National Unified Operational Prediction Capability

NUOPC Layer Reference

ESMF 8.9.0 beta snapshot

Content Standards Committee (CSC) Members

January 17, 2025

NUOPC
CMA/CSC Committee
http://www.weather.gov/nuopc

Contents

1 Description **3**

2 Design and Implementation Notes **3**
2.1 Generic Components . . . 3
2.1.1 Component Specialization . . . 4
2.1.2 Partial Specialization . . . 6
2.2 Field Dictionary . . . 7
2.2.1 Field Dictionary file . . . 7
2.2.2 Preloaded Field Dictionary . . . 8
2.3 Metadata . . . 8
2.3.1 Driver Component Metadata . . . 8
2.3.2 Model Component Metadata . . . 10
2.3.3 Mediator Component Metadata . . . 13
2.3.4 Connector Component Metadata . . . 15
2.3.5 State Metadata . . . 18
2.3.6 Field Metadata . . . 18
2.4 Initialization . . . 21
2.4.1 Phase Maps, Semantic Specialization Labels, and Component Labels . . . 21
2.4.2 Field Pairing . . . 21
2.4.3 Namespaces . . . 22
2.4.4 Using Coupling Sets for Coupling Multiple Nests . . . 23
2.4.5 Connection Options . . . 24
2.4.6 Data-Dependencies during Initialize . . . 27
2.4.7 Transfer of Grid/Mesh/LocStream Objects between Components . . . 27
2.4.8 Field and Grid/Mesh/LocStream Reference Sharing . . . 28
2.4.9 Field Mirroring . . . 29
2.5 Timekeeping . . . 30
2.6 Component Hierarchies . . . 31
2.7 Resource Control and Threaded Components . . . 32
2.8 External NUOPC Interface . . . 37

3 API **40**
3.1 Generic Component: NUOPC_Driver . . . 40
3.2 Generic Component: NUOPC_ModelBase . . . 41
3.3 Generic Component: NUOPC_Model . . . 43
3.4 Generic Component: NUOPC_Mediator . . . 45
3.5 Generic Component: NUOPC_Connector . . . 47
3.6 General Generic Component Methods . . . 48
3.7 Field Dictionary Methods . . . 48
3.8 Free Format Methods . . . 48
3.9 Utility Routines . . . 48
3.10 Auxiliary Routines . . . 48

4 Standardized Component Dependencies **49**
4.1 Fortran components that are statically built into the executable . . . 50
4.2 Fortran components that are provided as shared libraries . . . 53
4.3 Components that are loaded during run-time as shared objects . . . 54
4.4 Components that depend on components . . . 55
4.5 Components written in C/C++ . . . 57

5 NUOPC Layer Compliance **60**
5.1 The Compliance Checker 60
5.2 The Component Explorer 62

6 Appendix A: Run Sequence Implementation **65**

7 Appendix B: Initialize Phase Definition Versions **66**
7.1 NUOPC_Driver IPD implementation 70
7.2 NUOPC_ModelBase IPD implementation 72
7.3 NUOPC_Model IPD implementation 73
7.3.1 Initialize Phase Specialization - label_SetClock 76
7.3.2 Initialize Phase Specialization - label_DataInitialize 76
7.3.3 Run Phase Specialization - label_SetRunClock 76
7.3.4 Run Phase Specialization - label_CheckImport 76
7.3.5 Run Phase Specialization - label_Advance 76
7.3.6 Run Phase Specialization - label_TimestampExport 77
7.3.7 Finalize Phase Specialization - label_Finalize 77
7.4 NUOPC_Mediator IPD implementation 77
7.5 NUOPC_Connector IPD implementation 78

1 Description

The NUOPC Layer is an add-on to the standard ESMF library. It consists of generic code of two different kinds: *utility routines* and *generic components*. The NUOPC Layer further implements a dictionary for standard field metadata.

The utility routines are subroutines and functions that package frequently used calling sequences of ESMF methods into single calls. Unlike the pure ESMF API, which is very class centric, the utility routines of the NUOPC Layer often implement tasks that involve several ESMF classes.

The generic components are provided in form of Fortran modules that implement GridComp and CplComp specific methods. Generic components are useful when implementing NUOPC compliant driver, model, mediator, or connector components. The provided generic components form a hierarchy that allows the developer to pick and choose the appropriate level of specification for a certain application. Depending on how specific the chosen level, generic components require more or less specialization to result in fully implemented components.

2 Design and Implementation Notes

The NUOPC Layer is implemented in Fortran on top of the public ESMF Fortran API.

The NUOPC utility routines form a very straightforward Fortran API, accessible through the `NUOPC` Fortran module. The interfaces only use native Fortran types and public ESMF derived types. In order to access the utility API of the NUOPC Layer, user code must include the following two `use` lines:

```
use ESMF
use NUOPC
```

2.1 Generic Components

The NUOPC generic components are implemented as a *collection* of Fortran modules. Each module implements a single, well specified set of standard `ESMF_GridComp` or `ESMF_CplComp` methods. The nomenclature of the generic component modules starts with the `NUOPC_` prefix and continues with the kind: `Driver`, `Model`, `Mediator`, or `Connector`. The four kinds of generic components implemented by the NUOPC Layer are:

- `NUOPC_Driver` - A generic driver component. It implements a child component harness, made of State and Component objects, that follows the NUOPC Common Model Architecture. It is specialized by plugging `Model`, `Mediator`, and `Connector` components into the harness. `Driver` components can be plugged into the harness to construct component hierarchies. The generic `Driver` initializes its child components according to a standard Initialization Phase Definition, and drives their Run() methods according a customizable run sequence.
- `NUOPC_Model` - A generic model component that wraps a model code so it is suitable to be plugged into a generic `Driver` component.
- `NUOPC_Mediator` - A generic mediator component that wraps custom coupling code (flux calculations, averaging, etc.) so it is suitable to be plugged into a generic `Driver` component.
- `NUOPC_Connector` - A generic component that implements Field matching based on metadata and executes simple transforms (Regrid and Redist). It can be plugged into a generic `Driver` component.

The user code accesses the desired generic component(s) by including a `use` line for each one. Each generic component defines a small set of public names that are made available to the user code through the `use` statement. At a minimum the `SetServices` method is made public. Some of the generic components define additional public routines and labels as part of their user interface. It is recommended to rename entries of an imported generic component module, such as `SetServices`, in the local scope as part of the `use` association to prevent potential name clashes.

```
use NUOPC_<GenericComp>, &
  <GenericComp>SS      => SetServices
```

A generic component is used by user code to implement a specialized version of the generic component. The user component derives from the generic component code by implementing its own public `SetServices` routine that calls into the generic `SetServices` routine via the `NUOPC_CompDerive()` method. Typically this should be the first call made before doing anything else. It is through this mechanism that the deriving component *inherits* functionality that is implemented in the generic component. The example below shows how a specific *model* component is implemented, deriving from the generic `NUOPC_Model`:

```
use NUOPC_Model, &
  modelSS => SetServices

subroutine SetServices(model, rc)
  type(ESMF_GridComp)  :: model
  integer, intent(out) :: rc

  ! derive from NUOPC_Model
  call NUOPC_CompDerive(model, modelSS, rc=rc)

  ! specialize model
  !... calls to NUOPC_CompSpecialize() here

end subroutine
```

2.1.1 Component Specialization

After the call to `NUOPC_CompDerive()` in a component's `SetServices()` method, the component is connected to all of the generic code provided by NUOPC for the respective component kind. In order to function properly, e.g. as an atmosphere model, ocean model, driver, etc., the component must be *specialized*.

The `NUOPC_CompSpecialize()` method is used to link specific user provided routines to pre-defined NUOPC specialization points. The labels of the pre-defined specialization points are use associated named constants made available by the respective generic component module. The naming of all pre-defined specialization labels starts with the `label_` prefix, and is followed by a short intent of the specialization. E.g. `label_Advertise` refers to the specialization point responsible for advertising Fields in the import- and exportStates of the component.

There are pre-defined specialization labels for Initialize, Run, and Finalize phases. Section 2.4.1 discusses the semantic labeling of specializations in greater detail. Lists of *all* pre-defined specialization labels for Initialize, Run, and Finalize, for each of the generic NUOPC component kinds, are provided at the beginning of the respective API sections. (Driver: 3.1, Model: 3.3, Mediator: 3.4, Connector: 3.5)

The following code snippet shows a full specialization of `NUOPC_Model`, using three specialization labels:

```
use NUOPC_Model, &
```

```
    modelSS => SetServices

  subroutine SetServices(model, rc)
    type(ESMF_GridComp)  :: model
    integer, intent(out) :: rc

    rc = ESMF_SUCCESS

    ! derive from NUOPC_Model
    call NUOPC_CompDerive(model, modelSS, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

    ! specialize model
    call NUOPC_CompSpecialize(model, specLabel=label_Advertise, &
      specRoutine=Advertise, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out
    call NUOPC_CompSpecialize(model, specLabel=label_RealizeProvided, &
      specRoutine=Realize, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out
    call NUOPC_CompSpecialize(model, specLabel=label_Advance, &
      specRoutine=Advance, rc=rc)
    if (ESMF_LogFoundError(rcToCheck=rc, msg=ESMF_LOGERR_PASSTHRU, &
      line=__LINE__, &
      file=__FILE__)) &
      return  ! bail out

  end subroutine
```

The user implemented specialization routines must follow the NUOPC interface definition.

```
  subroutine SpecRoutine(comp, rc)
    type(ESMF_*Comp)      :: comp
    integer, intent(out)  :: rc
  end subroutine
```

Here `type(ESMF_*Comp)` either corresponds to `type(ESMF_GridComp)` for Models, Mediators, and Drivers, or `type(ESMF_CplComp)` for Connectors.

2.1.2 Partial Specialization

Components that are derived from a generic component may choose to only specialize certain aspects, leaving other aspects unspecified. This allows a hierarchy of generic components to be implemented with a high degree of code re-use. The variable level of specialization supports the very differing user needs. Figure 1 depicts the inheritance structure of the standard generic components implemented by the NUOPC Layer. There are two trees, one is rooted in `ESMF_GridComp`, while the other is rooted in `ESMF_CplComp`.

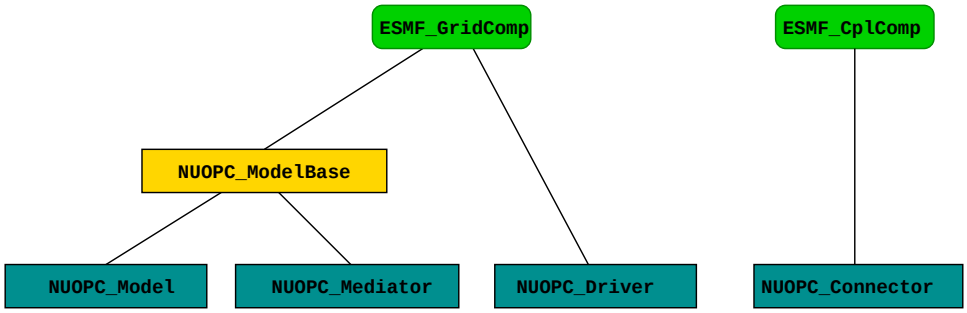

Figure 1: The NUOPC Generic Component inheritance structure. The tree on the left is rooted in `ESMF_GridComp`, while the tree on the right is rooted in `ESMF_CplComp`. The ESMF data types are shown in green. The four main NUOPC Generic Component kinds are shown in dark blue boxes. The yellow box shows a partial specialization in the inheritance tree.

2.2 Field Dictionary

The NUOPC Layer uses standard metadata on Fields to guide the decision making process that is implemented in generic code. The generic `NUOPC_Connector` component, for instance, uses the `StandardName` Attribute to construct a list of matching Fields between the import and export States. The NUOPC Field Dictionary provides a software implementation of a controlled vocabulary for the `StandardName` Field Attribute. It also associates each registered `StandardName` with `CanonicalUnits`. Currently the NUOPC Layer uses the `CanonicalUnits` entry to verify that Fields are provided in their canonical units. In the future, this entry may help support automatic unit conversion among exchanged fields.

The NUOPC Field Dictionary is set up by loading its content from a YAML 1.2 file. See section 2.2.1 for details.

Users can extend the dictionary by adding entries (field definitions or synonyms) to the YAML file, or by calling the `NUOPC_FieldDictionaryAddEntry()` interface.

2.2.1 Field Dictionary file

In a given NUOPC application, the NUOPC Field Dictionary can be set up by calling the `NUOPC_FieldDictionarySetup()` method to read in a properly-formatted YAML file. This feature is intended to improve the interoperability of codes that use the NUOPC Layer, as it allows a broader scientific community to contribute to the growth and upkeep of a common NUOPC Field Dictionary file shared among different Earth System Models. At this time, an initial version of the NUOPC Field Dictionary file is available through the dedicated GitHub repository: https://github.com/ESCOMP/NUOPCFieldDictionary, hosted within the Earth System Community Modeling Portal (ESCOMP).

A NUOPC Field Dictionary YAML file is codified as a YAML map (an unordered association of unique keys to values) with only one key: `field_dictionary`. The value associated with this key is itself a YAML map that should include the mandatory key `entries` (pointing to the complete set of dictionary entries), and may include the optional keys: `version_number`, `last_modified`, `institution`, `contact`, `source`, and `description`. These optional keys are intended to hold information about the file itself and are currently ignored by the NUOPC Layer.

Entries in the NUOPC Field dictionary are organized as YAML lists of maps. List items under the `entries` keyword must be indented and preceded with a hyphen (−).

A dictionary entry fully defines a Field if it includes both the `standard_name` and `canonical_units` keys and their associated values. This entry may also include a brief narrative describing the Field, stored as the value of the optional key `description`.

Synonyms can be defined by adding separate entries that include both the `alias` key, associated with either a single synonym (YAML scalar, e.g. `alias: <name>`) or a comma-separated list of synonyms within square brackets (YAML flow sequence, e.g. `alias: [<name1>, <name2>, ...]`), and the `standard_name` key associated with the original Field name to be substituted. The original Field name must be fully defined in the dictionary file. While adding one `alias` keyword to a Field definition dictionary entry is allowed and will be parsed by the NUOPC Layer, it is recommended that all synonyms be included as separate entries.

A NUOPC Field dictionary sample file is included below.

```
field_dictionary:
    version_number: 0.0.1
    last_modified:  2018-03-14T11:01:19Z
    institution:    National ESPC, CSC & MCL Working Groups
    contact:        esmf_support@ucar.edu
```

```
source:         https://github.com/ESCOMP/NUOPCFieldDictionary
description:    Community-based dictionary for shared coupling fields

entries:
  - standard_name: air_pressure
    canonical_units: Pa
    description: Air pressure
  - standard_name: air_temperature
    canonical_units: K
    description:
      Bulk temperature of the air,
      not the surface (skin) temperature
  - alias: p
    standard_name: air_pressure
  - alias: [ t, temp ]
    standard_name: air_temperature
```

2.2.2 Preloaded Field Dictionary

A version of the NUOPC Field Dictionary is preloaded by the NUOPC Layer at start-up, and, at this time, consists of the entries show in the table below. The value of the `StandardName` Attribute in each of these entries complies with the Climate and Forecast (CF) conventions guidelines.

2.3 Metadata

The NUOPC Layer makes extensive use of the ESMF Attribute class to implement metadata on Components, States, and Fields. ESMF Attribute Packages (or AttPacks for short) are used to build an Attribute hierarchy for each object.

In some cases the lowest level NUOPC AttPack contains a nested AttPack defined by ESMF. For all objects, the highest level of the NUOPC AttPack hierarchy is implemented with convention="NUOPC", purpose="Instance". The public NUOPC Layer API allows a user to add Attributes to the highest AttPack hierarchy level.

2.3.1 Driver Component Metadata

The Driver Component metadata is implemented through ESMF_Info. It can be accessed using the JSON Pointer "/NUOPC/Instance/" prefix followed by the "Attribute name" as per the table below. E.g. "Verbosity" is accessed using `key="/NUOPC/Instance/Verbosity"`.

Note that some of the Attribute names in the following table are longer than the table column width. In these cases the Attribute name had to be broken into multiple lines. When that happens, a hyphen shows up to indicate the line break. The hyphen is *not* part of the Attribute name!

Attribute name	Definition	Controlled vocabulary
`Kind`	String value indicating component kind.	Driver

`Verbosity`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control verbosity of the generic component implementation. Higher bits are available for user level verbosity control. **bit 0**: Intro/Extro of methods with indentation. **bit 1**: Intro/Extro with memory info. **bit 2**: Intro/Extro with garbage collection info. **bit 3**: Intro/Extro with local VM info. **bit 4**: Intro/Extro with ImportState info. **bit 5**: Intro/Extro with ExportState info. **bit 6**: Log hierarchy protocol details. **bit 8**: Log Initialize phase with >>>, <<<, and currTime. **bit 9**: Log Run phase with >>>, <<<, and currTime. **bit 10**: Log Finalize phase with >>>, <<<, and currTime. **bit 11**: Log info about data dependency during initialize resolution. **bit 12**: Log run sequence execution. **bit 13**: Log Component creation and destruction. **bit 14**: Log State creation and destruction.	0, 1, 2, ... "off" = 0 (default), "low": some verbosity, bits: 0, 8, 9, 10, 13 "high": more verbosity, bits: 0, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14 "max": all lower 16 bits
`Profiling`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control profiling of the generic component implementation. Higher bits are available for user level profiling control. **bit 0**: Top level profiling of *Initialize* phases. **bit 1**: Specialization point profiling of *Initialize* phases. **bit 2**: Additional profiling of internals of *Initialize* phases. **bit 3**: Top level profiling of *Run* phases. **bit 4**: Specialization point profiling of *Run* phases. **bit 5**: Additional profiling of internals of *Run* phases. **bit 6**: Top level profiling of *Finalize* phases. **bit 7**: Specialization point profiling of *Finalize* phases. **bit 8**: Additional profiling of internals of *Finalize* phases. **bit 9**: Leading barrier for *Initialize* phases. **bit 10**: Leading barrier for *Run* phases. **bit 11**: Leading barrier for *Finalize* phases. **bit 12**: Run sequence iteration events.	0, 1, 2, ... "off" = 0 (default), "low": Top level profiling. "high": Top level, specialization point profiling, and additional profiling of internals. "max": All lower 16 bits set.
`CompLabel`	String value holding the label under which the component was added to its parent driver.	*no restriction*
`InitializePhaseMap`	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
`RunPhaseMap`	List of string values, mapping the logical NUOPC run phases to the actual ESMF run phase number under which the entry point is registered.	*label-string*=Z, where *label-string* can be chosen freely, and Z = actual ESMF phase number.

`FinalizePhaseMap`	List of string values, mapping the logical NUOPC finalize phases to the actual ESMF finalize phase number under which the entry point is registered.	*label-string*=Z, where *label-string* can be chosen freely, and Z = actual ESMF phase number.
`Internal-InitializePhaseMap`	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
`NestingGeneration`	Integer value enumerating nesting level.	0, 1, 2, ...
`Nestling`	Integer value enumerating siblings within the same generation.	0, 1, 2, ...
`Initialize-DataResolution`	String value indicating whether the resolution loop is disabled or enabled.	false, true
`Initialize-DataComplete`	String value indicating whether all initialize data dependencies have been satisfied.	false, true
`Initialize-DataProgress`	String value indicating whether progress is being made resolving initialize data dependencies.	false, true
`HierarchyProtocol`	String value specifying the hierarchy protocol.	"PushUpAllExportsAndUnsatisfiedImpor - activates field mirroring of all exports and unsatisfied imports. By default use reference sharing for the mirrored fields and geom objects. This is the default behavior without having `HierarchyProtocol` set. "ConnectProvidedFields"- no field mirroring, only connect to externally provided fields in the import- and exportStates. "Explorer" - like the default, but do not use reference sharing. *All other values currently disable the hierarchy protocol.*

2.3.2 Model Component Metadata

The Model Component metadata is implemented through ESMF_Info. It can be accessed using the JSON Pointer "/NUOPC/Instance/" prefix followed by the "Attribute name" as per the table below. E.g. "Verbosity" is accessed using `key="/NUOPC/Instance/Verbosity"`.

