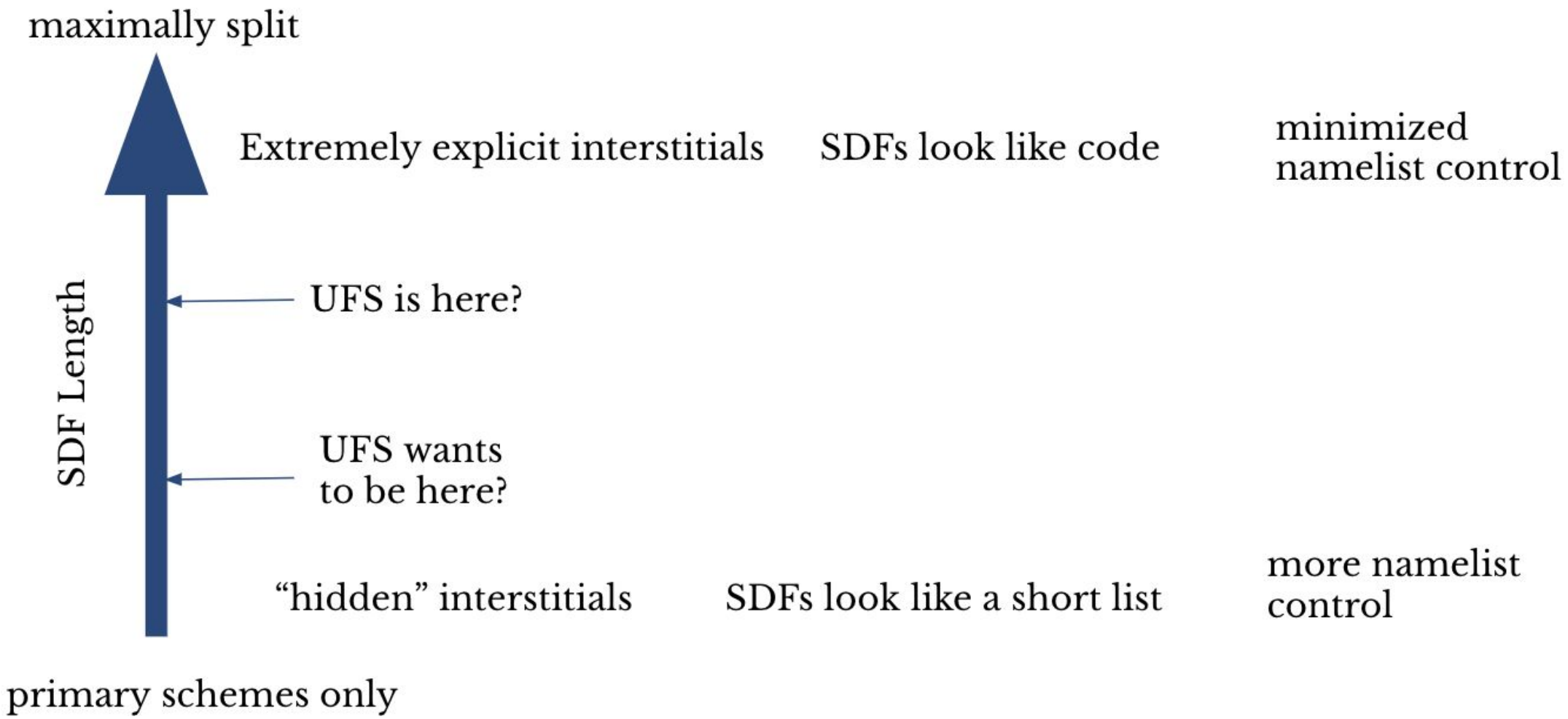
2. Scheme-generic: for code that is needed for all schemes in a group/class (e.g. GFS_MP_generic_pre)

3. Suite-level: for code that is applicable to one or more scheme groups (e.g. GFS_suite_interstitial_4)

Goals: Primary scheme interoperability, suite configurability, future maintainability, strict reproducibility

Assembling Physics Suites

Interstitial Scheme Organization



Host-side Coding and the CCPP API

Host-side Metadata (1)