Note that some of the Attribute names in the following table are longer than the table column width. In these cases the Attribute name had to be broken into multiple lines. When that happens, a hyphen shows up to indicate the line break. The hyphen is *not* part of the Attribute name!

Attribute name	**Definition**	**Controlled vocabulary**
`Kind`	String value indicating component kind.	Model

`Verbosity`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control verbosity of the generic component implementation. Higher bits are available for user level verbosity control. **bit 0**: Intro/Extro of methods with indentation. **bit 1**: Intro/Extro with memory info. **bit 2**: Intro/Extro with garbage collection info. **bit 3**: Intro/Extro with local VM info. **bit 4**: Intro/Extro with ImportState info. **bit 5**: Intro/Extro with ExportState info. **bit 8**: Log Initialize phase with >>>, <<<, and currTime. **bit 9**: Log Run phase with >>>, <<<, and currTime. **bit 10**: Log Finalize phase with >>>, <<<, and currTime. **bit 11**: Log info about data dependency during initialize resolution. **bit 12**: Log run sequence execution.	0, 1, 2, ... "off" = 0 (default), "low": some verbosity, bits: 0, 8, 9, 10, 13 "high": more verbosity, bits: 0, 4, 5, 8, 9, 10, 11, 12, 13, 14 "max": all lower 16 bits
`Profiling`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control profiling of the generic component implementation. Higher bits are available for user level profiling control. **bit 0**: Top level profiling of *Initialize* phases. **bit 1**: Specialization point profiling of *Initialize* phases. **bit 2**: Additional profiling of internals of *Initialize* phases. **bit 3**: Top level profiling of *Run* phases. **bit 4**: Specialization point profiling of *Run* phases. **bit 5**: Additional profiling of internals of *Run* phases. **bit 6**: Top level profiling of *Finalize* phases. **bit 7**: Specialization point profiling of *Finalize* phases. **bit 8**: Additional profiling of internals of *Finalize* phases. **bit 9**: Leading barrier for *Initialize* phases. **bit 10**: Leading barrier for *Run* phases. **bit 11**: Leading barrier for *Finalize* phases.	0, 1, 2, ... "off" = 0 (default), "low": Top level profiling. "high": Top level, specialization point profiling, and additional profiling of internals. "max": All lower 16 bits set.

`Diagnostic`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control diagnostic of the generic component implementation. Higher bits are available for user level diagnostic control. **bit 0**: Dump fields of the importState on entering *Initialize* phases. **bit 1**: Dump fields of the exportState on entering *Initialize* phases. **bit 2**: Dump fields of the importState on exiting *Initialize* phases. **bit 3**: Dump fields of the exportState on exiting *Initialize* phases. **bit 4**: Dump fields of the importState on entering *Run* phases. **bit 5**: Dump fields of the exportState on entering *Run* phases. **bit 6**: Dump fields of the importState on exiting *Run* phases. **bit 7**: Dump fields of the exportState on exiting *Run* phases. **bit 8**: Dump fields of the importState on entering *Finalize* phases. **bit 9**: Dump fields of the exportState on entering *Finalize* phases. **bit 10**: Dump fields of the importState on exiting *Finalize* phases. **bit 11**: Dump fields of the exportState on exiting *Finalize* phases.	0, 1, 2, ... "off" = 0 (default), "max": All lower 16 bits set.
`CompLabel`	String value holding the label under which the component was added to its parent driver.	*no restriction*
`InitializePhaseMap`	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
`RunPhaseMap`	List of string values, mapping the logical NUOPC run phases to the actual ESMF run phase number under which the entry point is registered.	*label-string*=Z, where *label-string* can be chosen freely, and Z = actual ESMF phase number.
`FinalizePhaseMap`	List of string values, mapping the logical NUOPC finalize phases to the actual ESMF finalize phase number under which the entry point is registered.	*label-string*=Z, where *label-string* can be chosen freely, and Z = actual ESMF phase number.
`Internal-InitializePhaseMap`	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
`NestingGeneration`	Integer value enumerating nesting level.	0, 1, 2, ...
`Nestling`	Integer value enumerating siblings within the same generation.	0, 1, 2, ...
`Initialize-DataComplete`	String value indicating whether all initialize data dependencies have been satisfied.	false, true

`Initialize-DataProgress`	String value indicating whether progress is being made resolving initialize data dependencies.	false, true
`HierarchyProtocol`	String value specifying the hierarchy protocol.	"PushUpAllExportsAndUnsatisfiedImpor for field mirroring and connecting, "ConnectProvidedFields" to only connect provided fields (no mirroring), *All other values currently disable the hierarchy protocol.*

2.3.3 Mediator Component Metadata

The Mediator Component metadata is implemented through ESMF_Info. It can be accessed using the JSON Pointer "/NUOPC/Instance/" prefix followed by the "Attribute name" as per the table below. E.g. "Verbosity" is accessed using `key="/NUOPC/Instance/Verbosity"`.

Note that some of the Attribute names in the following table are longer than the table column width. In these cases the Attribute name had to be broken into multiple lines. When that happens, a hyphen shows up to indicate the line break. The hyphen is *not* part of the Attribute name!

Attribute name	**Definition**	**Controlled vocabulary**
`Kind`	String value indicating component kind.	Mediator
`Verbosity`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control verbosity of the generic component implementation. Higher bits are available for user level verbosity control. **bit 0**: Intro/Extro of methods with indentation. **bit 1**: Intro/Extro with memory info. **bit 2**: Intro/Extro with garbage collection info. **bit 3**: Intro/Extro with local VM info. **bit 4**: Intro/Extro with ImportState info. **bit 5**: Intro/Extro with ExportState info. **bit 8**: Log Initialize phase with >>>, <<<, and currTime. **bit 9**: Log Run phase with >>>, <<<, and currTime. **bit 10**: Log Finalize phase with >>>, <<<, and currTime. **bit 11**: Log info about data dependency during initialize resolution. **bit 12**: Log run sequence execution.	0, 1, 2, ... "off" = 0 (default), "low": some verbosity, bits: 0, 8, 9, 10, 13 "high": more verbosity, bits: 0, 4, 5, 8, 9, 10, 11, 12, 13, 14 "max": all lower 16 bits

`Profiling`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control profiling of the generic component implementation. Higher bits are available for user level profiling control. **bit 0**: Top level profiling of *Initialize* phases. **bit 1**: Specialization point profiling of *Initialize* phases. **bit 2**: Additional profiling of internals of *Initialize* phases. **bit 3**: Top level profiling of *Run* phases. **bit 4**: Specialization point profiling of *Run* phases. **bit 5**: Additional profiling of internals of *Run* phases. **bit 6**: Top level profiling of *Finalize* phases. **bit 7**: Specialization point profiling of *Finalize* phases. **bit 8**: Additional profiling of internals of *Finalize* phases. **bit 9**: Leading barrier for *Initialize* phases. **bit 10**: Leading barrier for *Run* phases. **bit 11**: Leading barrier for *Finalize* phases.	0, 1, 2, ... "off" = 0 (default), "low": Top level profiling. "high": Top level, specialization point profiling, and additional profiling of internals. "max": All lower 16 bits set.
`Diagnostic`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control diagnostic of the generic component implementation. Higher bits are available for user level diagnostic control. **bit 0**: Dump fields of the importState on entering *Initialize* phases. **bit 1**: Dump fields of the exportState on entering *Initialize* phases. **bit 2**: Dump fields of the importState on exiting *Initialize* phases. **bit 3**: Dump fields of the exportState on exiting *Initialize* phases. **bit 4**: Dump fields of the importState on entering *Run* phases. **bit 5**: Dump fields of the exportState on entering *Run* phases. **bit 6**: Dump fields of the importState on exiting *Run* phases. **bit 7**: Dump fields of the exportState on exiting *Run* phases. **bit 8**: Dump fields of the importState on entering *Finalize* phases. **bit 9**: Dump fields of the exportState on entering *Finalize* phases. **bit 10**: Dump fields of the importState on exiting *Finalize* phases. **bit 11**: Dump fields of the exportState on exiting *Finalize* phases.	0, 1, 2, ... "off" = 0 (default), "max": All lower 16 bits set.
`CompLabel`	String value holding the label under which the component was added to its parent driver.	*no restriction*

`InitializePhaseMap`	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
`RunPhaseMap`	List of string values, mapping the logical NUOPC run phases to the actual ESMF run phase number under which the entry point is registered.	*label-string*=Z, where *label-string* can be chosen freely, and Z = actual ESMF phase number.
`FinalizePhaseMap`	List of string values, mapping the logical NUOPC finalize phases to the actual ESMF finalize phase number under which the entry point is registered.	*label-string*=Z, where *label-string* can be chosen freely, and Z = actual ESMF phase number.
`Internal-InitializePhaseMap`	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
`NestingGeneration`	Integer value enumerating nesting level.	0, 1, 2, ...
`Nestling`	Integer value enumerating siblings within the same generation.	0, 1, 2, ...
`Initialize-DataComplete`	String value indicating whether all initialize data dependencies have been satisfied.	false, true
`Initialize-DataProgress`	String value indicating whether progress is being made resolving initialize data dependencies.	false, true
`HierarchyProtocol`	String value specifying the hierarchy protocol.	"PushUpAllExportsAndUnsatisfiedImpor for field mirroring and connecting, "ConnectProvidedFields" to only connect provided fields (no mirroring), *All other values currently disable the hierarchy protocol.*

2.3.4 Connector Component Metadata

The Connector Component metadata is implemented through ESMF_Info. It can be accessed using the JSON Pointer "/NUOPC/Instance/" prefix followed by the "Attribute name" as per the table below. E.g. "Verbosity" is accessed using `key="/NUOPC/Instance/Verbosity"`.

Attribute name	**Definition**	**Controlled vocabulary**
`Kind`	String value indicating component kind.	Connector

`Verbosity`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control verbosity of the generic component implementation. Higher bits are available for user level verbosity control. **bit 0**: Intro/Extro of methods with indentation. **bit 1**: Intro/Extro with memory info. **bit 2**: Intro/Extro with garbage collection info. **bit 3**: Intro/Extro with local VM info. **bit 4**: Intro/Extro with ImportState info. **bit 5**: Intro/Extro with ExportState info. **bit 8**: Log FieldTransferPolicy. **bit 9**: Log bond level info. **bit 10**: Log CplList construction. **bit 11**: Log GeomObject Transfer. **bit 12**: Log looping over all elements in CplList for RouteHandle computation, FieldSharing, and Timestamp propagation. **bit 13**: Log Run phase with >>>, <<<, and currTime. **bit 14**: Log info about RouteHandle execution. **bit 15**: Log info about RouteHandle release.	0, 1, 2, ... "off" = 0 (default), "low": some verbosity, bits: 0, 13 "high": more verbosity, bits: 0, 4, 5, 8, 9, 10, 11, 12, 13, 14, 15 "max": all lower 16 bits
`Profiling`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control profiling of the generic component implementation. Higher bits are available for user level profiling control. **bit 0**: Top level profiling of *Initialize* phases. **bit 1**: Specialization point profiling of *Initialize* phases. **bit 2**: Additional profiling of internals of *Initialize* phases. **bit 3**: Top level profiling of *Run* phases. **bit 4**: Specialization point profiling of *Run* phases. **bit 5**: Additional profiling of internals of *Run* phases. **bit 6**: Top level profiling of *Finalize* phases. **bit 7**: Specialization point profiling of *Finalize* phases. **bit 8**: Additional profiling of internals of *Finalize* phases. **bit 9**: Leading barrier for *Initialize* phases. **bit 10**: Leading barrier for *Run* phases. **bit 11**: Leading barrier for *Finalize* phases.	0, 1, 2, ... "off" = 0 (default), "low": Top level profiling. "high": Top level, specialization point profiling, and additional profiling of internals. "max": All lower 16 bits set.

`Diagnostic`	String value, converted into an integer, and interpreted as a bit field. The lower 16 bits (0-15) are reserved to control diagnostic of the generic component implementation. Higher bits are available for user level diagnostic control. **bit 0**: Dump fields of the importState on entering *Initialize* phases. **bit 1**: Dump fields of the exportState on entering *Initialize* phases. **bit 2**: Dump fields of the importState on exiting *Initialize* phases. **bit 3**: Dump fields of the exportState on exiting *Initialize* phases. **bit 4**: Dump fields of the importState on entering *Run* phases. **bit 5**: Dump fields of the exportState on entering *Run* phases. **bit 6**: Dump fields of the importState on exiting *Run* phases. **bit 7**: Dump fields of the exportState on exiting *Run* phases. **bit 8**: Dump fields of the importState on entering *Finalize* phases. **bit 9**: Dump fields of the exportState on entering *Finalize* phases. **bit 10**: Dump fields of the importState on exiting *Finalize* phases. **bit 11**: Dump fields of the exportState on exiting *Finalize* phases.	0, 1, 2, ... "off" = 0 (default), "max": All lower 16 bits set.
`CompLabel`	String value holding the label under which the component was added to its parent driver.	*no restriction*
`InitializePhaseMap`	List of string values, mapping the logical NUOPC initialize phases, of a specific Initialize Phase Definition (IPD) version, to the actual ESMF initialize phase number under which the entry point is registered.	IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z > 0 and Y, Z < 10
`RunPhaseMap`	List of string values, mapping the logical NUOPC run phases to the actual ESMF run phase number under which the entry point is registered.	*label-string*=Z, where *label-string* can be chosen freely, and Z = actual ESMF phase number.
`FinalizePhaseMap`	List of string values, mapping the logical NUOPC finalize phases to the actual ESMF finalize phase number under which the entry point is registered.	*label-string*=Z, where *label-string* can be chosen freely, and Z = actual ESMF phase number.
`CplList`	List of StandardNames of the connected Fields. Each StandardName entry may be followed by a colon separated list of connection options. The details are discussed in section 2.4.5	*Standard names* as per field dictionary, followed by *connection options* defined in section 2.4.5.
`CplSetList`	List of coupling sets. Each coupling set is identified by a string value.	*no restriction*

ConnectionOptions	String value specifying the connection options to be applied to all the fields in the CplList by default.	*Connection options* defined in section 2.4.5.
EpochEnable	String value specifying whether EPOCH support is enabled inside the Connector. The default setting is "true".	false, true
EpochEnterKeepAlloc	String value specifying whether to keep internal allocations when entering the EPOCH for reuse. The default setting is "true".	false, true
EpochExitKeepAlloc	String value specifying whether to keep internal allocations when exiting the EPOCH for reuse. The default setting is "true".	false, true
EpochThrottle	Integer specifying the maximum number of outstanding EPOCH messages between any two PETs. The default throttle level is 10.	Any positive integer.

2.3.5 State Metadata

The State metadata is implemented through ESMF_Info. It can be accessed using the JSON Pointer "/NUOPC/Instance/" prefix followed by the "Attribute name" as per the table below. E.g. "Namespace" is accessed using `key="/NUOPC/Instance/Namespace"`.

Attribute name	**Definition**	**Controlled vocabulary**
Namespace	String value holding the namespace that applies to all of the objects contained in the State.	*no restriction*
FieldTransferPolicy	String value indicating to Connector whether to mirror transfer Fields into this State.	transferNone, transferAll, transferAllWithNamespace
CplSet	String value holding the coupling set name that applies to all of the objects contained in the State.	*no restriction*

2.3.6 Field Metadata

The Field metadata is implemented through ESMF_Info. It can be accessed using the JSON Pointer "/NUOPC/Instance/" prefix followed by the "Attribute name" as per the table below. E.g. "StandardName" is accessed using `key="/NUOPC/Instance/StandardName"`.

Attribute name	**Definition**	**Controlled vocabulary**
StandardName	String value	*no restriction*
Units	String value	*no restriction*
LongName	String value	*no restriction*
ShortName	String value	*no restriction*
Connected	Connected status.	false, true
ProducerConnection	String value indicating whether the Field has been connected with a producer.	open, targeted, connected

ConsumerConnection	String value indicating whether the Field has been connected with a consumer.	open, targeted, connected
Updated	String value indicating updated status during initialization.	false, true
ProducerTransferOffer	String value indicating a producer component's ability to transfer information about the advertised Field, including its GeomObject.	will provide, can provide, cannot provide
ProducerTransferAction	String value indicating the action a producer component is supposed to take with respect to transferring Field information, including its GeomObject.	provide, accept
ConsumerTransferOffer	String value indicating a consumer component's ability to transfer information about the advertised Field, including its GeomObject.	will provide, can provide, cannot provide
ConsumerTransferAction	String value indicating the action a consumer component is supposed to take with respect to transferring Field information, including its GeomObject.	provide, accept
SharePolicyField	String value indicating a component's policy with respect to sharing the Field data allocation.	share, not share
ShareStatusField	String value indicating the status with respect to sharing the underlying Field data allocation that was negotiated.	shared, not shared
SharePolicyGeomObject	String value indicating a component's policy with respect to sharing the Grid or Mesh on which the advertised Field object is defined.	share, not share
ShareStatusGeomObject	String value indicating the status with respect to sharing the underlying GeomObject that was negotiated.	shared, not shared
UngriddedLBound	Integer value list. If present equals the ungriddedLBound of the provider field during a GeomObject transfer.	*no restriction*
UngriddedUBound	Integer value list. If present equals the ungriddedUBound of the provider field.during a GeomObject transfer.	*no restriction*
GridToFieldMap	Integer value list. If present equals the gridToFieldMap of the provider field.during a GeomObject transfer.	*no restriction*
ArbDimCount	Integer value. If present equals the arbDimCount of the provider field.during a GeomObject transfer.	*no restriction*
MinIndex	Integer value list. If present equals the minIndex (of tile 1) of the provider field.during a GeomObject transfer.	*no restriction*
MaxIndex	Integer value list. If present equals the maxIndex (of tile 1) of the provider field.during a GeomObject transfer.	*no restriction*

`TypeKind`	Integer value. If present equals the integer representation of `typekind` of the provider field.during a GeomObject transfer.	*implementation dependent range*
`GeomLoc`	Integer value. If present equals the integer representation of `staggerloc` (for Grid) or `meshloc` (for Mesh) of the provider field.during a GeomObject transfer.	*implementation dependent range*

2.4 Initialization

2.4.1 Phase Maps, Semantic Specialization Labels, and Component Labels

The NUOPC layer adds an abstraction on top of the ESMF phase index. ESMF introduces the concept of standard component methods: `Initialize`, `Run`, and `Finalize`. ESMF further recognizes the need for being able to split each of the standard methods into multiple phases. On the ESMF level, phases are implemented by a simple integer `phase` index. With NUOPC, logical phase labels are introduced that are mapped to the ESMF phase indices.

The NUOPC Layer introducing three component level attributes: `InitializePhaseMap`, `RunPhaseMap`, and `FinalizePhaseMap`. These attributes map logical NUOPC phase labels to integer ESMF phase indices. A NUOPC compliant component fully documents its available phases through the phase maps.