- Metadata is needed on the host side in order to describe what data is available for the physics to use.
  - Should have a metadata file for every host file that allocates memory used by physics
  - No restrictions on hosts using DDTs to store data
  - OK to have DDT definition and declaration in same module

```
module example_vardefs

  implicit none

!!> \section arg_table_example_vardefs
!! \htmlinclude example_vardefs.html
!!

  integer, parameter            :: r15 = selected_real_kind
  integer                       :: ex_int
  real(kind=8), dimension(:,:)  :: ex_real1
  character(len=64)             :: errmsg
  logical                       :: errflg

!!> \section arg_table_example_ddt
!! \htmlinclude example_ddt.html
!!

  type ex_ddt
    logical                :: l
    real, dimension(:,:)   :: r
  end type ex_ddt

  type(ex_ddt) :: ext

end module example_vardefs
```

Host-side Coding and the CCPP API

Host-side Metadata (2)

- Example of host module where variables are declared and DDTs defined
  - Notice:
    - Type = module
    - Intrinsic types declared here
    - DDT type definition has metadata
    - DDT instance has metadata

```
[ccpp-table-properties]
  name = arg_table_example_vardefs
  type = module

[ccpp-arg-table]
  name = arg_table_example_vardefs
  type = module
[ex_int]
  standard_name = example_int
  long_name = ex. int
  units = none
  dimensions = ()
  type = integer
[ex_real]
  standard_name = example_real
  long_name = ex. real
  units = m
  dimensions = (horizontal_loop_extent,vertical_layer_dimension)
  type = real
  kind = kind=8
[ex_ddt]
  standard_name = example_ddt
  long_name = ex. ddt
  units = DDT
  dimensions = ()
  type = ex_ddt
[ext]
  standard_name = example_ddt_instance
  long_name = ex. ddt inst
  units = DDT
  dimensions = ()
  type = ex_ddt
```

Host-side Coding and the CCPP API

Host-side Metadata (3)

- Example of DDT definition metadata
  - Notice:
    - Local name is as the variable is referenced in module (DDT_instance%component)
    - Use `horizontal_loop_extent` for horizontal dimension
    - No intent attribute
    - `Active` attribute
      - Expression using standard names for when a variable is conditionally allocated or available
      - True by default if omitted
    - With the CCPP, it is possible to not only refer to components of DDTs, but also to slices of arrays with provided metadata as long as these are contiguous in memory

```
########################################################################
[ccpp-table-properties]
  name = arg_table_example_ddt
  type = ddt

[ccpp-arg-table]
  name = arg_table_example_ddt
  type = ddt
[ext%1]
  standard_name = example_flag
  long_name = ex. flag
  units = flag
  dimensions =
  type = logical
[ext%r]
  standard_name = example_real3
  long_name = ex. real
  units = kg
  dimensions = (horizontal_loop_extent,vertical_layer_dimension)
  type = real
  kind = r15
[ext%r(;,1)]
  standard_name = example_slice
  long_name = ex. slice
  units = kg
  dimensions = (horizontal_loop_extent,vertical_layer_dimension)
  type = real
  kind = r15
[nwfa2d]
  standard_name = tendency_of_water_friendly_aerosols_at_surface
  long_name = instantaneous water-friendly sfc aerosol source
  units = kg-1 s-1
  dimensions = (horizontal_loop_extent)
  type = real
  kind = kind_phys
  active = (flag_for_microphysics_scheme == flag_for_thompson_microphysics_scheme .and. flag_for_aerosol_physics
```

Host-side Coding and the CCPP API

CCPP API (1)

The CCPP API is autogenerated at build-time for the given suites. It consists of 5 methods and a few utility variables.

Methods

1. `ccpp_physics_init(cdata, suite_name, [group_name], ierr=ierr)`
   - Calls init phase of given SDF group or entire suite (once per model run)
   - E.g. reading lookup tables, reading input datasets, computing derived quantities, broadcasting information to all MPI ranks, etc
   - For entire domain (access to all data an MPI task owns)
2. `ccpp_physics_finalize(cdata, suite_name, [group_name], ierr=ierr)`
   - Calls finalize phase of given SDF group or entire suite (once per model run)
   - E.g. deallocating variables, resetting flags from *initialized* to *non-initialized*, etc
   - For entire domain (access to all data an MPI task owns)

Host-side Coding and the CCPP API

CCPP API (2)

The CCPP API is autogenerated at build-time for the given suites. It consists of 5 methods and a few utility variables.