The generic `NUOPC_Driver` uses the `InitializePhaseMap` on each of its child component during the initialization stage to correctly interact with each component. The `RunPhaseMap` is used when setting up run sequences in the Driver. The `NUOPC_DriverAddRunElement()` takes the `phaseLabel` argument, and uses the `RunPhaseMap` attribute internally to translates the label into the corresponding ESMF phase index. The `FinalizePhaseMap` is currently not used by the NUOPC Layer

Appendix B, section 7, lists the supported logical phase labels for reference. User code very rare needs to interact with the `InitializePhaseMap` or its entries directly. Instead, user code specializes the initialization behavior of a component through the semantic specialization labels discussed below.

NUOPC implements a very powerful initialization procedure. This procedure is, among other functions, capable of handling component hierarchies, transfer of geometries, reference sharing, and resolving data dependencies during initialization. The initialization features are discussed in detail in their respective sections of this document.

From the user level, specialization of the initialization is accessbile through the *semantic specialization labels*. These labels are predefined named constants that are passed into the `NUOPC_CompSpecialize()` method, together with the user provided routine, implementing the required actions. On a technical level, the user routine must follow the standard interface defined by NUOPC. Semantically, the purpose of each specialization point is indicated by the name of the predefined specialization label. For a definition of the labels, and the ascribed purpose, see the SEMANTIC SPECIALIZATION LABELS section under each of the generic component kinds. (Driver: 3.1, Model: 3.3, Mediator: 3.4, Connector: 3.5)

Finally, under NUOPC, each component is associated with a label when it is added to a driver through the `NUOPC_DriverAddComp()` call. Multiple instances of the same component can be added to a driver, provided each instance is given a unique label. Connectors between components are identified by providing the label of the source component and destination component.

2.4.2 Field Pairing

The NUOPC Model and Mediator components are required to advertise their import and export Fields with a standard set of Field metadata. This set includes the `StandardName` attribute. The NUOPC Layer implements a strategy of pairing advertised Fields that is based primarily on the `StandardName` of the Fields, and in more complex situations further utilizes the `Namespace` attribute on States.

Field pairing is accomplished as part of the initialization procedure and is a collective effort of the Driver and its child components: Models, Mediator, Connectors. The Connectors are the most active players when it comes to Field pairing. The end result of the process is where each Connector has a list of Fields that it connects between its importState and its exportState. Each connector keeps this list in its component level metadata as `CplList` attribute.

During the first stage of Field pairing, each Connector matches all of the Fields in its importState to all of the Fields

in its exportState by looking at their StandardName attribute. For every match a *bondLevel* is calculated and stored in the Field on the export side, i.e. on the consumer side of the connection, in the Field's ConsumerConnection attribute. The larges found bondLevel is kept for each Field on the export side.

The bondLevel is a measure of how strong the pairing is considering the namespace rules explained in section 2.4.3. Without the use of namespaces the bondLevel for all Field pairs that match by their StandardName is equal to 1.

After the first stage, there may be umbiguous Field pairs present. Ambiguous Field pairs are those that map different producer Fields (i.e. Fields in the importState of a Connector) to the *same* consumer Field (i.e. a Field in the exportState of a Connector). While the NUOPC Layer support having multiple consumer Fields connected to a single producer Field, it does not support the opposite condition. The second stage of Field pairing is responsible for disambiguating Field pairs with the same consumer Field.

Field pair disambiguation is based on the *bondLevel* that was calculated and stored on the consumer side Field for each pair during the first stage. The disambiguation rule simply selects the connection with the highest bondLevel and discards all lesser connection to the same consumer side Field. However, if the highest bondLevel is not unique, i.e. there are multiple pairs with the same bondLevel, disambiguation is not possible and an error is returned to the Driver by the Connector that finds the ambiguity first.

Assuming that the disambiguation step was successful, each Connector holds a valid CplList attribute with entries that correspond to the Field pairs that it is responsible for. At this stage the Driver can still overwrite this attribute and implement custom pairs if that is desired.

2.4.3 Namespaces

Namespaces are used to control and fine-tune the disambiguation of Field pairs during the initialization. The general procedure of Field pairing and disambiguation is outlined in section 2.4.2, here the use of namespaces is described.

The NUOPC Layer implements namespaces through the Namespace attribute on ESMF_State objects. The value of this attribute is a simple character string. The NUOPC Layer automatically creates the import and export States of every Model and Mediator component that is added to a Driver. The Namespace attribute of these States is automatically set to the compLabel string that was provided during NUOPC_DriverAdd(). Doing this places every Field that is advertised through these States inside the component's unique namespace.

A secondary namespace can be added to a State using the NUOPC_StateNamespaceAdd() method. This creates a new State that is nested inside of an existing State, and sets the Namespace attribute of the new State. Fields that are advertised inside of such a nested State are in a namespace with two parts: NS1:NS2. Here NS1 is the preset namespace of the import or export State (equal to the compLabel), and NS2 is a freely chosen namespace string.

During Field pairing the namespace on each side of the connection is considered in the two part format NS1:NS2. The first part is equal to the compLabel of the corresponding component, and NS2 is either the namespace of a nested State, or empty if the Field is not inside a nested State. Using this format, the calculation of the *bondLevel* during Field pairing is governed by the following rules:

- Namespace matching is done in a cross wise fashion, meaning NS1 from one side is compared to NS2 of the other side, and vice versa.
- The bondLevel is incremented by one counter for each cross-wise match between namespaces. (Considering that the bondLevel starts out as 1 for any Field pair with matching standard names, the maximum bondLevel that can be reached is 3.)
- Finding one side of the cross-wise comparison being an empty string is neither counted as a match nor a mismatch. The bondLevel remains unchanged.

- A Field pair for which a mis-match in either of the two cross-wise namespace comparisons is detected is discarded from the possible pairs. It is not further considered.

In practice then, a component that targets a specific other component with its advertised Fields would add a secondary namespace to its import or export State, and set that namespace to the compLabel of the targeted component. This increases the bondLevel for each pair from 1 to 2. An even higher bondLevel of 3 is achieved when both sides target each other by specifying the other component's compLabel through a secondary namespace.

In conclusion, namespaces can affect the bondLevel calculation for each pair, but they do not affect how pairs are constructed and disambiguated. In particular, the requirement for unambiguous Field pairs for each consumer Field remains unchanged, and it is an error condition if the highest bondLevel for a consumer Field does not correspond to a unique Field pair.

2.4.4 Using Coupling Sets for Coupling Multiple Nests

The NUOPC Layer can couple multiple data sets by adding nested states to the import and export states of a `NUOPC_Model`. Each nested state is given a couple set identifier at the time it is added to the parent state. This identifier guarantees a `NUOPC_Connector` will only pair fields within this nested state to fields in a connected state with an identical identifier.

During `label_Advertise`, before calling `NUOPC_Advertise` (using methods **??** or **??**), add nested states to import and export states using `NUOPC_AddNestedState`. Each nested state is given a couple set identifier using the CplSet argument, see **??**. The nested states can then be used to advertise and realize fields. Each nested state may contain fields with identical standard names or unique standard names. Fields in each nested state will only connect to fields in another state if that state has an identical couple set identifier.

For a complete example of how to couple sets using the NUOPC API, see https://github.com/esmf-org/nuopc-app-prototypes/tree/develop/AtmOcnCplSetProto. The following code snippets demonstrates the critical pieces of code used to add a nested state with a couple set identifier.

```
subroutine Advertise(model, rc)
  type(ESMF_GridComp)  :: model
  integer, intent(out) :: rc

  ! local variables
  type(ESMF_State) :: importState, exportState
  type(ESMF_State) :: NStateImp1, NStateImp2
  type(ESMF_State) :: NStateExp1, NStateExp2

  rc = ESMF_SUCCESS

  ! query model for importState and exportState
  call NUOPC_ModelGet(model, importState=importState, &
    exportState=exportState, rc=rc)
  ! check rc

  ! add nested import states with couple set identifier
  call NUOPC_AddNestedState(importState, &
    CplSet="Nest1", nestedStateName="NestedStateImp_N1", &
    nestedState=NStateImp1, rc=rc)
  ! check rc
```

```
    call NUOPC_AddNestedState(importState, &
      CplSet="Nest2", nestedStateName="NestedStateImp_N2", &
      nestedState=NStateImp2, rc=rc)
    ! check rc

    ! add nested export states with couple set identifier
    call NUOPC_AddNestedState(exportState, &
      CplSet="Nest1", nestedStateName="NestedStateExp_N1", &
      nestedState=NStateExp1, rc=rc)
    ! check rc
    call NUOPC_AddNestedState(exportState, &
      CplSet="Nest2", nestedStateName="NestedStateExp_N2", &
      nestedState=NStateExp2, rc=rc)
    ! check rc

    ! importable field: sea_surface_temperature
    call NUOPC_Advertise(NStateImp1, &
      StandardName="sea_surface_temperature", name="sst", rc=rc)
    ! check rc
    call NUOPC_Advertise(NStateImp2, &
      StandardName="sea_surface_temperature", name="sst", rc=rc)
    ! check rc

    ! exportable field: air_pressure_at_sea_level
    call NUOPC_Advertise(NStateExp1, &
      StandardName="air_pressure_at_sea_level", name="pmsl", rc=rc)
    ! check rc
    call NUOPC_Advertise(NStateExp2, &
      StandardName="air_pressure_at_sea_level", name="pmsl", rc=rc)
    ! check rc

    ! exportable field: surface_net_downward_shortwave_flux
    call NUOPC_Advertise(NStateExp1, &
      StandardName="surface_net_downward_shortwave_flux", name="rsns", rc=rc)
    ! check rc
    call NUOPC_Advertise(NStateExp2, &
      StandardName="surface_net_downward_shortwave_flux", name="rsns", rc=rc)
    ! check rc

  end subroutine
```

2.4.5 Connection Options

Once the field pairing discussed in the previous sections is completed, each Connector component holds an attribute by the name of `CplList`. The `CplList` is a list type attribute with as many entries as there are fields for which the Connector component is responsible for connecting. The first part of each of these entries is always the `StandardName` of the associated field. See section 2.2 for a discussion of the NUOPC field dictionary and standard names.

After the `StandardName` part, each `CplList` entry may optionally contain a string of *connection options*. Each Driver component has the chance as part of the `label_ModifyInitializePhaseMap` specialization, to modify

the CplList attribute of all the Connectors that it drives.

The individual connection options are colon separated, leading to the following format for each CplList entry:

```
StandardName[:option1[:option2[: ...]]]
```

The format of the options is:

```
OptionName=value1[=spec1][,value2[=spec2][, ...]]
```

OptionName and the value strings are case insensitive. There are single and multi-valued options as indicated in the table below. For single valued options only value1 is relevant. If the same option is listed multiple times, only the first occurrence will be used. If an option has a default value, it is indicated in the table. If a value requires additional specification via =spec then the specifications are listed in the table.

OptionName	**Definition**	**Type**	**Values**
dstMaskValues	List of integer values that defines the mask values.	multi	List of integers.
dumpWeights	Enable or disable dumping of the interpolation weights into a file.	single	true, false(default)
extrapDistExponent	The exponent to raise the distance to when calculating weights for the nearest_idavg extrapolation method.	single	real(default 2.0)
extrapMethod	Fill in points not mapped by the regrid method.	single	none(default), nearest_idavg, nearest_stod, nearest_d, creep, creep_nrst_d
extrapNumLevels	The number of levels to output for the extrapolation methods that fill levels. When a method is used that requires this, then an error will be returned, if it is not specified.	single	integer
extrapNumSrcPnts	The number of source points to use for the extrapolation methods that use more than one source point.	single	integer(default 8)
ignoreDegenerate	Ignore degenerate cells when checking the input Grids or Meshes for errors.	single	true, false(default)
ignoreUnmatchedIndices	Ignore unmatched sequence indices when redistributing between source and destination index space.	single	true, false(default)

pipelineDepth	Maximum number of outstanding non-blocking communication calls during the parallel interpolation. Only relevant for cases where the automatic tuning procedure fails to find a setting that works well on a given hardware.	single	integer
poleMethod	Extrapolation method around the pole(s).	single	none(default), allavg, npntavg=*"integer indicating number of points"*, teeth
remapMethod	Redistribution or interpolation to compute the regridding weights.	single	redist, bilinear(default), patch, nearest_stod, nearest_dtos, conserve, conserve_2nd
srcMaskValues	List of integer values that defines the mask values.	multi	List of integers.
srcTermProcessing	Number of terms in each partial sum of the interpolation to process on the source side. This setting impacts the bit-for-bit reproducibility of the parallel interpolation results between runs. The strictest bit-for-bit setting is achieved by setting the value to 1.	single	integer
termOrder	Order of the terms in each partial sum of the interpolation. This setting impacts the bit-for-bit reproducibility of the parallel interpolation results between runs. The strictest bit-for-bit setting is achieved by setting the value to srcseq.	single	free(default), srcseq, srcpet
unmappedAction	The action to take when unmapped destination elements are encountered.	single	ignore(default), error
zeroRegion	The region of destination elements set to zero before adding the result of the sparse matrix multiplication. The available options support total, selective, or no zeroing of destination elements.	single	total(default), select, empty

2.4.6 Data-Dependencies during Initialize

For multi-model applications it is not uncommon that during start-up one or more components depends on data from one or more other components. These types of data-dependencies during initialize can become very complex very quickly. Finding the "correct" sequence to initialize all components for a complex dependency graph is not trivial. The NUOPC Layer deals with this issue by repeatedly looping over all components that indicate that their initialization has data dependencies on other components. The loop is finally exited when either all components have indicated completion of their initialization, or a dead-lock situation is being detected by the NUOPC Layer.

The data-dependency resolution loop considers all components that have specialized `label_DataInitialize`. Participating components communicate their current status to the NUOPC Layer via Field and Component metadata. Every time a component's `label_DataInitialize` specialization routine is called, it is responsible for checking the Fields in the importState and for initializing any internal data structures and Fields in the exportState. Fields that are fully initialized in the exportState must be indicated by setting their `Updated` Attribute to "true". This is used by the NUOPC Layer to ensure that there is continued progress during the resolution loop iterations. Once the component is fully initialized it must further set its `InitializeDataComplete` Attribute to "true" before returning.

During the execution of the data-dependency resolution loop the NUOPC Layer calls all of the Connectors *to* a Model/Mediator component before calling the component's `label_DataInitialize`. Doing so ensures that all the currently available Fields are passed to the component before it tries to access them. Once a component has set its `InitializeDataComplete` Attribute to "true", it, and the Connectors to it, will no longer be called during the remainder of the resolution loop.

When *all* of the components that participate in the data-dependency resolution loop have set their `InitializeDataComplete` Attribute to "true", the NUOPC Layer successfully exits the data-dependency resolution loop. The loop is also interrupted before all `InitializeDataComplete` Attributes are set to "true" if a full cycle completes without any indicated progress. The NUOPC Layer flags this situation as a potential dead-lock and returns with error.

2.4.7 Transfer of Grid/Mesh/LocStream Objects between Components

There are modeling scenarios where the need arises to transfer physical grid information from one component to another. One common situation is that of modeling systems that utilize Mediator components to implement the interactions between Model components. In these cases the Mediator often carries out computations on a Model's native grid and performs regridding to the grid of other Model components. It is both cumbersome and error prone to recreate the Model grid in the Mediator. To solve this problem, NUOPC implements a transfer protocol for `ESMF_Grid`, `ESMF_Mesh`, and `ESMF_LocStream` objects (generally referred to as GeomObjects) between Model and/or Mediator components during initialization.

The NUOPC Layer transfer protocol for GeomObjects is based on two Field attributes: `TransferOfferGeomObject` and `TransferActionGeomObject`. The `TransferOfferGeomObject` attribute is used by the Model and/or Mediator components to indicate for each Field their intent for the associated GeomObject. The predefined values of this attribute are: "will provide", "can provide", and "cannot provide". The `TransferOfferGeomObject` attribute must be set during `label_Advertise`.

The generic Connector uses the intents from both sides and constructs a response according to the table below. The Connector's response is available during `label_RealizeProvided`. It sets the value of the `TransferActionGeomObject` attribute to either "provide" or "accept" on each Field. Fields indicating `TransferActionGeomObject` equal to "provide" must be realized on a Grid, Mesh, or LocStream object in the Model/Mediator before returning from `label_RealizeProvided`.

Fields that hold "accept" for the value of the `TransferActionGeomObject` attribute require two additional negotiation steps. During `label_AcceptTransfer` the Model/Mediator component can access the transferred

Grid/Mesh/LocStream on the Fields that have the "accept" value. However, only the DistGrid, i.e. the decomposition and distribution information of the Grid/Mesh/LocStream is available at this stage, not the full physical grid information such as the coordinates. At this stage the Model/Mediator may modify this information by replacing the DistGrid object in the Grid/Mesh/LocStream. The DistGrid that is set on the Grid/Mesh/LocStream objects when leaving the Model/Mediator phase `label_AcceptTransfer` will consequently be used by the generic Connector to fully transfer the Grid/Mesh/LocStream object. The fully transferred objects are available on the Fields with "accept" during Model/Mediator phase `label_RealizeAccepted`, where they are used to realize the respective Field objects. At this point all Field objects are fully realized and the initialization process can proceed as usual.

The following table shows how the generic Connector sets the `TransferActionGeomObject` attribute on the Fields according to the incoming value of `TransferOfferGeomObject`.

`TransferOfferGeomObject` **Incoming side A**	`TransferOfferGeomObject` **Incoming side B**	**Outgoing setting by generic Connector**
"will provide"	"will provide"	A:`TransferActionGeomObject`="provide" B:`TransferActionGeomObject`="provide"
"will provide"	"can provide"	A:`TransferActionGeomObject`="provide" B:`TransferActionGeomObject`="accept"
"will provide"	"cannot provide"	A:`TransferActionGeomObject`="provide" B:`TransferActionGeomObject`="accept"
"can provide"	"will provide"	A:`TransferActionGeomObject`="accept" B:`TransferActionGeomObject`="provide"

"can provide"	"can provide"	if (A is import side) then
		A:`TransferActionGeomObject`="provide"
		B:`TransferActionGeomObject`="accept"
		if (B is import side) then
		A:`TransferActionGeomObject`="accept"
		B:`TransferActionGeomObject`="provide"

"can provide"	"cannot provide"	A:`TransferActionGeomObject`="provide" B:`TransferActionGeomObject`="accept"
"cannot provide"	"will provide"	A:`TransferActionGeomObject`="accept" B:`TransferActionGeomObject`="provide"
"cannot provide"	"can provide"	A:`TransferActionGeomObject`="accept" B:`TransferActionGeomObject`="provide"
"cannot provide"	"cannot provide"	Flagged as error!

2.4.8 Field and Grid/Mesh/LocStream Reference Sharing

For coupling scenarios with a very high coupling frequency, or for situations where large data volumes are exchanged (e.g. 3D volumetric fields), it can be necessary for fields and geom objects (Grid, Mesh, and LocStreams) to share their data via references. Reference sharing greatly reduces the coupling cost compared to local or remote copies.

In the current implementation, in order for NUOPC components to be coupled via reference sharing, they must only have data defined (i.e. have DEs) on PETs that are part of both components. Further, the distribution of data across the PETs must be identical for both components. If these conditions are met, and both sides of the connection indicate that they are willing to participate in reference sharing, the NUOPC Connector will handle technical details. The Connector will provide fields to the components that reference the exact same data allocations in memory. Notice however that once reference sharing is active, the NUOPC Layer cannot protect against components violating the data access conventions. Specifically fields in the importState are not to be modified by the component. Reference sharing requires a higher level of "trust" between the components. NUOPC therefore requires that both sides of a connection

agree to reference sharing.