Methods

3. `ccpp_physics_timestep_init(cdata, suite_name, [group_name], ierr=ierr)`
   - Calls timestep_init phase of given SDF group or entire suite (once per physics timestep)
   - E.g. updating quantities that depend on the valid time, for example solar insolation angle, aerosol emission rates and other values obtained from climatologies
   - For entire domain (access to all data an MPI task owns)
4. `ccpp_physics_timestep_finalize(cdata, suite_name, [group_name], ierr=ierr)`
   - Calls timestep_finalize phase of given SDF group or entire suite (once per physics timestep)
   - For entire domain (access to all data an MPI task owns)

Host-side Coding and the CCPP API

CCPP API (3)

The CCPP API is autogenerated at build-time for the given suites. It consists of 5 methods and a few utility variables.

Methods

5. `ccpp_physics_run(cdata, suite_name, [group_name], ierr=ierr)`
   - Calls run phase of given SDF group or entire suite (called during integration time loop)
   - For each chunk/block (can be different than all horizontal points owned by MPI task)

Variables

- Error code for handling in CCPP (`errmsg`)
- Error message associated with the error code (`errflg`)
- Loop counter for subcycling loops (`loop_cnt`)
- Loop extent for subcycling loops (`loop_max`)
- Number of block for explicit data blocking in CCPP (`blk_no`)

Host-side Coding and the CCPP API

Preparing to use the CCPP API

Prior to using the CCPP API, the host model needs to declare and initialize a variable of `ccpp_t` (often referred to as `cdata`).

```
use ccpp_types,      only: ccpp_t
type(ccpp_t) :: cdata
cdata%blk_no = 1
cdata%thrd_no = 1
```

Note: One can have an array of ccpp_t for each block/thread depending on the domain decomposition and threading strategy.

Deallocation of the ccpp_t can optionally be done at the end of the run.

Host-side Coding and the CCPP API

Examples

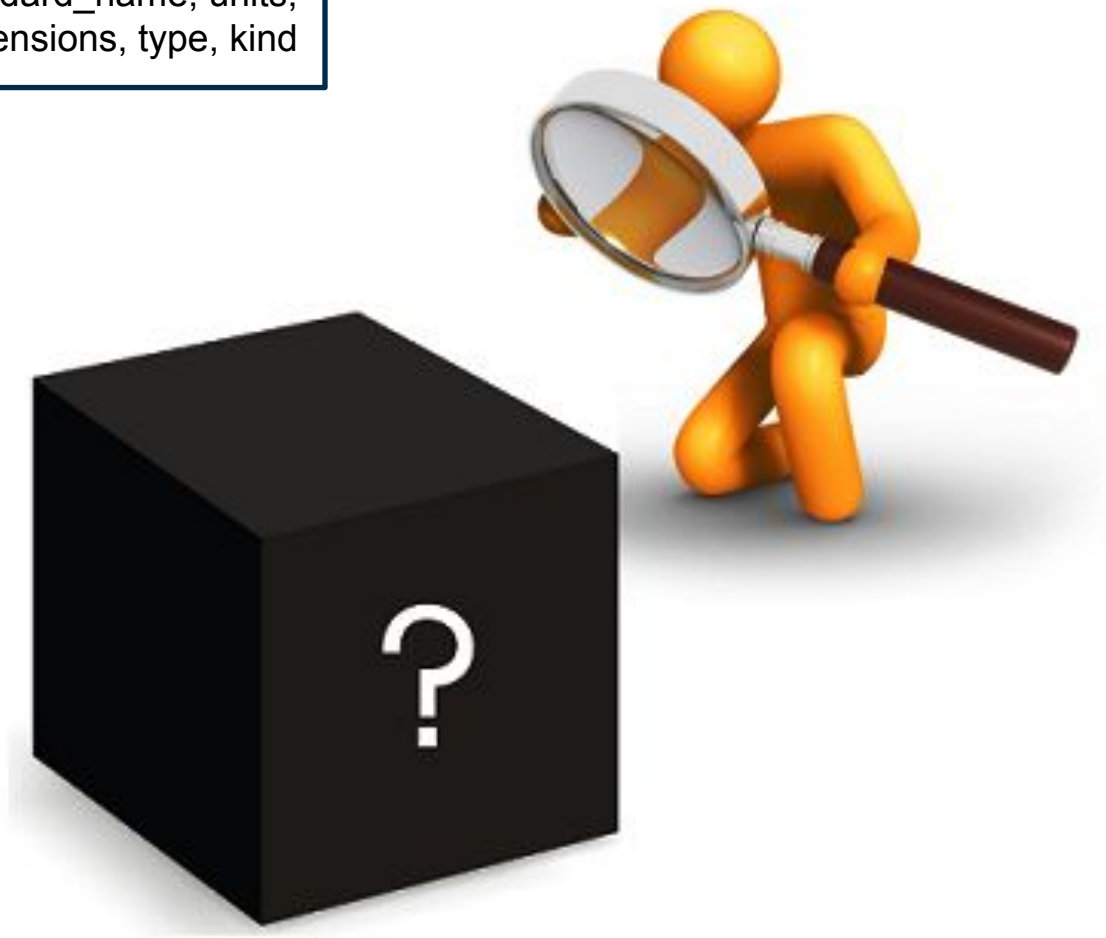
- The CCPP SCM interfaces directly with the CCPP API within its original source code.
  - Declares the `ccpp_t` with all other model data in `scm_type_defs.F90`
  - Initializes `ccpp_t` and calls all non-run phases of the CCPP in the ”main” section of `scm.F90`
  - Calls `ccpp_physics_run` for the entire suite at once from subroutines within `scm_time_integration.F90`, that is called during the main time loop
- The UFS adds an additional abstraction layer between the existing host code and the CCPP.
  - `CCPP_data.F90` contains the `ccpp_t` variables
  - `CCPP_driver.F90` does all interfacing with the CCPP API and initializes the `ccpp_t` variables
  - All phases are called from subroutines known by existing host code (using the correct domain decomposition for each phase) and error checking is performed after returning from the CCPP phases.