A component uses the SharePolicyField and SharePolicyGeomObject attributes on each field to indicate whether it is willing to reference share the data of a field, and/or the geom object on which the field is built. A setting of share indicates a component's willingness to share, while not share indicates the opposite. The share policy attributes are automatically set when a field is advertised via the NUOPC_Advertise() method. By default this method sets both share policies to not share.

When a Connector negotiates the connections between two components, it first considers the transfer offer attributes (i.e. TransferOfferGeomObject) on both sides for each field to determine the TransferActionGeomObject attribute for both side. The details of this protocol are outline in section 2.4.7. There are two cases to consider for each field that are relevant for reference sharing:

The simple case is where the Connector determines that for a specific field both sides must provide the field and geom object. This is indicated by TransferActionGeomObject being set to provide on both sides. For this case the ShareStatusField and ShareStatusGeomObject attributes are automatically set to not shared for all the fields, preventing any reference sharing.

The more interesting case is where one side of the connection receives the TransferActionGeomObject on a field set to provide, while the other side receives accept. In this case, the next step is for the Connector to take the SharePolicyField and SharePolicyGeomObject attributes on both sides into consideration. For each of the two attributes separately, if one side indicates not share, both sides will receive the associated ShareStatus set to not shared. However, if both sides of the connection indicate a SharePolicy of share, the Connector must further inspect the petLists to see if reference sharing is possible for the specific field. Under the current implementation a field is sharable with another component if all the PETs on which the field holds DEs are also in the other component's petList. If this condition is not met for the specific field, then the associated ShareStatus is set to not shared. Otherwise the ShareStatus is set to shared

During later phases of the Initialization protocol the Connector performs different operations, depending on how the TransferActionGeomObject, ShareStatusField, and ShareStatusGeomObject attributes were set as per the above protocol:

- For a field that has ShareStatusGeomObject equal to share, the geom object provided by the provider component will be made available to the acceptor component.
- For a field that has ShareStatusField equal to share, the Connector realizes the field for the acceptor component using the data allocation reference provided by the field of the provider component.

2.4.9 Field Mirroring

In some cases it is useful for a NUOPC component to match the set of fields advertised by another component, e.g. in order to connect to every field. NUOPC provides the concept of *field mirroring* that allows automatic matching by "mirroring" the fields of another component in their import- or exportState into their own States. One purpose of this is to automatically resolve the import data dependencies of a component, by setting up a component that exactly provides all of the needed fields.

The field mirror capability is also useful with NUOPC Mediators since these components often exactly reflect, in separate States, the sets of fields of each of the connected components. The field mirroring capability, therefore, can be used to ensure that a Mediator is always capable of accepting fields from connected components, and removes the need to specify field lists in multiple places, i.e., both within a set of Model components connected to a Mediator and within the Mediator itself.

To access the field mirror capability, a component sets the FieldTransferPolicy attribute during label_Advertise. The attribute is set on the Import- and/or Export- States to trigger field mirroring for each

state, respectively. The default value of "transferNone" indicates that no fields should be mirrored. The other available options are "transferAll" and "transferAllWithNamespace". Both options mirror transfer all of the fields from all of the connected States into the State that carries the attribute. The "transferAll" option results in flat structure with all of the mirrored fields added directly to the acceptor State. A flat structure like this is typically the preferred situation for an ExportState, where the same fields might be connected to multiple consumer components. The "transferAllWithNamespace" option also mirrors all of the field from the connected State, but creates separate Namespaces for each connection, placing the associated mirrored fields into the respective nested State. A nested structure like this useful for an ImportState where connections are being made with multiple producer components. In this case the consumer component can query the "Namespace" attribute of each nested State to infer the component label of the associated producer components.

Each Connector considers the `FieldTransferPolicy` attribute on both its import and export States. Each State that has the `FieldTransferPolicy` attribute set to "transferAll" or "transferAllWithNamespace", will have then fields of the respecive other State mirror transferred. If *both* States have the `FieldTransferPolicy` attribute set to trigger the mirror transfer, then fields are mirrored in both directions (i.e. import to export and export to import).

The transfer process works as follows: First, the `TransferOfferGoemObject` attribute is reversed between the providing side and accepting side. This is because if a field from the providing component is to be mirrored and it *can* provide its own geometric object, then the mirrored field on the accepting side should be set to *accept* a geometric object. The mirrored field is advertised in the accepting State using a call to `NUOPC_Advertise()` such that the mirrored field shares the same StandardName.

Components have the opportunity to modify or remove any of the mirrored Fields in their Import/ExportState by using the `label_ModifyAdvertised` specialization point. After this point the initialization sequence continues as usual. Since the mirrored fields have been advertised with matching StandardName attribute, the field pairing algorithm now matches them in the usual manner, thereby establishing a connection between the original and the mirrored fields.

2.5 Timekeeping

The NUOPC Layer associates an internal clock with three of its four generic component kinds: `NUOPC_Driver`, `NUOPC_Model`, and `NUOPC_Mediator`. The `NUOPC_Connector` is the only NUOPC component kind that does not have an internal clock object that is managed by NUOPC.

The component internal clocks are implemented as `ESMF_Clock` objects. The interaction beween these clock objects between a parent component (driver) and its child components (models, mediators, and drivers) is defined by the NUOPC timekeeping behavior described below.

For a simple run sequence with only a single coupling time-step, the driver clock sets the `startTime`, `stopTime`, and `timeStep` to be the beginning, the end, and the coupling period of the run, respectively. At the beginning of executing the run sequence, the driver clock `currTime` is set to its `startTime`. As the driver component executes the run sequence, it passes its clock to each child component that it executes. At the end of each full sweep through the run sequence the driver `currTime` is incremented by `timeStep` (i.e. the coupling period). This continues until the driver clock `stopTime` has been reached, and the run is complete.

When a child component is being called during the execution of the driver run sequence, it receives the driver/parent clock. This access is read-only, and the child component is only allowed to inspect but not modify the parent clock. The child component is expected to run forward a single coupling period, i.e. one `timeStep` on the parent clock. Specifically this means that the `currTime` on the child clock must match the `currTime` on the parent clock. It then must take a single `timeStep` of the parent clock forward, using its own clock to do so. The child component can implement this forward step by taking multiple smaller advances on its own clock.

The generic NUOPC component implementation provides the following assistance to implement the above described

behavior:

- During initialization of a component, its clock is set as a copy of its parent clock. Specifically the settings for `startTime`, `stopTime`, `timeStep`, and `currTime` are propagated. Alarms are not propagated.
- A component can customize aspects of its clock during initialization by using the `label_SetClock` specialization point.
- During run time, the default `label_SetRunClock` specialization checks that the `currTime` matches between child and parent clock. It further checks that the child clock can reach the parent's `currTime`+`timeStep`, i.e. the next coupling time, by an integral number of it's own time steps. If so, the `stopTime` on the child clock is set to the parent's `currTime`+`timeStep`.
  - It can be useful to customize `label_SetRunClock`, e.g. if the parent uses dynamic coupling periods, or in case of a run sequence with multiple coupling periods. In these cases the component must react to the parent `timeStep` provided during execution of the run sequence. In general the `currTime` match should be implemented, followed by setting the child's `timeStep` according to the information provided on the parent clock. Finally the the `stopTime` on the child clock should be set as to return at the next coupling time determined by the parent clock.
- Once past the `label_SetRunClock` specialization, the component checks the timestamps on the fields in the import state. This is done by calling into the `label_CheckImport` specialization point. The default implementation simply checks that all import fields are at `currTime` of the child clock.
  - In more complex situations, where the interaction between different components happens with different coupling periods, it can be necessary to specialialize the `label_CheckImport` of a component. For example, a component might receive fields in its import state that carry different timestamps. Consequently, `label_CheckImport` must implement a more complex relationship between the component's `currTime`, and the timestamps on each import field.
- Finally the component clock is stepped forward from `currTime` to `stopTime`, using the `timeStep` interval set in the child clock. During this loop, the `label_Advance` specialization is called for each time step. The `label_Advance` specialization is responsible for any accumulating and averaging that may be necessary.
  - In practice often the `timeStep` on the child clock is chosen to be identical to that of the parent clock. This way the `label_Advance` specialization is only called once for every coupling period. In this approach the details about potentially smaller model time steps, and associated accumulation and averaging is handled below the NUOPC cap layer of a model.
- After the `stopTime` has been reached on the child clock, the `label_TimestampExport` specialization point is called before the component returns to the parent. The default implementation simply timestamps all the fields in the export state with the `currTime` of of the child clock.

2.6 Component Hierarchies

The NUOPC Layer supports component hierarchies. The key function to support this capability is the ability for a generic `NUOPC_Driver` to add another `NUOPC_Driver` component as a child, and to drive it much like a `NUOPC_Model` component. The interactions upward and downward the hierarchy tree are governed by the standard NUOPC component interaction protocols.

In the current implementation, data-dependencies during initialization can be resolved throughout the entire component hierarchy. The implementation is based on a sweep algorithm that continues up and down the hierarchy until either all data-dependencies have been resolved, or a dead-lock situation has been detected and flagged.

Along the downward direction, the interaction of a driver with its children allows the driver to mirror its child components' fields, and to transfer or share geom objects and fields up the component hierarchy. All of the interactions of a driver with its child components are handled by explicit `NUOPC_Connector` instances. These instances are automatically added by the driver when needed.

The detailed behavior of a `NUOPC_Driver` component within a component hierarchy depends on the setting of the `HierarchyProtocol` attribute on the driver component itself. Section 2.3.1 lists all of the driver attributes defined by NUOPC. By default the `HierarchyProtocol` attribute is unset. For unset `HierarchyProtocol` or when set to `PushUpAllExportsAndUnsatisfiedImports`, the driver component pushes all the fields from its children exportStates into its own exportState, and all unsatisfied fields in its children importStates into its own importState. This is done using the standard Field Mirroring protocol discussed under 2.4.9. Further the driver sets the `SharePolicyGeomObject`, and `SharePolicyField` to `share` for all the fields it mirrors. This triggers the reference share protocol as described in section 2.4.8.

When the `HierarchyProtocol` is set to `Explorer`, the driver component still mirrors the fields from its child components' import- and exportStates, as was done for the default, however, the share policies will not be set. This protocol option is used by the NUOPC ComponentExplorer to connect to user provided components.

Finally, for a setting of `HierarchyProtocol` to `ConnectProvidedFields`, the driver does not modify its own import- and exportState. Instead connections are made only between fields that have been added to the driver states externally. This is useful for the situation where a `NUOPC_Driver` component is called directly via ESMF component method from a level that is outside of NUOPC. In this situation, field and/or geom object sharing must be activated explicitly if desired.

2.7 Resource Control and Threaded Components

Each instance of a NUOPC component within an application is defined on a fixed set of compute resources. The association of resources occurs when the component is added to its parent component via the `NUOPC_DriverAddComp()` call. Subsequently when any of the component's Initialize, Run, or Finalize phases is called, the component code executes on the associated resources.

The primary control of resource management under NUOPC is implemented through the `petList` argument that is accepted by `NUOPC_DriverAddComp()`. This argument holds a list of Persistent Execution Thread (PET) ids of the parent component on which the child component is to execute. By default, i.e. when `petList` is *not* specified, *all* of the parent PETs are associated with the added child component. Using custom `petList` constructions, a driver has control of exactly how its child components are sharing the available PET resources.

Notice that the *order* of PETs listed in a `petList` is significant. The local PET labeling inside a child component always goes from `0` to `size(petList)-1`. The order in which the child PETs correspond to the parent PETs is that specified by the `petList`. It is erroneous to list the same parent PET multiple times in the *same* `petList` argument.

For the following discussion it is convenient to think of PETs as simple MPI processes. While this is not strictly correct on a technically ESMF level, there are currently no features available to NUOPC where this interpretation would lead to inconsistencies. One of the key consequences of equating each PET to a simple MPI process is that each PET can only execute a single component's code at any given time. Therefore, in order to allow components to execute concurrently, a necessary condition is to define them on exclusive petLists. Of course the data dependencies between components must also support concurrent execution. Often this requires careful placement of Connectors in the run sequence and the introduction of time lags. However, this is more of a scientific than the resource control question covered in this section.

Many model components today implement the hybrid MPI+OpenMP paradigm to support scalability to larger core counts than would be possible in a purely MPI or OpenMP approach. NUOPC supports hybrid MPI+OpenMP components in two ways: NUOPC *aware* and NUOPC *unaware*. In the NUOPC *unaware* approach, the application is

launched only on those MPI ranks that are going to participate in the hybrid execution with OpenMP. Usually this means that the MPI launch system (mpirun, mpiexec, aprun, srun, etc.), and a set of environment variables get involved in correctly associating the desired number of hardware cores with each MPI process, and to assure correct affinities. In this approach NUOPC is not at all involved in the resource management, and OpenMP threading happens purely on the user level.

The NUOPC *unaware* hybrid MPI+OpenMP approach provides a quick way to run hybrid applications that consist of a single model component, or where all of the model components use the same hybrid approach with the same ratio of OpenMP threads per MPI rank. In this case, shell-based user level resource control is often sufficient. However, for more complex coupling scenarios the NUOPC *aware* hybrid approach provides additional levels of control that are often needed to achieve optimal utilization of the available resources

Under the NUOPC *aware* resource control, some components might be purely MPI based, while others use the hybrid approach. Different hybrid components can be configured to run with different threading levels. This is possible independent on whether the components use the same or exclusive sets of resources.

Besides the already discussed `petList` argument, there are two additional optional arguments to `NUOPC_DriverAddComp()`. It is through those arguments that the advanced resource control features under NUOPC are implemented. One of these arguments is `compSetVMRoutine`. This argument allows the user to point to a specific public method of the child component. The signature of this method is the same as for the `compSetServicesRoutine` argument. If `compSetVMRoutine` is provided, it will be called *before* `compSetServicesRoutine`. The purpose of `compSetVMRoutine` is to allow the child component to set specific aspects of its own ESMF virtual machine (VM) before instantiating it. The ESMF reference manual discusses the details of this procedure under the "User-code SetVM method" section. Based on the information provided there, a user could implement a custom `compSetVMRoutine` method for a component. However, for convenience, NUOPC provides a generic implementation that can be passed into `compSetVMRoutine`. For most common situation, the generic implementation provided by NUOPC is sufficient, and there is no need for the user to provide a custom implementation of `compSetVMRoutine`.

Utilizing the generic `SetVM` method provided by NUOPC involves a few steps. First, the component implementation must make the generic `SetVM` *public* inside its own *cap* module:

```
module MODEL

  !-------------------------------------------------------------------------
  ! MODEL Component.
  !-------------------------------------------------------------------------

  use ESMF
  use NUOPC
  use NUOPC_Model, &
    modelSS    => SetServices

  implicit none

  private

  public SetVM, SetServices   ! Here making SetVM and SetServices public.

  !-------------------------------------------------------------------------
  contains
  !-------------------------------------------------------------------------
  ...
```

```
end module
```

Second, the driver component that adds MODEL via `NUOPC_DriverAddComp()` as a child component, must make a `USE` association to the `SetVM`:

```
module driver

  !-----------------------------------------------------------------------------
  ! Code that specializes generic NUOPC_Driver
  !-----------------------------------------------------------------------------

  use MPI
  use ESMF
  use NUOPC
  use NUOPC_Driver, &
    driverSS             => SetServices

  use MODEL, only: &
    modelSS     => SetServices, &
    modelSVM    => SetVM            ! Here making USE association to SetVM.

  implicit none

  private

  public SetServices

  !-----------------------------------------------------------------------------
  contains
  !-----------------------------------------------------------------------------
  ...
end module
```

Third, the driver can now pass the `modelSVM` into `NUOPC_DriverAddComp()` via the `compSetVMRoutine` argument, essentially providing the generic `SetVM` method.

Finally, the generic `SetVM` implementation needs to be informed about the specific resource control request. This is handled through *the other* optional argument to `NUOPC_DriverAddComp()` alluded to earlier. This is the `info` argument.

The `info` argument is of `type(ESMF_Info)`, which implements a structured key/value pair class. An `info` object must first be created via `ESMF_InfoCreate()` before any key/value pairs can be set.

```
    type(ESMF_Info)               :: info
    ...
    info = ESMF_InfoCreate(rc=rc)
    ! check rc
```

NUOPC resource control is implemented under the `/NUOPC/Hint/PePerPet` *structure*. The following table documents the available *keys* under this structure, the supported *values*, and their meaning. Notice that *structure* and *keys* are case sensitive, while *values* are case insensitive.

key	value	Meaning
MaxCount	Positive integer	The *maximum* number of Processing Elements (PEs), i.e. cores or hardware threads, associated with each child PET. The procedure is this: the PEs associated with the incoming parent PETs (e.g. via petList), are grouped by single system image (SSI), i.e. shared memory domain or hardware node. Within each SSI the PEs are divided by the MaxCount to determine how many child PETs are needed for each SSI. The PEs on each SSI are then associated with the child PETs. Note that this procedure only then results in every child PET holding exactly MaxCount PEs when the number of PEs per SSI brought in by the parent PETs is a *multiple* of MaxCount. Parent PETs that for the child VM gave up their PEs, and are not executing as child PETs, are paused for the duration of the child component execution. They resume execution under the parent VM once the child component returns control to the parent.
OpenMpHandling	String: *none*, *set*, *init*, or *pin* (the default)	For "none", OpenMP handling is completely left to the user. In this case the user child component code will typically want to query the child VM for the local number of PEs under each child PET. This number then would be used in an explicit call to omp_set_num_threads() in order to set the OpenMP thread number according to the available PEs under each child PET. For "set", the NUOPC/ESMF layer make the call to omp_set_num_threads() under each child PET with the appropriate number of PEs. For "init", the NUOPC/ESMF layers sets the number of OpenMP threads in each team, and triggers the instantiation of all the threads in the team. For "pin", the NUOPC/ESMF layers sets the number of OpenMP threads in each team, triggers the instantiation of the team, and pins each OpenMP thread to the corresponding PE.
OpenMpNumThreads	Positive integer	By default the "set", "init", or "pin" option under OpenMpHandling sets the number of OpenMP threads in each team equal to the number of PEs under each PET. Setting OpenMpNumThreads, this default can be overwritten. The option allows the user to under- or oversubscribe the PEs held by each PET.
ForceChildPthreads	Logical: .true., or .false. (the default)	By default (.false.) each PET executes under the same thread as its parent PET. Typically this means that PETs execute directly as the MPI process under which they were created. In some cases it is beneficial to create a separate Pthread for each child PET. This can be accomplished by setting the value to .true..

PthreadMinStackSize	Positive integer	The minimum stack size in *byte* of each child PET that is executing as Pthread. By default child PETs do *not* execute as Pthreads. Therefore the stack size by default is equal to that of the parent PET. However, if ForceChildPthreads is set to true, all child PETs are instantiated as Pthreads. This means that the stack size *cannot* be *unlimited*. ESMF implements a default minimum stack size for child PETs of 20MiB. This minimum default can be changed (up or down) via the PthreadMinStackSize key. The system limit or ulimit commands can be used to further *increase* the stack size of child PETs. Any limit set lower than the PthreadMinStackSize, or set to *unlimited*, will result in usage of the PthreadMinStackSize if set, or the 20MiB default. Note further that when OpenMP is used inside the child component, each child PET becomes the root thread of each of the OpenMP thread teams. It is therefore the root thread stack size that is affected by PthreadMinStackSize. The stack size of all the *other* OpenMP threads in each team is set via environment variable OMP_STACKSIZE as usual.