Framework Scripts and Building

ccpp_prebuild.py (1)

- The CCPP "ecosystem" relies on a set of python scripts in order to:
  1. Collect and **compare** information about data needed by the physics and supplied by the host
  2. Generate "caps" (AKA custom physics drivers) for a given set of suites that provides the data coupling and call sequences.
  3. Generate the API for the host to interact with
  4. Help the host's build system to compile the autogenerated code and physics

standard_name, units, dimensions, type, kind



Framework Scripts and Building

ccpp_prebuild.py (2)

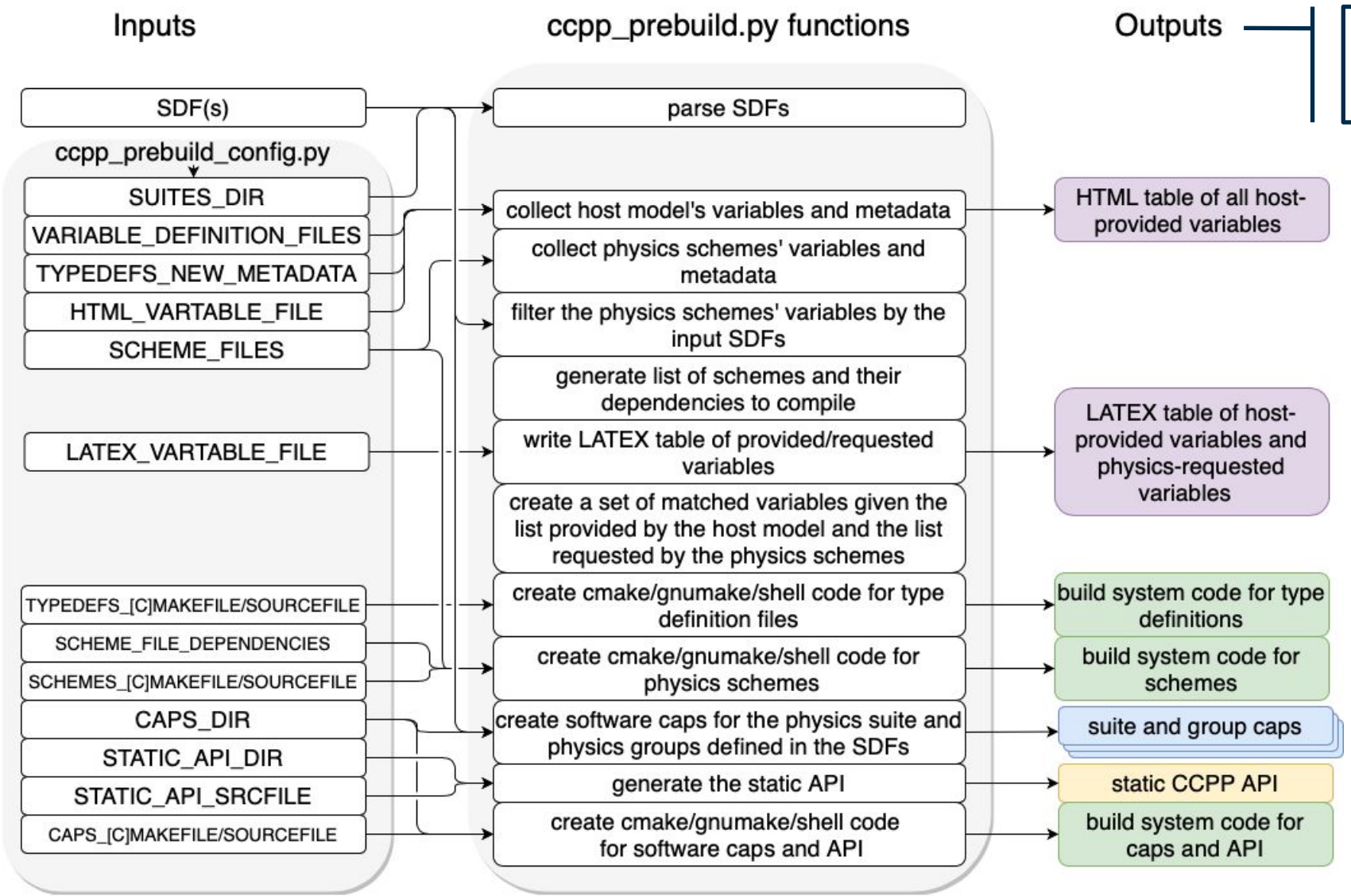
- Each host needs a configuration file to provide the main script with:
  - Path to host Fortran files that define what variables are available to the physics
  - Path to physics scheme files
  - Build path
  - Paths of where to put script outputs
  - Path to SDFs
  - Information for how host module and DDT variables are referenced in the code

ccpp_prebuild_config.py

```
# Host model identifier
HOST_MODEL_IDENTIFIER = "SCM"

# Add all files with metadata tables on the host model side,
# relative to basedir = top-level directory of host model
VARIABLE_DEFINITION_FILES = [
    'scm/src/scm_type_defs.F90',
    'scm/src/scm_physical_constants.F90',
    ]