The following code snippet demonstrates a typical resource control request using the generic SetVM routine and an info object. This request is suitable for a hybrid MPI+OpenMP component where every child PET is expected to run 4-way OpenMP threaded.

```
call ESMF_InfoSet(info, key="/NUOPC/Hint/PePerPet/MaxCount", value=4, rc=rc)
! check rc
call NUOPC_DriverAddComp(driver, "MODEL1", modelSS, modelSVM, info=info, rc=rc)
! check rc
```

A second child component can be created that uses the same parent resources as the first, but sets up 8-way OpenMP threading under each child PET.

```
call ESMF_InfoSet(info, key="/NUOPC/Hint/PePerPet/MaxCount", value=8, rc=rc)
! check rc
call NUOPC_DriverAddComp(driver, "MODEL2", modelSS, modelSVM, info=info, rc=rc)
! check rc
```

If the default settings for some of the keys are not appropriate, they can be set explicitly. Here for instance a child component with the same number of PETs as the previous 4-way OpenMP threaded case is created, but is instructed to not handle any of the OpenMP aspects.

```
call ESMF_InfoSet(info, key="/NUOPC/Hint/PePerPet/MaxCount", value=4, rc=rc)
! check rc
call ESMF_InfoSet(info, key="/NUOPC/Hint/PePerPet/OpenMpHandling", &
  value="none", rc=rc)
```

```
! check rc
call NUOPC_DriverAddComp(driver, "MODEL3", modelSS, modelSVM, info=info, rc=rc)
! check rc
```

In this example, all three child components "MODEL1", "MODEL2", and "MODEL3" use the exact same parent resources. Due to this fact all three components can only execute sequentially. However, each child component manages the resources provided by the parent differently, and independently. Through this tailored approach, NUOPC allows optimal use of the available resources by each component. `NUOPC_Connector` components defined between components work as usual, taking care of all the required data movements automatically and completely transparent to the user.

In order to obtain best performance when using NUOPC *aware* resource control for hybrid parallelism, it is *strongly recommended* to set `OMP_WAIT_POLICY=PASSIVE` in the environment. This is one of the standard OpenMP environment variables. The `PASSIVE` setting ensures that OpenMP threads relinquish the hardware threads (i.e. cores) as soon as they have completed their work. Without that setting ESMF resource control threads can be delayed, and context switching between components becomes more expensive.

2.8 External NUOPC Interface

Complete applications can easily be built by assembling NUOPC compliant components. Many such NUOPC applications are in productive use across several institutions. The top level of such applications is typically implemented via a very thin application layer holding the main program that calls into the top level driver component that derives from `NUOPC_Driver`. Model components sit under the top level driver, interacting with one another and the driver through the NUOPC protocols. Complex systems have one or more component hierarchy levels under the top level driver as discussed in the previous section.

There are situation, however, where a NUOPC application needs to be controlled by an outside component. Such an outside component does not derive from any of the generic NUOPC components, and cannot be expected to implement the complete NUOPC protocol. Typically such an external component implements its own control structure outside of NUOPC and ESMF. One example of such a situation are data assimilation systems that want to drive a NUOPC forecast application.

In order to facilitate the external access into a NUOPC application, the `NUOPC_Driver` provides an *external interface*. This interface is implemented through the standard ESMF component methods: Initialize, Run, and Finalize. This interface with the top level NUOPC driver allows an external component to control and interact with the entire NUOPC application.

The standard ESMF component interfaces hold `importState`, `exportState`, and a `clock` argument. These arguments are used to pass data in and out of the NUOPC application, and control the time stepping of the NUOPC model, respectively. The top level driver of a NUOPC application has access to any field that is advertised by any of the components and therefore serves as a single point of access for the entire application.

The external NUOPC interface is currently defined by the Initialize, Run, and Finalize phases documented in the following table.

methodFlag	**phaseLabel**	**Meaning**
`ESMF_METHOD_INITIALIZE`	label_ExternalAdvertise	Called after the external component has set up the import- and exportStates with fields (advertised) that it plans to interact with. On the NUOPC application side this call will got through the complete advertise cylce.

ESMF_METHOD_INITIALIZE	label_ExternalRealize	Called after the external component has been informed about the connected status of the fields in the import- and exportState. On the NUOPC application side this call will finish setting up RouteHandles between all components involved.
ESMF_METHOD_INITIALIZE	label_ExternalDataInit	Trigger a complete data initialize throughout the NUOPC application. The expectation is that all components reset their data consistent with the clock argument.
ESMF_METHOD_RUN		The default Run() method steps the NUOPC application forward in time according to the clock argument.
ESMF_METHOD_FINALIZE	label_ExternalReset	Inform the NUOPC application about a clock reset.
ESMF_METHOD_FINALIZE		Completely finalize and shut down the NUOPC application.

Here methodFlag and phaseLabel corrsepond to the respective arguments of method NUOPC_CompSearchPhaseMap(). This method is used to determine the actual ESMF phase index needed when calling into ESMF_GridCompInitialize(), ESMF_GridCompRun(), or ESMF_GridCompFinalize(). In cases where no phaseLabel is indicated, the default phase is used for the implementation, accessible by not specifying the argument.

For a complete example of how the *External NUOPC API* is used in practice, see https://github.com/esmf-org/nuopc-app-prototypes/tree/develop/ExternalDriverAPIProto. The following code snippets demonstrates the critical pieces of code from the external layer interacting with NUOPC/ESMF.

```
! Create the external level import/export States
! NOTE: The "stateintent" must be specified, and it must be set from the
! perspective of the external level:
! -> state holding fields exported by the external level to the ESM component
externalExportState = ESMF_StateCreate(stateintent=ESMF_STATEINTENT_EXPORT, rc=rc)
  ! check rc
! -> state holding fields imported by the external level from the ESM component
externalImportState = ESMF_StateCreate(stateintent=ESMF_STATEINTENT_IMPORT, rc=rc)
  ! check rc

! Advertise field(s) in external import state to receive from the NUOPC layer
call NUOPC_Advertise(externalImportState, &
  StandardNames=(/"sea_surface_temperature"/), &
  TransferOfferGeomObject="cannot provide", SharePolicyField="share", rc=rc)
  ! check rc

! Call "ExternalAdvertise" Initialize for the earth system Component
call NUOPC_CompSearchPhaseMap(nuopcApp, methodflag=ESMF_METHOD_INITIALIZE, &
  phaseLabel=label_ExternalAdvertise, phaseIndex=phase, rc=rc)
  ! check rc
call ESMF_GridCompInitialize(nuopcApp, phase=phase, clock=clock, &
  importState=externalExportState, exportState=externalImportState, userRc=urc, rc=rc)
  ! check rc and urc

! Call "ExternalRealize" Initialize for the earth system Component
```

```
call NUOPC_CompSearchPhaseMap(nuopcApp, methodflag=ESMF_METHOD_INITIALIZE, &
  phaseLabel=label_ExternalRealize, phaseIndex=phase, rc=rc)
  ! check rc
call ESMF_GridCompInitialize(nuopcApp, phase=phase, clock=clock, &
  importState=externalExportState, exportState=externalImportState, userRc=urc, rc=rc)
  ! check rc and urc

! Call "ExternalDataInit" Initialize for the earth system Component
call NUOPC_CompSearchPhaseMap(nuopcApp, methodflag=ESMF_METHOD_INITIALIZE, &
  phaseLabel=label_ExternalDataInit, phaseIndex=phase, rc=rc)
  ! check rc
call ESMF_GridCompInitialize(nuopcApp, phase=phase, clock=clock, &
  importState=externalExportState, exportState=externalImportState, userRc=urc, rc=rc)
  ! check rc and urc

! Explicit time stepping loop on the external level, here based on ESMF_Clock
do while (.not.ESMF_ClockIsStopTime(clock, rc=rc))
  ! Run the earth system Component: i.e. step ESM forward by timestep
  call ESMF_GridCompRun(nuopcApp, clock=clock, &
    importState=externalExportState, exportState=externalImportState, userRc=urc, rc=rc
  ! check rc and urc
  ! Advance the clock
  call ESMF_ClockAdvance(clock, rc=rc)
  ! check rc
end do

! Finalize the earth system Component
call ESMF_GridCompFinalize(nuopcApp, clock=clock, &
  importState=externalExportState, exportState=externalImportState, userRc=urc, rc=rc)
  ! check rc and urc
```

3 API

3.1 Generic Component: NUOPC_Driver

MODULE:

```
module NUOPC_Driver
```

DESCRIPTION:
Component that drives and coordinates initialization of its child components: Model, Mediator, and Connector components. For every Driver time step the same run sequence, i.e. sequence of Model, Mediator, and Connector `Run` methods is called. The run sequence is fully customizable. The default run sequence implements explicit time stepping.

SUPER:

```
ESMF_GridComp
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

```
subroutine SetServices(driver, rc)
  type(ESMF_GridComp)   :: driver
  integer, intent(out)  :: rc
```

SEMANTIC SPECIALIZATION LABELS:

- Initialize:
  - **label_SetModelServices**
    - Optional. By default driver has no child components.
    - Use `NUOPC_DriverAddComp()` repeatedly to add child components to the driver.
    - Use `NUOPC_CompAttributeSet()` or `NUOPC_CompAttributeIngest()` to set attributes on child components.
    - Create and set driver clock with startTime, stopTime, and timeStep, if not done by the driver's parent.
  - **label_SetRunSequence**
    - Optional. By default drive child components in the sequence they were added.
    - Define and set a RunSequence either by calling `NUOPC_DriverIngestRunSequence()`, or by using the `NUOPC_DriverNewRunSequence()` and `NUOPC_DriverAddRunElement()` API.
  - **label_ModifyInitializePhaseMap**
    - Optional. By default InitializePhaseMap attributes are not modified.
    - Modify the InitializePhaseMap attribute on the child components as desired. This is very rarely needed.
  - **label_ModifyCplLists**
    - Optional. By default CplList attributes are not modified.

        * Modify the CplList attribute on the child components as desired. This can be useful to set custom Connection Options for specific Field pairs.
    - **label_PreChildrenAdvertise**
        * Optional.
        * Allow driver to execute specific code before calling the Advertise phase of its children.
    - **label_PostChildrenAdvertise**
        * Optional.
        * Allow driver to execute specific code after calling the Advertise phase of its children.
    - **label_PreChildrenRealize**
        * Optional.
        * Allow driver to execute specific code before calling the Realize phase of its children.
    - **label_PostChildrenRealize**
        * Optional.
        * Allow driver to execute specific code after calling the Realize phase of its children.
    - **label_PreChildrenDataInitialize**
        * Optional.
        * Allow driver to execute specific code before calling the DataInitialize phase of its children.
    - **label_PostChildrenDataInitialize**
        * Optional.
        * Allow driver to execute specific code after calling the DataInitialize phase of its children.
- Run:
    - **label_SetRunClock**
        * Optional. By default driver clock is left unchanged if the parent component has no valid clock. If there is a valid parent clock, the current time is checked between it and the driver clock. An error is returned if the current time does not agree. Otherwise (current time does agree between both clocks), the driver clock stop time is adjusted to a single time step of the parent clock in the future. This ensures that the driver returns at the appropriate parent time step, even if that might change dynamically during the run.
        * Modify the driver clock before executing RunSequence. This is very rarely needed.
    - **label_ExecuteRunSequence**
        * Optional. By default use NUOPC generic RunSequence execution.
        * Implement a custom RunSequence execution. This is very rarely needed.
- Finalize:
    - **label_Finalize**
        * Optional. By default do nothing.
        * Destroy any objects created during Initalize.

3.2 Generic Component: NUOPC_ModelBase

MODULE:

```
module NUOPC_ModelBase
```

DESCRIPTION:

Partial specialization of a component with a default *explicit* time dependency. Each time the `Run` method is called the component steps one timeStep forward on the passed in parent clock. The component flags incompatibility during `Run` if the current time of the incoming clock does not match the current time of the internal clock.

SUPER:

```
ESMF_GridComp
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

```
subroutine SetServices(modelBase, rc)
  type(ESMF_GridComp)   :: modelBase
  integer, intent(out)  :: rc
```

SEMANTIC SPECIALIZATION LABELS:

- Initialize:
  - **label_Advertise**
    - Required in order to advertise fields.
    - Use `NUOPC_Advertise()` to advertise specific fields in the Import- and ExportState of the component.
    - Alternatively set the FieldTransferPolicy attribute on the Import- and ExportState of the component to request field mirroring.
  - **label_ModifyAdvertised**
    - Optional. By default do not modify the advertised fields.
    - Mostly used when field mirroring was requested during Advertise.
    - Remove undesired advertised fields in the Import- and ExportState of the component.
    - Adjust attributes e.g. for TransferOffer on advertised fields.
  - **label_RealizeProvided**
    - Required in order to realize fields.
    - Use `NUOPC_Realize()` to realize fields previously advertised, and for which this component is responsible for providing the Field allocation and/or the GeomObject.
  - **label_AcceptTransfer**
    - Optional. By default accept the Distribution of the transferred GeomObjects.
    - Change the distribution of any of the transferred GeomObjects.
  - **label_RealizeAccepted**
    - Optional. Needed for any fields for which component is accepting the GeomObject.
    - Use `NUOPC_Realize()` to realize fields previously advertised, and for which this component is accepting the GeomObject.
  - **label_SetClock**
    - Optional. By default create clock according to time information provided by driver.
    - Adjust and set the component clock.
  - **label_DataInitialize**

        * Optional. Needed to initialize data, and to participate in resolution of data dependencies between components during initialize.
        * Initialize data in fields.
        * Set NUOPC attributes used for data dependency resolution.

- Run:
  - **label_Advance**
    * Called every timeStep on the component internal clock.
    * Implement the forward integration of the model.
    * Ensure data in the export fields is updated before returning.
  - **label_AdvanceClock**
    * Optional. By default the component internal clock is advanced by one internal timeStep at the end of the Advance step.
  - **label_CheckImport**
    * Optional. By default check the timestamp of all import fields against the current time of the internal clock.
  - **label_SetRunClock**
    * Optional. By default do not adjust the internal clock when entering Run.
  - **label_TimestampExport**
    * Optinal. By default timestamp all export fields according to the current time of the component internal clock before returning.

- Finalize:
  - **label_Finalize**
    * Optional. By default do nothing.
    * Destroy any objects created during Initalize.

3.3 Generic Component: NUOPC_Model

MODULE:

```
module NUOPC_Model
```

DESCRIPTION:
Model component with a default *explicit* time dependency. Each time the `Run` method is called the model integrates one timeStep forward on the passed in parent clock. The internal clock is advanced at the end of each `Run` call. The component flags incompatibility during `Run` if the current time of the incoming clock does not match the current time of the internal clock.

SUPER:

```
NUOPC_ModelBase
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

```
subroutine SetServices(model, rc)
  type(ESMF_GridComp)   :: model
  integer, intent(out)  :: rc
```

SEMANTIC SPECIALIZATION LABELS:

- Initialize:
  - **label_Advertise**
    - Required in order to advertise fields.
    - Use `NUOPC_Advertise()` to advertise specific fields in the Import- and ExportState of the component.
    - Alternatively set the FieldTransferPolicy attribute on the Import- and ExportState of the component to request field mirroring.
  - **label_ModifyAdvertised**
    - Optional. By default do not modify the advertised fields.
    - Mostly used when field mirroring was requested during Advertise.
    - Remove undesired advertised fields in the Import- and ExportState of the component.
    - Adjust attributes e.g. for TransferOffer on advertised fields.
  - **label_RealizeProvided**
    - Required in order to realize fields.
    - Use `NUOPC_Realize()` to realize fields previously advertised, and for which this component is responsible for providing the Field allocation and/or the GeomObject.
  - **label_AcceptTransfer**
    - Optional. By default accept the Distribution of the transferred GeomObjects.
    - Change the distribution of any of the transferred GeomObjects.
  - **label_RealizeAccepted**
    - Optional. Needed for any fields for which component is accepting the GeomObject.
    - Use `NUOPC_Realize()` to realize fields previously advertised, and for which this component is accepting the GeomObject.
  - **label_SetClock**
    - Optional. By default create clock according to time information provided by driver.
    - Adjust and set the component clock.
  - **label_DataInitialize**
    - Optional. Needed to initialize data, and to participate in resolution of data dependencies between components during initialize.
    - Initialize data in fields.
    - Set NUOPC attributes used for data dependency resolution.
- Run:
  - **label_Advance**
    - Called every timeStep on the component internal clock.
    - Implement the forward integration of the model.
    - Ensure data in the export fields is updated before returning.
  - **label_AdvanceClock**

        * Optional. By default the component internal clock is advanced by one internal timeStep at the end of the Advance step.
    - **label_CheckImport**
        * Optional. By default check the timestamp of all import fields against the current time of the internal clock.
    - **label_SetRunClock**
        * Optional. By default do not adjust the internal clock when entering Run.
    - **label_TimestampExport**
        * Optinal. By default timestamp all export fields according to the current time of the component internal clock before returning.
- Finalize:
    - **label_Finalize**
        * Optional. By default do nothing.
        * Destroy any objects created during Initalize.

3.4 Generic Component: NUOPC_Mediator

MODULE:

```
module NUOPC_Mediator
```

DESCRIPTION:
Mediator component with a default *explicit* time dependency. Each time the `Run` method is called, the time stamp on the imported Fields must match the current time (on both the incoming and internal clock). Before returning, the Mediator time stamps the exported Fields with the same current time, before advancing the internal clock one timeStep forward.

SUPER:

```
NUOPC_ModelBase
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

```
subroutine SetServices(mediator, rc)
  type(ESMF_GridComp)   :: mediator
  integer, intent(out)  :: rc
```

SEMANTIC SPECIALIZATION LABELS:

- Initialize:
    - **label_Advertise**
        * Required in order to advertise fields.

    - Use `NUOPC_Advertise()` to advertise specific fields in the Import- and ExportState of the component.
    - Alternatively set the FieldTransferPolicy attribute on the Import- and ExportState of the component to request field mirroring.
  - **label_ModifyAdvertised**
    - Optional. By default do not modify the advertised fields.
    - Mostly used when field mirroring was requested during Advertise.
    - Remove undesired advertised fields in the Import- and ExportState of the component.
    - Adjust attributes e.g. for TransferOffer on advertised fields.
  - **label_RealizeProvided**
    - Required in order to realize fields.
    - Use `NUOPC_Realize()` to realize fields previously advertised, and for which this component is responsible for providing the Field allocation and/or the GeomObject.
  - **label_AcceptTransfer**
    - Optional. By default accept the Distribution of the transferred GeomObjects.
    - Change the distribution of any of the transferred GeomObjects.
  - **label_RealizeAccepted**
    - Optional. Needed for any fields for which component is accepting the GeomObject.
    - Use `NUOPC_Realize()` to realize fields previously advertised, and for which this component is accepting the GeomObject.
  - **label_SetClock**
    - Optional. By default create clock according to time information provided by driver.
    - Adjust and set the component clock.
  - **label_DataInitialize**
    - Optional. Needed to initialize data, and to participate in resolution of data dependencies between components during initialize.
    - Initialize data in fields.
    - Set NUOPC attributes used for data dependency resolution.

- Run:
  - **label_Advance**
    - Called every timeStep on the component internal clock.
    - Implement the forward integration of the model.
    - Ensure data in the export fields is updated before returning.
  - **label_AdvanceClock**
    - Optional. By default the component internal clock is advanced by one internal timeStep at the end of the Advance step.
  - **label_CheckImport**
    - Optional. By default check the timestamp of all import fields against the current time of the internal clock.
  - **label_SetRunClock**
    - Optional. By default do not adjust the internal clock when entering Run.
  - **label_TimestampExport**
    - Optinal. By default timestamp all export fields according to the current time of the component internal clock before returning.

- Finalize:
  - **label_Finalize**
    - Optional. By default do nothing.
    - Destroy any objects created during Initalize.

3.5 Generic Component: NUOPC_Connector

MODULE:

```
module NUOPC_Connector
```

DESCRIPTION:
Component that makes a unidirectional connection between model, mediator, and or driver components. During initialization field pairing is performed between the import and export side according to section 2.4.2, and paired fields are connected. By default the bilinear regrid method is used during `Run` to transfer data from the connected import Fields to the connected export Fields.

SUPER:

```
ESMF_CplComp
```

USE DEPENDENCIES:

```
use ESMF
```

SETSERVICES:

```
subroutine SetServices(connector, rc)
  type(ESMF_CplComp)    :: connector
  integer, intent(out)  :: rc
```

SEMANTIC SPECIALIZATION LABELS:

- Initialize:
  - **label_ComputeRouteHandle**
    - Optional. By default compute routehandles according to CplList attribute.
- Run:
  - **label_ExecuteRouteHandle**
    - Optional. By default execute routehandles stored in the Connector.
- Finalize:
  - **label_ReleaseRouteHandle**
    - Optional. By default release routehandles stored in the Connector.
  - **label_Finalize**
    - Optional. By default do nothing.
    - Destroy any objects created during Initalize.

3.6 General Generic Component Methods

3.7 Field Dictionary Methods

3.8 Free Format Methods

3.9 Utility Routines

3.10 Auxiliary Routines

Auxiliary routines are provided with the NUOPC Layer as a convenience to the user. Typically more work is needed on these methods before considering them NUOPC core functionality.

4 Standardized Component Dependencies

DEPRECATION NOTICE: The mechanism described in this section for defining build dependencies between components has been deprecated! The approach discussed here is based exclusively on GNU Makefiles. It has been superseded by the functionality implemented in the ESMX Layer. The ESMX approach addresses all of the issues discussed here, based on a more holistic solution. It includes a standard CMake based option, which is the recommended approach for all new NUOPC projects.

Most of the NUOPC Layer deals with specifying the interaction between ESMF components within a running ESMF application. ESMF provides several mechanisms of how an application can be made up of individual Components. This chapter deals with reigning in the many options supported by ESMF and setting up a standard way for assembling NUOPC compliant components into a working application.

ESMF supports single executable as well as some forms of multiple executable applications. Currently the NUOPC Layer only addresses the case of single executable applications. While it is generally true that executing single executable applications is easier and more widely supported than executing multiple executable applications, building a single executable from multiple components can be challenging. This is especially true when the individual components are supplied by different groups, and the assembly of the final application happens apart from the component development. The purpose of standardizing component dependencies as part of the NUOPC Layer is to provide a solution to the technical aspect of assembling applications built from NUOPC compliant components.

As with the other parts of the NUOPC Layer, the standardized component dependencies specify aspects that ESMF purposefully leaves unspecified. Having a standard way to deal with component dependencies has several advantages. It makes reading and understand NUOPC compliant applications more easily. It also provides a means to promote best practices across a wide range of application systems. Ultimately the goal of standardizing the component dependencies is to support "plug & build" between NUOPC compliant components and applications, where everything needed to use a component by a upper level software layer is supplied in a standard way, ready to be used by the software.

There is one aspect of the standardized component dependency that affects the component code itself: **The name of the public set services entry point into a NUOPC compliant component must be called "SetServices"**. The only exception to this rule are components that are written in C/C++ and made available for static linking. In this case, because of lack of namespace protection, the `SetServices` part must be followed by a component specific suffix. This will be discussed later in this chapter. For all other cases, unique namespaces exist that allow the entry point to be called `SetServices` across all components.

Having standardized the name of the single public entry point into a component solves the issue of having to communicate its name to the software layer that intends to use the component. At the same time, limiting the public entry point to a single accepted name does not remove any flexibility that is generally leveraged by ESMF applications. Within the context of the NUOPC Layer, there is great flexibility designed into the initialize steps. Removing the need to have to deal with alternative set services routines focuses and clarifies the NUOPC approach.

The remaining aspects of component dependency standardization all deal with build specific issues, i.e. how does the software layer that uses a component compile and link against the component code. For now the NUOPC Layer does not deal with the question on how the component itself is being built. Instead the focus is on the information that a component must provide about itself, and the format of this information, in order to be usable by another piece of software. This clear separation allows components to provide their own independent build system, which often is critical to ensure bit-for-bit reproducibility. At the same time it does not prevent build systems to be connected top-down if that is desirable.

Technically the problem of passing component specific build information up the build hierarchy is solved by using GNU makefile fragments that allow every component to provide information in form of variables to the upper level build system. The NUOPC Layer standardization requires that: **Every component must provide a makefile fragment that defines 6 variables:**

```
ESMF_DEP_FRONT
ESMF_DEP_INCPATH
ESMF_DEP_CMPL_OBJS
ESMF_DEP_LINK_OBJS
```

```
ESMF_DEP_SHRD_PATH
ESMF_DEP_SHRD_LIBS
```

The convention for makefile fragments is to provide them in files with a suffix of `.mk`. The NUOPC Layer currently adds no further restriction to the name of the makefile fragment file of a component. There seems little gain in standardizing the name of the NUOPC compliant makefile fragment of a component since the location must be made available anyway, and adding the specific file name at the end of the supplied path does not appear inappropriate.

The meaning of the 6 makefile variables is defined in a manner that supports many different situations, ranging from simple statically linked components to situations where components are made available in shared objects, not loaded by the application until needed during runtime. The design idea of the NUOPC Layer component makefile fragment is to have each component provide a simple makefile fragment that is self-describing. Usage of advanced options requires a more sophisticated build system on the software layer that *uses* the component, while at the same time the same standard format is able to keep simple situations simple.

An indepth understanding of the capabilities of the NUOPC Layer build dependency standard requires looking at various common cases in detail. The remainder of this chapter is dedicated to this effort. Here a general definition of each variable is provided.

- `ESMF_DEP_FRONT` - The name of the Fortran module to be used in a USE statement, or (if it ends in ".h") the name of the header file to be used in an #include statement, or (if it ends in ".so") the name of the shared object to be loaded at run-time.
- `ESMF_DEP_INCPATH` - The include path to find module or header files during compilation. Must be specified as absolute path.
- `ESMF_DEP_CMPL_OBJS` - Object files that need to be considered as compile dependencies. Must be specified with absolute path.
- `ESMF_DEP_LINK_OBJS` - Object files that need to be considered as link dependencies. Must be specified with absolute path.
- `ESMF_DEP_SHRD_PATH` - The path or list of paths to find shared libraries during link-time (and during run-time unless overridden by LD_LIBRARY_PATH). Must be specified as absolute paths.
- `ESMF_DEP_SHRD_LIBS` - Shared libraries that need to be specified during link-time, and must be available during run-time. Must be specified with absolute path.

The following sections discuss how the standard makefile fragment is utilized in common use cases. It shows how the `.mk` file would need to look like in these cases. Each section further contains hints of how a compliant `.mk` file can be auto-generated by the component build system (provider side), as well as hints on how it can be used by an upper level software layer (consumer side). Makefile segments provided in these hint sections are *not* part of the NUOPC Layer component dependency standard. They are only provided here as a convenience to the user, showing best practices of how the standard `.mk` files can be used in practice. Any specific compiler and linker flags shown in the hint sections are those compliant with the GNU Compiler Collection.

The NUOPC Layer standard only covers the contents of the `.mk` file itself.

4.1 Fortran components that are statically built into the executable

Statically building a component into the executable requires that the associated files (object files, and for Fortran the associated module files) are available when the application is being built. It makes the component code part of the executable. A change in the component code requires re-compilation and re-linking of the executable.

A NUOPC compliant Fortran component that defines its public entry point in a module called "ABC", where all component code is contained in a single object file called "abc.o", makes itself available by providing the following `.mk` file:

```
ESMF_DEP_FRONT      = ABC
ESMF_DEP_INCPATH    = <absolute path to associated ABC module file>
ESMF_DEP_CMPL_OBJS  = <absolute path>/abc.o
ESMF_DEP_LINK_OBJS  = <absolute path>/abc.o
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =
```

If, however, the component implementation is spread across several object files (e.g. abc.o and xyz.o), they must all be listed in the `ESMF_DEP_LINK_OBJS` variable:

```
ESMF_DEP_FRONT      = ABC
ESMF_DEP_INCPATH    = <absolute path to associated ABC module file>
ESMF_DEP_CMPL_OBJS  = <absolute path>/abc.o
ESMF_DEP_LINK_OBJS  = <absolute path>/abc.o <absolute path>/xyz.o
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =
```

In cases that require a large number of object files to be linked into the executable it is often more convenient to provide them in an archive file, e.g. "libABC.a". Archive files are also specified in `ESMF_DEP_LINK_OBJS`:

```
ESMF_DEP_FRONT      = ABC
ESMF_DEP_INCPATH    = <absolute path to associated ABC module file>
ESMF_DEP_CMPL_OBJS  = <absolute path>/abc.o
ESMF_DEP_LINK_OBJS  = <absolute path>/libABC.a
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =
```

Hints for the provider side: A build rule for creating a compliant self-describing `.mk` file can be added to the component's makefile. For the case that component "ABC" is implemented in object files listed in variable "OBJS", a build rule that produces "abc.mk" could look like this:

```
.PRECIOUS: %.o
%.mk : %.o
        @echo "# ESMF self-describing build dependency makefile fragment" > $@
        @echo >> $@
        @echo "ESMF_DEP_FRONT     = ABC"                          >> $@
        @echo "ESMF_DEP_INCPATH   = `pwd`"                        >> $@
        @echo "ESMF_DEP_CMPL_OBJS = `pwd`/"$<                     >> $@
        @echo "ESMF_DEP_LINK_OBJS = "$(addprefix `pwd`/, $(OBJS)) >> $@
        @echo "ESMF_DEP_SHRD_PATH = "                             >> $@
        @echo "ESMF_DEP_SHRD_LIBS = "                             >> $@

abc.mk: $(OBJS)
```

Hints for the consumer side: The format of the NUOPC compliant `.mk` files allows the consumer side to collect the information provided by multiple components into one set of internal variables. Notice that in the makefile code below it is critical to use the := style assignment instead of a simple = in order to have the assignment be based on the *current* value of the right hand variables.

```
include abc.mk
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
```

```
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

include xyz.mk
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_XYZ=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))
```

Besides the accumulation of information into the internal variables, there is a small amount of processing going on. The module name provided by the `ESMF_DEP_FRONT` variable is assigned to a pre-processor macro. The intention of this macro is to be used in a Fortran `USE` statement to access the Fortran module that contains the public access point of the component.

The include paths in `ESMF_DEP_INCPATH` are prepended with the appropriate compiler flag (here "-I"). The `ESMF_DEP_SHRD_PATH` and `ESMF_DEP_SHRD_LIBS` variables are also prepended by the respective compiler and linker flags in case a component brings in a shared library dependencies.

Once the `.mk` files of all component dependencies have been included and processed in this manner, the internal variables can be used in the build system of the application layer, as shown in the following example:

```
.SUFFIXES: .f90 .F90 .c .C

%.o : %.f90
        $(ESMF_F90COMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \
$(ESMF_F90COMPILEOPTS) $(ESMF_F90COMPILEPATHS) $(ESMF_F90COMPILEFREENOCPP) $<

%.o : %.F90
        $(ESMF_F90COMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \
$(ESMF_F90COMPILEOPTS) $(ESMF_F90COMPILEPATHS) $(ESMF_F90COMPILEFREECPP) \
$(ESMF_F90COMPILECPPFLAGS) $<

%.o : %.c
        $(ESMF_CXXCOMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \
$(ESMF_CXXCOMPILEOPTS) $(ESMF_CXXCOMPILEPATHSLOCAL) $(ESMF_CXXCOMPILEPATHS) \
$(ESMF_CXXCOMPILECPPFLAGS) $<

%.o : %.C
        $(ESMF_CXXCOMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \
$(ESMF_CXXCOMPILEOPTS) $(ESMF_CXXCOMPILEPATHSLOCAL) $(ESMF_CXXCOMPILEPATHS) \
$(ESMF_CXXCOMPILECPPFLAGS) $<

app: app.o appSub.o $(DEP_LINK_OBJS)
        $(ESMF_F90LINKER) $(ESMF_F90LINKOPTS) $(ESMF_F90LINKPATHS) \
$(ESMF_F90LINKRPATHS) -o $@ $^ $(DEP_SHRD_PATH) $(DEP_SHRD_LIBS) \
$(ESMF_F90ESMFLINKLIBS)

app.o: appSub.o
appSub.o: $(DEP_CMPL_OBJS)
```

4.2 Fortran components that are provided as shared libraries

Providing a component in form of a shared library requires that the associated files (object files, and for Fortran the associated module files) are available when the application is being built. However, different from the statically linked case, the component code does *not* become part of the executable, instead it will be loaded separately each time the executable is loaded during start-up. This requires that the executable finds the component shared libraries, on which it depends, during start-up. A change in the component code typically does not require re-compilation and re-linking of the executable, instead a new version of the component shared library will be loaded automatically when it is available at execution start-up.

A NUOPC compliant Fortran component that defines its public entry point in a module called "ABC", where all component code is contained in a single shared library called "libABC.so", makes itself available by providing the following `.mk` file:

```
ESMF_DEP_FRONT      = ABC
ESMF_DEP_INCPATH    = <absolute path to associated ABC module file>
ESMF_DEP_CMPL_OBJS  =
ESMF_DEP_LINK_OBJS  =
ESMF_DEP_SHRD_PATH  = <absolute path to libABC.so>
ESMF_DEP_SHRD_LIBS  = libABC.so
```

Hints for the provider side: The following build rule will create a compliant self-describing `.mk` file ("abc.mk") for a component that is made available as a shared library. The case assumes that component "ABC" is implemented in object files listed in variable "OBJS".

```
.PRECIOUS: %.so
%.mk : %.so
        @echo "# ESMF self-describing build dependency makefile fragment" > $@
        @echo >> $@
        @echo "ESMF_DEP_FRONT     = ABC"                          >> $@
        @echo "ESMF_DEP_INCPATH   = `pwd`"                        >> $@
        @echo "ESMF_DEP_CMPL_OBJS = "                             >> $@
        @echo "ESMF_DEP_LINK_OBJS = "                             >> $@
        @echo "ESMF_DEP_SHRD_PATH = `pwd`"                        >> $@
        @echo "ESMF_DEP_SHRD_LIBS = "$*                           >> $@

abc.mk:

abc.so: $(OBJS)
        $(ESMF_CXXLINKER) -shared -o $@ $<
        mv $@ lib$@
        rm -f $<
```

Hints for the consumer side: The format of the NUOPC compliant `.mk` files allows the consumer side to collect the information provided by multiple components into one set of internal variables. This is independent on whether some or all of the components are provided as shared libraries.

The path specified in `ESMF_DEP_SHRD_PATH` is required when building the executable in order for the linker to find the shared library. Depending on the situation, it may be desirable to also encode this search path into the executable through the RPATH mechanism as shown below. However, in some cases, e.g. when the actual shared library to be used during execution is *not* available from the same location as during build-time, it may not be useful to encode the RPATH. In either case, having set the `LD_LIBRARY_PATH` environment variable to the desired location of the shared library at run-time will ensure that the correct library file is found.

Notice that in the makefile code below it is critical to use the := style assignment instead of a simple = in order to have the assignment be based on the *current* value of the right hand variables.

```
include abc.mk
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
  $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))
```

(Here `COMMA` is a variable that contains a single comma which would cause syntax issues if it was written into the "addprefix" command directly.)

The internal variables set by the above makefile code can then be used by exactly the same makefile rules shown for the statically linked case. In fact, component "ABC" that comes in through "abc.mk" could either be a statically linked component or a shared library component. The makefile code shown here for the consumer side handles both cases alike.

4.3 Components that are loaded during run-time as shared objects

Making components available in the form of shared objects allows the executable to be built in the complete absence of any information that depends on the component code. The only information required when building the executable is the name of the shared object file that will supply the component code during run-time. The shared object file of the component can be replaced at will, and it is not until run-time, when the executable actually tries to access the component, that the shared object must be available to be loaded.

A NUOPC compliant component where all component code, including its public access point, is contained in a single shared object called "abc.so", makes itself available by providing the following `.mk` file:

```
  ESMF_DEP_FRONT      = abc.so
  ESMF_DEP_INCPATH    =
  ESMF_DEP_CMPL_OBJS  =
  ESMF_DEP_LINK_OBJS  =
  ESMF_DEP_SHRD_PATH  =
  ESMF_DEP_SHRD_LIBS  =
```

The other parts of the `.mk` file may be utilized in special cases, but typically the shared object should be self-contained.

It is interesting to note that at this level of abstraction, there is no more difference between a component written in Fortran, and a component written in in C/C++. In both cases the public entry point available in the shared object must be `SetServices` as required by the NUOPC Layer component dependency standard. (NUOPC does allow for customary name mangling by the Fortran compiler.)

Hints for the provider side: The following build rule will create a compliant self-describing `.mk` file ("abc.mk") for a component that is made available as a shared object. The case assumes that component "ABC" is implemented in object files listed in variable "OBJS".

```
.PRECIOUS: %.so
%.mk : %.so
        @echo "# ESMF self-describing build dependency makefile fragment" > $@
        @echo >> $@
        @echo "ESMF_DEP_FRONT     = "$<                              >> $@
        @echo "ESMF_DEP_INCPATH   = "                                >> $@
        @echo "ESMF_DEP_CMPL_OBJS = "                                >> $@
        @echo "ESMF_DEP_LINK_OBJS = "                                >> $@
```

```
        @echo "ESMF_DEP_SHRD_PATH = "                                  >> $@
        @echo "ESMF_DEP_SHRD_LIBS = "                                  >> $@

abc.mk:

abc.so: $(OBJS)
        $(ESMF_CXXLINKER) -shared -o $@ $<
        rm -f $<
```

Hints for the consumer side: The format of the NUOPC compliant `.mk` files still allows the consumer side to collect the information provided by multiple components into one set of internal variables. This still holds when some or all of the components are provided as shared objects. In fact it is very simple to make all of the component sections in the consumer makefile handle both cases.

Notice that in the makefile code below it is critical to use the := style assignment instead of a simple = in order to have the assignment be based on the *current* value of the right hand variables.

```
include abc.mk
ifneq (,$(findstring .so,$(ESMF_DEP_FRONT)))
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_SO_ABC=\"$(ESMF_DEP_FRONT)\"
else
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
endif
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
  $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))
```

The above makefile segment supports component "ABC" that is described in "abc.mk" to be made available as a Fortran static component, a Fortran shared library, or a shared object. The conditional around assigning variable `DEP_FRONTS` either leads to having set the macro `FRONT_ABC` as before, or setting a different macro `FRONT_SO_ABC`. The former indicates that a Fortran module is available for the component and requires a `USE` statement in the code. The latter macro indicates that the component is made available through a shared object, and the macro can be used to specify the name of the shared object in the associated call.

Again the internal variables set by the above makefile code can be used by the same makefile rules shown for the statically linked case.

4.4 Components that depend on components

The NUOPC Layer supports component hierarchies where a component can be a child of another component. This hierarchy of components translates into component build dependencies that must be dealt with in the NUOPC Layer standardization of component dependencies.

A component that sits in an intermediate level of the component hierarchy depends on the components "below" while at the same time it introduces a dependency by itself for the parent further "up" in the hierarchy. Within the NUOPC Layer component dependency standard this means that the intermediate component functions as a consumer of its child components' `.mk` files, and as a provider of its own `.mk` file that is then consumed by its parent. In practice this double role translates into passing link dependencies and shared library dependencies through to the parent, while the front and compile dependency is simply defined my the intermediate component itself.