 # How parent variables (module variables, derived data types)
 # are referenced in the model
 TYPEDEFS_NEW_METADATA = {
     'ccpp_types' : {
         'ccpp_types' : '',
         'ccpp_t' : 'cdata',
         },
     'GFS_typedefs' : {
         'GFS_typedefs' : '',
         'GFS_control_type' : 'physics%Model',
         },
     }

# Add all physics scheme files relative to basedir
SCHEME_FILES = {
    'ccpp/physics/physics/GFS_DCNV_generic.f90' ,
    'ccpp/physics/physics/sfc_sice.f',
    }

 # Default build dir, relative to current working directory,
 # if not specified as command-line argument
 DEFAULT_BUILD_DIR = 'scm/bin'
```

Framework Scripts and Building

ccpp_prebuild.py outputs



We will look at examples in another session.

Framework Scripts and Building

Using ccpp_prebuild.py

In practice, models have integrated the call to ccpp_prebuild.py in their build systems.

```
./ccpp/framework/scripts/ccpp_prebuild.py \
--config=./ccpp/config/ccpp_prebuild_config.py \
[--suites=suite1,suite2] \
[--verbose] \
[--clean] \
[--debug]
```

Where the script is called from varies by host

Internal name in the SDF (not filename)

Inserts additional checks on array sizes

INFO: CCPP prebuild step completed successfully.

Framework Scripts and Building

After compilation…