Consider a NUOPC compliant component that defines its public entry point in a module called "ABC", and where all component code is contained in a single object file called "abc.o". Further assume that component "ABC" depends on two components "XXX" and "YYY", where "XXX" provides the `.mk` file:

```
ESMF_DEP_FRONT      = XXX
ESMF_DEP_INCPATH    = <absolute path to the associated XXX module file>
ESMF_DEP_CMPL_OBJS  = <absolute path>/xxx.o
ESMF_DEP_LINK_OBJS  = <absolute path>/xxx.o
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =
```

and "YYY" provides the following:

```
ESMF_DEP_FRONT      = YYY
ESMF_DEP_INCPATH    = <absolute path to the associated XXX module file>
ESMF_DEP_CMPL_OBJS  =
ESMF_DEP_LINK_OBJS  =
ESMF_DEP_SHRD_PATH  = <absolute path to libYYY.so>
ESMF_DEP_SHRD_LIBS  = libYYY.so
```

Then the `.mk` file provided by "ABC" needs to contain the following information:

```
ESMF_DEP_FRONT      = ABC
ESMF_DEP_INCPATH    = <absolute path to the associated ABC module file>
ESMF_DEP_CMPL_OBJS  = <absolute path>/abc.o
ESMF_DEP_LINK_OBJS  = <absolute path>/abc.o <absolute path>/xxx.o
ESMF_DEP_SHRD_PATH  = <absolute path to libYYY.so>
ESMF_DEP_SHRD_LIBS  = libYYY.so
```

Hints for an intermediate component that is consumer and provider: For the consumer side it is convenient to collect the information provided by multiple component dependencies into one set of internal variables. However, the details on how some of the imported information is processed into the internal variables depends on whether the intermediate component is going to make itself available for static or dynamic access.

In the static case all link and shared library dependencies must be passed to the next higher level, and these dependencies should simply be collected and passed on to the next level:

```
include xxx.mk
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_XXX=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(ESMF_DEP_SHRD_PATH)
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(ESMF_DEP_SHRD_LIBS)

include yyy.mk
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_YYY=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(ESMF_DEP_SHRD_PATH)
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(ESMF_DEP_SHRD_LIBS)

.PRECIOUS: %.o
```

```
%.mk : %.o
 @echo "# ESMF self-describing build dependency makefile fragment" > $@
 @echo >> $@
 @echo "ESMF_DEP_FRONT     = ABC"                          >> $@
 @echo "ESMF_DEP_INCPATH   = `pwd`"                        >> $@
 @echo "ESMF_DEP_CMPL_OBJS = `pwd`/"$<                     >> $@
 @echo "ESMF_DEP_LINK_OBJS = `pwd`/"$< $(DEP_LINK_OBJS)    >> $@
 @echo "ESMF_DEP_SHRD_PATH = " $(DEP_SHRD_PATH)            >> $@
 @echo "ESMF_DEP_SHRD_LIBS = " $(DEP_SHRD_LIBS)            >> $@
```

In the case where the intermediate component is linked into a dynamic library, or a dynamic object, all of its object and shared library dependencies can be linked in. In this case it is more useful to do some processing on the shared library dependencies, and not to include them in the produced `.mk` file.

```
include xxx.mk
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_XXX=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
  $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

include yyy.mk
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_YYY=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
  $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))

.PRECIOUS: %.o
%.mk : %.o
 @echo "# ESMF self-describing build dependency makefile fragment" > $@
 @echo >> $@
 @echo "ESMF_DEP_FRONT     = ABC"                          >> $@
 @echo "ESMF_DEP_INCPATH   = `pwd`"                        >> $@
 @echo "ESMF_DEP_CMPL_OBJS = `pwd`/"$<                     >> $@
 @echo "ESMF_DEP_LINK_OBJS = `pwd`/"$<                     >> $@
 @echo "ESMF_DEP_SHRD_PATH = "                             >> $@
 @echo "ESMF_DEP_SHRD_LIBS = "                             >> $@
```

4.5 Components written in C/C++

ESMF provides a basic C API that supports writing components in C or C++. There is currently no C version of the NUOPC Layer API available, making it harder, but not impossible to write NUOPC Layer compliant ESMF components in C/C++. For the sake of completeness, the NUOPC component dependency standardization does cover the case of components being written in C/C++.

The issue of whether a component is written in Fortran or C/C++ only matters when the dependent software layer has a compile dependency on the component. In other words, components that are accessed through a shared object have no compile dependency, and the language is of no effect (see 4.3). However, components that are statically linked or made available through shared libraries do introduce compile dependencies. These compile dependencies become language

dependent: a Fortran component must be accessed via the `USE` statement, while a component with a C interface must be accessed via `#include`.

The decision between the three cases: compile dependency on a Fortran component, compile dependency on a C/C++ component, or no compile dependency can be made on the `ESMF_DEP_FRONT` variable. By default it is assumed to contain the name of the Fortran module that provides the public entry point into a component written in Fortran. However, if the contents of the `ESMF_DEP_FRONT` variable ends in `.h`, it is interpreted as the header file of a component with a C interface. Finally, if it ends in `.so`, there is no compile dependency, and the component is accessible through a shared object.

A NUOPC compliant component written in C/C++ that defines its public access point in "abc.h", where all component code is contained in a single object file called "abc.o", makes itself available by providing the following `.mk` file:

```
ESMF_DEP_FRONT      = abc.h
ESMF_DEP_INCPATH    = <absolute path to abc.h>
ESMF_DEP_CMPL_OBJS  = <absolute path>/abc.o
ESMF_DEP_LINK_OBJS  = <absolute path>/abc.o
ESMF_DEP_SHRD_PATH  =
ESMF_DEP_SHRD_LIBS  =
```

Hints for the implementor:

There are a few subtle complications to cover for the case where a component with C interface comes in as a compile dependency. First there is Fortran name mangling of symbols which includes underscores, but also changes to lower or upper case letters. The ESMF C interface provides a macro (`FTN_X`) that deals with the underscore issue on the C component side, but it cannot address the lower/upper case issue. The ESMF convention for using C in Fortran assumes all external symbols lower case. The NUOPC Layer follows this convention in accessing components with C interface from Fortran.

Secondly, there is no namespace protection of the public entry points. For this reason, the public entry point cannot just be `setservices` for all components written in C. Instead, for components with C interface, the public entry point must be `setservices_name`, where "name" is the same as the root name of the header file specified in `ESMF_DEP_FRONT`. (The absence of namespace protection is still an issue where multiple C components with the same name are specified. This case requires that components are renamed to something more unique.)

Finally there is the issue of providing an explicit Fortran interface for the public entry point. One way of handling this is to provide the explicit Fortran interface as part of the components header file. This is essentially a few lines of Fortran code that can be used by the upper software layer to implement the explicit interface. As such it must be protected from being processed by the C/C++ compiler:

```
#if (defined  __STDC__ || defined __cplusplus)

// ---------- C/C++ block ------------

#include "ESMC.h"
extern "C" {
  void FTN_X(setservices_abc)(ESMC_GridComp gcomp, int *rc);
}

#else

!! ---------- Fortran block ----------

interface
  subroutine setservices_abc(gcomp, rc)
    use ESMF
```

```
    type(ESMF_GridComp)  :: gcomp
    integer, intent(out) :: rc
  end subroutine
end interface

#endif
```

An upper level software layer that intends to use a component that comes with such a header file can then use it directly on the Fortran side to make the component available with an explicit interface. For example, assuming the macro `FRONT_H_ATMF` holds the name of the associated header file:

```
#ifdef FRONT_H_ATMF
module ABC
#include FRONT_H_ATMF
end module
#endif
```

This puts the explicit interface of the `setservices_abc` entry point into a module named "ABC". Except for this small block of code, the C/C++ component becomes indistinguishable from a component implemented in Fortran.

Hints for the provider side: Adding a build rule for creating a compliant self-describing `.mk` file into the component's makefile is straightforward. For the case that the component in "abc.h" is implemented in object files listed in variable "OBJS", a build rule that produces "abc.mk" could look like this:

```
.PRECIOUS: %.o
%.mk : %.o
 @echo "# ESMF self-describing build dependency makefile fragment" > $@
 @echo >> $@
 @echo "ESMF_DEP_FRONT     = abc.h"     >> $@
 @echo "ESMF_DEP_INCPATH   = `pwd`"      >> $@
 @echo "ESMF_DEP_CMPL_OBJS = `pwd`/"$<   >> $@
 @echo "ESMF_DEP_LINK_OBJS = `pwd`/"$<   >> $@
 @echo "ESMF_DEP_SHRD_PATH = "           >> $@
 @echo "ESMF_DEP_SHRD_LIBS = "           >> $@

abc.mk:

abc.o: abc.h
```

Hints for the consumer side: The format of the NUOPC compliant `.mk` files still allows the consumer side to collect the information provided by multiple components into one set of internal variables. This still holds even when any of the provided components could come in as a Fortran component for static linking, as a C/C++ component for static linking, or as a shared object. All of the component sections in the consumer makefile can be made capable of handling all three cases. However, if it is clear that a certain component is for sure supplied as one of these flavors, it may be clearer to hard-code support for only one mechanism for this component.

Notice that in the makefile code below it is critical to use the `:=` style assignment instead of a simple = in order to have the assignment be based on the *current* value of the right hand variables.

This example shows how the section for a specific component can be made compatible with all component dependency modes:

```
include abc.mk
```

```
ifneq (,$(findstring .h,$(ESMF_DEP_FRONT)))
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_H_ABC=\"$(ESMF_DEP_FRONT)\"
else ifneq (,$(findstring .so,$(ESMF_DEP_FRONT)))
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_SO_ABC=\"$(ESMF_DEP_FRONT)\"
else
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
endif
DEP_FRONTS    := $(DEP_FRONTS) -DFRONT_ABC=$(ESMF_DEP_FRONT)
DEP_INCS      := $(DEP_INCS) $(addprefix -I, $(ESMF_DEP_INCPATH))
DEP_CMPL_OBJS := $(DEP_CMPL_OBJS) $(ESMF_DEP_CMPL_OBJS)
DEP_LINK_OBJS := $(DEP_LINK_OBJS) $(ESMF_DEP_LINK_OBJS)
DEP_SHRD_PATH := $(DEP_SHRD_PATH) $(addprefix -L, $(ESMF_DEP_SHRD_PATH)) \
  $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESMF_DEP_SHRD_PATH))
DEP_SHRD_LIBS := $(DEP_SHRD_LIBS) $(addprefix -l, $(ESMF_DEP_SHRD_LIBS))
```

The above makefile segment will end up setting macro `FRONT_H_ABC` to the header file name, if the component described in "abc.mk" is a C/C++ component. It will instead set macro `FRONT_SO_ABC` to the shared object if this is how the component is made available, or set macro `FRONT_ABC` to the Fortran module name if that is the mechanism for gaining access to the component code. The calling code can use these macros to activate the corresponding code, as well as has access to the required name string in each case

The internal variables set by the above makefile code can be used by the same makefile rules shown for the statically linked case. This usage implements the correct dependency rules, and passes the macros through the compiler flags.

5 NUOPC Layer Compliance

The NUOPC Layer introduces a modeling system architecture based on Models, Mediators, Connectors, and Drivers. The Layer defines the rules of engagement between these components. Many of these rules are formulated on the basis of metadata. This metadata can be expected for compliance.

One of the challenges when inspecting a component for NUOPC Layer compliance is that many of the rules of engagement are run-time rules. This means that they address the dynamical behavior of a component during run-time. For this reason, comprehensive compliance testing cannot be done statically but requires the execution of code.

Currently there are two sets of tools available to address the issue of NUOPC Layer compliance testing. The *Compliance Checker* is a runtime analysis tool that can be enabled by setting an ESMF environment variable at runtime. When active, the Compliance Checker intercepts all interactions between components that go throught the ESMF component interface, and analyzes them with respect to the NUOPC Layer rules of engagement. Warnings are printed to the log files when issues or non-compliances are detected.

The *Component Explorer* is another compliance testing tool. It focuses on interacting with a single component, and analyzing it during the early initialization phases. The Component Explorer and Compliance Checker are compatible with each other and it is often useful to use them both at the same time.

5.1 The Compliance Checker

The NUOPC Compliance Checker is a run-time analysis tool that can be turned on for any ESMF application. The Compliance Checker is turned off by default, as to not negatively affect performance critical runs. The Compliance Checker is enabled by setting the following ESMF runtime environment variable:

```
ESMF_RUNTIME_COMPLIANCECHECK=ON
```

As a run-time variable, setting it does not require recompilation of the ESMF library or the user application. The same

executable and library will start to generate Compliance Checker output when the above variable is found set during execution.

The function of the Compliance Checker is to intercept all interactions between the components of an ESMF application, and to analyze them according to the NUOPC Layer rules of engagement. The following aspects are currently reported on:

- Presence of the standard ESMF Initialize, Run, and Finalize methods and the number of phases in each.
- Timekeeping and whether it conforms with the NUOPC Layer rules.
- Fields or FieldBundles (not Arrays/ArrayBundles) being passed between Components.
- Details about the Fields being passed through import and export States.
- Component and Field metadata.

Besides the above aspects, the output of the Compliance Checker also provides a means to easily get an idea of the exact dynamical control flow between the components of an application.

The Compliance Checker uses the ESMF Log facility to produce the compliance report during the execution of an ESMF application. The output is located in the default ESMF Log files. There are advantages of using the existing Log facility to generate the compliance report. First, the ESMF Log facility offers time stamping of messages, and deals with all of the file access and multi-PET issues. Second, going through the ESMF Log guarantees that all the output appears in the correct chronological order. This applies to all of the output, including entries from other ESMF system levels or from the user level.

A sample output of the Compliance Checker output in action:

```
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:>START register compliance check.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: phase Zero for Initialize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:             5 phase(s) of Initialize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:             1 phase(s) of Run registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:             1 phase(s) of Finalize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:>STOP register compliance check.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM2MED:>START register compliance check.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM2MED: phase Zero for Initialize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM2MED:             3 phase(s) of Initialize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM2MED:             1 phase(s) of Run registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM2MED:             1 phase(s) of Finalize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM2MED:>STOP register compliance check.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:MED2ATM:>START register compliance check.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:MED2ATM: phase Zero for Initialize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:MED2ATM:             3 phase(s) of Initialize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:MED2ATM:             1 phase(s) of Run registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:MED2ATM:             1 phase(s) of Finalize registered.
20131108 172844.458 INFO     PET0 COMPLIANCECHECKER:|->|->|->:MED2ATM:>STOP register compliance check.
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: >START InitializePrologue for phase=          0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: importState name: modelComp 1 Import State
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: importState stateintent: ESMF_STATEINTENT_IMPORT
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: importState itemCount:           0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: exportState name: modelComp 1 Export State
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: exportState stateintent: ESMF_STATEINTENT_EXPORT
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: exportState itemCount:           0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:ESMF Stats: the virtual memory used by this PET (in KB): 974868
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:ESMF Stats: the physical memory used by this PET (in KB): 49440
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:ESMF Stats: ESMF Fortran objects referenced by the ESMF garbage collection: 0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM:ESMF Stats: ESMF objects (F & C++) referenced by the ESMF garbage collection: 0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|->|->|->:ATM: >STOP InitializePrologue for phase=          0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: >START InitializeEpilogue for phase=          0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM:ESMF Stats: the virtual memory used by this PET (in KB): 974868
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM:ESMF Stats: the physical memory used by this PET (in KB): 49448
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM:ESMF Stats: ESMF Fortran objects referenced by the ESMF garbage collection: 0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM:ESMF Stats: ESMF objects (F & C++) referenced by the ESMF garbage collection: 0
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: GridComp level attribute check: convention: 'NUOPC', purpose: 'General'.
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <ShortName> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <LongName> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <Description> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <ModelType> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <ReleaseDate> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <PreviousVersion> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <ResponsiblePartyRole> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <Name> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <EmailAddress> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <PhysicalAddress> present but NOT set!
20131108 172844.459 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> Component level attribute: <URL> present but NOT set!
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <Verbosity> present and set: high
20131108 172844.459 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <InitializePhaseMap>[1] present and set: IPDv02p1=1
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <InitializePhaseMap>[2] present and set: IPDv02p3=2
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <InitializePhaseMap>[3] present and set: IPDv02p4=3
```

```
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <InitializePhaseMap>[4] present and set: IPDv02p5=5
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <NestingGeneration> present and set:           0
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: Component level attribute: <Nestling> present and set:           0
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: importState name: modelComp 1 Import State
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: importState stateintent: ESMF_STATEINTENT_IMPORT
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: importState itemCount:          0
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: exportState name: modelComp 1 Export State
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: exportState stateintent: ESMF_STATEINTENT_EXPORT
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: exportState itemCount:          0
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: The incoming Clock was not modified.
20131108 172844.460 WARNING  PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: ==> The internal Clock is not present!
20131108 172844.460 INFO     PET0 COMPLIANCECHECKER:|<-|<-|<-:ATM: >STOP InitializeEpilogue for phase=          0
```

All of the output generated by the Compliance Checker contains the string `COMPLIANCECHECK`, which can be used to grep on. The checker currently generates two types of messages, `INFO` for general analysis output, and `WARNING` for when issues with respect to the NUOPC Layer rules are detected.

In practice, when dealing with applications that have been componentized down to a very low level of the model, the output generated by the Compliance Checker can become overwhelming. For this reason a `depth` parameter is available that can be specified for the Compliance Checker environment variable:

```
ESMF_RUNTIME_COMPLIANCECHECK=ON:depth=4
```

This will limit the number of component levels that the Compliance Checker parses (here 4 levels), starting from the top level application.

5.2 The Component Explorer

The NUOPC Component Explorer is a run-time tool that can be used to gain insight into a NUOPC Layer compliant component, or to test a component's compliance. The Component Explorer is currently available as a separate download from the prototype repository:

```
https://github.com/esmf-org/nuopc-app-prototypes/tree/develop/AtmOcnProto
```

There are two parts to the Component Explorer. First the script `nuopcExplorerScript` is used to compile and link the explorer application specifically against a specified component. This part of the explorer leverages and tests the standardized component dependencies discussed in section 4. This step is initiated by calling the explorer script with the component's mk-file as an argument:

```
./nuopcExplorerScript <component-mk-file>
```

Any issues found during this step are reported. The successful completion of this step will produce an executable called `nuopcExplorerApp`. Success is indicated by

```
SUCCESS: nuopcExplorerApp successfully built
...exiting nuopcExplorerScript.
```

and failure by

```
FAILURE: nuopcExplorerApp failed to build
...exiting nuopcExplorerScript.
```

The second part of the Component Explorer is the explorer application itself. It can either be built using the explorer script as outlined above (recommended when a makefile fragment for the component is available) or by using the makefile directly:

```
make nuopcExplorerApp
```

In the second case the resulting `nuopcExplorerApp` is not tied to a specific component, instead the executable expects a component in form of a shared object to be specified as a command line argument when executing `nuopcExplorerApp`. In either case the explorer application needs to be started according to the execution requirements of the component it attempts to explore. This may mean that input files must be present, and that the executable be launched on a sufficient number of processes. In terms of the common `mpirun` tool, launching of `nuopcExplorerApp` may look like this

```
mpirun -np X ./nuopcExplorerApp
```

for an executable that was built against a specific component. Or like this

```
mpirun -np X ./nuopcExplorerApp <component-shared-object-file>
```

for an executable that expects a the component in form of a shared object.

The `nuopcExplorerApp` expects to find a configuration file by the name of `explorer.config` in the run directory. The configuration file contains several basic model parameter used to explore the component. An example configuration file is shown here:

```
### NUOPC Component Explorer configuration file ###

start_year:               2009
start_month:              12
start_day:                01
start_hour:               00
start_minute:             0
start_second:             0

stop_year:                2009
stop_month:               12
stop_day:                 03
stop_hour:                00
stop_minute:              0
stop_second:              0

step_seconds:             21600

filter_initialize_phases: no

enable_run:               yes
enable_finalize:          yes
```

The `nuopcExplorerApp` starts to interact with the specified component, using the information read in from the configuration file. During the interaction the finding are reported to stdout, with output that will look similar to this:

```
 NUOPC Component Explorer App
 ----------------------------
 Exploring a component with a Fortran module front...
 Model component # 1 InitializePhaseMap:
   IPDv00p1=1
   IPDv00p2=2
```

```
  IPDv00p3=3
  IPDv00p4=4
Model component # 1 // name = ocnA
  ocnA: <LongName>    : Attribute is present but NOT set!
  ocnA: <ShortName>   : Attribute is present but NOT set!
  ocnA: <Description> : Attribute is present but NOT set!
     --------
  ocnA: importState // itemCount = 2
  ocnA: importState // item # 001 // [FIELD] name = pmsl
              <StandardName> = air_pressure_at_sea_level
                     <Units> = Pa
                  <LongName> = Air Pressure at Sea Level
                 <ShortName> = pmsl
  ocnA: importState // item # 002 // [FIELD] name = rsns
              <StandardName> = surface_net_downward_shortwave_flux
                     <Units> = W m-2
                  <LongName> = Surface Net Downward Shortwave Flux
                 <ShortName> = rsns
     --------
  ocnA: exportState // itemCount = 1
  ocnA: exportState // item # 001 // [FIELD] name = sst
              <StandardName> = sea_surface_temperature
                     <Units> = K
                  <LongName> = Sea Surface Temperature
                 <ShortName> = sst
```

Turning on the Compliance Checker (see section 5.1) will result in additional information in the log files.

6 Appendix A: Run Sequence Implementation

The NUOPC Driver utilizes an internal class to parametrize the run sequence. The `NUOPC_RunSequence` provides a unified data structure that allows simple as well as complex time loops to be encoded and executed. There are entry points that allow different run phases to be mapped against distinctly different time loops. Figure 2 depicts the data structures surrounding the `NUOPC_RunSequence`, starting with the InternalState of the `NUOPC_Driver` generic component.

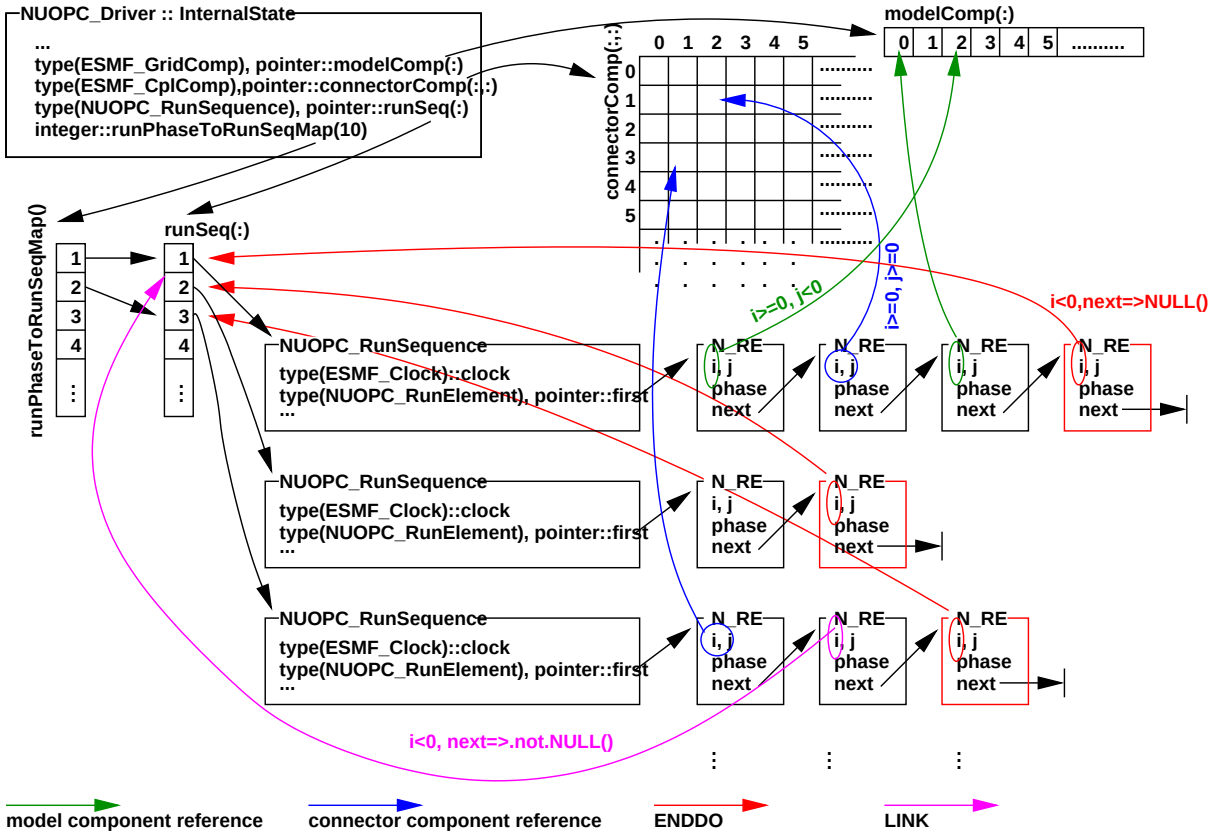

Figure 2: NUOPC_RunSequence class as it relates to the surrounding data structures.

7 Appendix B: Initialize Phase Definition Versions

IMPORTANT: Use of explicit Initialize Phase Definition versions and phase labels is deprecated - this section is provided only for reference for NUOPC caps that still use the IPD syntax. See the section on Semantic Specialization Labels for the preferred method of specializing NUOPC caps.

The interaction between NUOPC compliant components during the initialization process is regulated by the **Initialize Phase Definition** or **IPD**. The IPDs are versioned, with a higher version number indicating backward compatibility with all previous versions.

There are two perspectives of looking at the IPD. From the driver perspective the IPD regulates the sequence in which it must call the different phases of the Initialize() routines of its child components. To this end the generic `NUOPC_Driver` component implements support for IPDs up to a version specified in the API documentation.

The other angle of looking at the IPD is from the driver's child components. From this perspective the IPD assigns specific meaning to each initialize phase. The child components of a driver can be divided into two groups with respect to the meaning the IPD assigns to each initialize phase. In one group are the model, mediator, and driver components, and in the other group are the connector components. Child components publish their available initialize phases through the `InitializePhaseMap` attribute.

The driver also calls into its own internal initialize methods. This allows the driver to participate in the initialization of its children in a structured fashion. The internal initialization phases of a driver are published via the `InternalInitializePhaseMap` attribute.

The following tables document the meaning of each initialization phase of the available IPD versions for the child components and for the driver component itself. The phases are listed in the sequence in which the driver calls them.

IPDv00 label	Component	Meaning
`IPDv00p1`	driver-internal	*unspecified by NUOPC*
`IPDv00p1`	models, mediators, drivers	Advertise their import and export Fields.
`IPDv00p1`	connectors	Construct their `CplList` Attribute.
`IPDv00p2`	driver-internal	*unspecified by NUOPC*
`IPDv00p2`	models, mediators, drivers	Realize their import and export Fields.
`IPDv00p2a`	connectors	Set the `Connected` Attribute on each import and export Field according to the `CplList` Attribute. Reconcile the import and export States.
`IPDv00p2b`	connectors	Precompute the RouteHandle.
`IPDv00p3`	driver-internal	*unspecified by NUOPC*
`IPDv00p3`	models, mediators, drivers	Check for compatibility of their Fields' `Connected` status.
`IPDv00p4`	driver-internal	*unspecified by NUOPC*
`IPDv00p4`	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields.

IPDv01 label	Component	Meaning
`IPDv01p1`	driver-internal	*unspecified by NUOPC*
`IPDv01p1`	models, mediators, drivers	Advertise their import and export Fields.
`IPDv01p1`	connectors	Construct their `CplList` Attribute.
`IPDv01p2`	driver-internal	Modify the `CplList` Attributes on the Connectors.
`IPDv01p2`	models, mediators, drivers	*unspecified/unused by NUOPC*
`IPDv01p2`	connectors	Set the `Connected` Attribute on each import and export Field according to the `CplList` Attribute.

IPDv01p3	driver-internal	*unspecified by NUOPC*
IPDv01p3	models, mediators, drivers	Realize their "connected" import and export Fields.
IPDv01p3a	connectors	Reconcile the import and export States.
IPDv01p3b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv01p4	driver-internal	*unspecified by NUOPC*
IPDv01p4	models, mediators, drivers	Check for compatibility of their Fields' Connected status.
IPDv01p5	driver-internal	*unspecified by NUOPC*
IPDv01p5	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields.

IPDv02 label	**Component**	**Meaning**
IPDv02p1	driver-internal	*unspecified by NUOPC*
IPDv02p1	models, mediators, drivers	Advertise their import and export Fields.
IPDv02p1	connectors	Construct their CplList Attribute.
IPDv02p2	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv02p2	models, mediators, drivers	*unspecified/unused by NUOPC*
IPDv02p2	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute.
IPDv02p3	driver-internal	*unspecified by NUOPC*
IPDv02p3	models, mediators, drivers	Realize their "connected" import and export Fields.
IPDv02p3a	connectors	Reconcile the import and export States.
IPDv02p3b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv02p4	driver-internal	*unspecified by NUOPC*
IPDv02p4	models, mediators, drivers	Check for compatibility of their Fields' Connected status.
IPDv02p5	driver-internal	*unspecified by NUOPC*
IPDv02p5	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields.
A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:		
Run()	connectors	Loop over all Connectors that connect *to* the Component that is currently indexed by the outer loop.
IPDv02p5	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields and set the Updated and InitializeDataComplete Attributes accordingly.
Repeat these two steps until all data dependencies have been statisfied, or a dead-lock situation is detected.		

IPDv03 label	**Component**	**Meaning**
IPDv03p1	driver-internal	*unspecified by NUOPC*
IPDv03p1	models, mediators, drivers	Advertise their import and export Fields and set the TransferOfferGeomObject Attribute.
IPDv03p1	connectors	Construct their CplList Attribute.
IPDv03p2	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv03p2	models, mediators, drivers	*unspecified/unused by NUOPC*

IPDv03p2	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute. Set the TransferActionGeomObject Attribute.
IPDv03p3	driver-internal	*unspecified by NUOPC*
IPDv03p3	models, mediators, drivers	Realize their "connected" import and export Fields that have TransferActionGeomObject equal to "provide".
IPDv03p3	connectors	Transfer the Grid/Mesh/LocStream objects (only DistGrid) for Field pairs that have a provider and an acceptor side.
IPDv03p4	driver-internal	*unspecified by NUOPC*
IPDv03p4	models, mediators, drivers	Optionally modify the decomposition and distribution information of the accepted Grid/Mesh/LocStream by replacing the DistGrid.
IPDv03p4	connectors	Transfer the full Grid/Mesh/LocStream objects (with coordinates) for Field pairs that have a provider and an acceptor side.
IPDv03p5	driver-internal	*unspecified by NUOPC*
IPDv03p5	models, mediators, drivers	Realize all Fields that have TransferActionGeomObject equal to "accept" on the transferred Grid/Mesh/LocStream objects.
IPDv03p5a	connectors	Reconcile the import and export States.
IPDv03p5b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv03p6	driver-internal	*unspecified by NUOPC*
IPDv03p6	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv03p7	driver-internal	*unspecified by NUOPC*
IPDv03p7	models, mediators, drivers	Handle Field data initialization. Timestamp the export Fields.
A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:		
Run()	connectors	Loop over all Connectors that connect *to* the Component that is currently indexed by the outer loop.
IPDv03p7	models, mediators, drivers	Handle Field data initialization. Time stamp the export Fields and set the Updated and InitializeDataComplete Attributes accordingly.
Repeat these two steps until all data dependencies have been statisfied, or a dead-lock situation is detected.		

IPDv04 label	**Component**	**Meaning**
IPDv04p1	driver-internal	*unspecified by NUOPC*
IPDv04p1	models, mediators, drivers	Advertise their import and export Fields and set the TransferOfferGeomObject Attribute.
IPDv04p1a	connectors	Consider all connection possibilities for their CplList Attribute.
IPDv04p1b	connectors	Unambiguous construction of their CplList Attribute.
IPDv04p2	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv04p2	models, mediators, drivers	*unspecified/unused by NUOPC*
IPDv04p2	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute. Set the TransferActionGeomObject Attribute.
IPDv04p3	driver-internal	*unspecified by NUOPC*
IPDv04p3	models, mediators, drivers	Realize their "connected" import and export Fields that have TransferActionGeomObject equal to "provide".

IPDv04p3	connectors	Transfer the Grid/Mesh/LocStream objects (only DistGrid) for Field pairs that have a provider and an acceptor side.
IPDv04p4	driver-internal	*unspecified by NUOPC*
IPDv04p4	models, mediators, drivers	Optionally modify the decomposition and distribution information of the accepted Grid/Mesh/LocStream by replacing the DistGrid.
IPDv04p4	connectors	Transfer the full Grid/Mesh/LocStream objects (with coordinates) for Field pairs that have a provider and an acceptor side.
IPDv04p5	driver-internal	*unspecified by NUOPC*
IPDv04p5	models, mediators, drivers	Realize all Fields that have TransferActionGeomObject equal to "accept" on the transferred Grid/Mesh/LocStream objects.
IPDv04p5a	connectors	Reconcile the import and export States.
IPDv04p5b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv04p6	driver-internal	*unspecified by NUOPC*
IPDv04p6	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv04p7	driver-internal	*unspecified by NUOPC*
IPDv04p7	models, mediators, drivers	Handle Field data initialization. Timestamp the export Fields.
A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:		
Run()	connectors	Loop over all Connectors that connect *to* the Component that is currently indexed by the outer loop.
IPDv04p7	models, mediators, drivers	Handle Field data initialization. Time stamp the export Fields and set the Updated and InitializeDataComplete Attributes accordingly.
Repeat these two steps until all data dependencies have been statisfied, or a dead-lock situation is detected.		

IPDv05 label	**Component**	**Meaning**
IPDv05p1	driver-internal	Advertise import and export Fields and set the TransferOfferGeomObject Attribute. Optionally set FieldTransferPolicy Attribute on States.
IPDv05p1	models, mediators, drivers	Advertise their import and export Fields and set the TransferOfferGeomObject Attribute. Optionally set FieldTransferPolicy Attribute on States.
IPDv05p1	connectors	Consider FieldTransferPolicy Attribute on import and export States. Advertise Fields to be transferred.
IPDv05p2	driver-internal	Optionally modify import and export States before connectors construct CplList Attribute.
IPDv05p2	models, mediators, drivers	Optionally modify import and export States before connectors construct CplList Attribute.
IPDv05p2a	connectors	Consider all connection possibilities for their CplList Attribute.
IPDv05p2b	connectors	Unambiguous construction of their CplList Attribute.
IPDv05p3	driver-internal	Modify the CplList Attributes on the Connectors.
IPDv05p3	models, mediators, drivers	*unspecified/unused by NUOPC*
IPDv05p3	connectors	Set the Connected Attribute on each import and export Field according to the CplList Attribute. Set the TransferActionGeomObject Attribute.

IPDv05p4	driver-internal	Realize "connected" import and export Fields that have TransferActionGeomObject equal to "provide".
IPDv05p4	models, mediators, drivers	Realize their "connected" import and export Fields that have TransferActionGeomObject equal to "provide".
IPDv05p4	connectors	Transfer the Grid/Mesh/LocStream objects (only DistGrid) for Field pairs that have a provider and an acceptor side.
IPDv05p5	driver-internal	Optionally modify the decomposition and distribution information of the accepted Grid/Mesh/LocStream by replacing the DistGrid.
IPDv05p5	models, mediators, drivers	Optionally modify the decomposition and distribution information of the accepted Grid/Mesh/LocStream by replacing the DistGrid.
IPDv05p5	connectors	Transfer the full Grid/Mesh/LocStream objects (with coordinates) for Field pairs that have a provider and an acceptor side.
IPDv05p6	driver-internal	Realize all Fields that have TransferActionGeomObject equal to "accept" on the transferred Grid/Mesh/LocStream objects.
IPDv05p6	models, mediators, drivers	Realize all Fields that have TransferActionGeomObject equal to "accept" on the transferred Grid/Mesh/LocStream objects.
IPDv05p6a	connectors	Reconcile the import and export States.
IPDv05p6b	connectors	Precompute the RouteHandle according to the CplList Attribute.
IPDv05p7	driver-internal	*unspecified by NUOPC*
IPDv05p7	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv05p8	driver-internal	*unspecified by NUOPC*
IPDv05p8	models, mediators, drivers	Handle Field data initialization. Timestamp the export Fields.
A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:		
Run()	connectors	Loop over all Connectors that connect *to* the Component that is currently indexed by the outer loop.
IPDv05p8	models, mediators, drivers	Handle Field data initialization. Time stamp the export Fields and set the Updated and InitializeDataComplete Attributes accordingly.
Repeat these two steps until all data dependencies have been satisfied, or a dead-lock situation is detected.		

7.1 NUOPC_Driver IPD implementation

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Ensure that the InitializePhaseMap and InternalInitializePhaseMap attributes are set consistent with the available NUOPC Initialize Phase Definition (IPD) versions (see section 7 for a precise definition). The default implementation uses IPDv02 for InitializePhaseMap, and sets
    - IPDv02p1 (NUOPC PROVIDED)
    - IPDv02p3 (NUOPC PROVIDED)
    - IPDv02p5 (NUOPC PROVIDED).

    The default implementation uses IPDv05 for InternalInitializePhaseMap, and sets
    - IPDv05p1 (NUOPC PROVIDED)
    - IPDv05p2 (NUOPC PROVIDED)

    * IPDv05p3 (NUOPC PROVIDED)
    * IPDv05p4 (NUOPC PROVIDED)
    * IPDv05p6 (NUOPC PROVIDED)
    * IPDv05p8 (NUOPC PROVIDED).

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - A default Initialize entry point for the higher level (e.g. application level) to initialize the Driver with a single call.
  - Internally calls into the `InitializePhaseMap`: IPDv02p1, IPDv02p3, IPDv02p5 phases in sequence.
- `InitializePhaseMap`: IPDv02p1 (NUOPC PROVIDED)
  - Allocate and initialize internal data structures.
  - If the internal clock is not yet set, set the default internal clock to be a copy of the incoming clock, but only if the incoming clock is valid.
  - *Required specialization* to set component services: `label_SetModelServices`.
    * Call `NUOPC_DriverAddComp()` for all Model, Mediator, and Connector components to be added.
    * Optionally replace the default clock.
  - Create States for all of the child GridComps.
  - Create Connectors to/from Driver component itself.
  - Set default run sequence.
  - Execute Initialize phase=0 for all Model, Mediator, and Connector components. This is the method where each component is required to initialize its `InitializePhaseMap` Attribute.
  - *Optional specialization* to analyze and modify the `InitializePhaseMap` Attribute of the child components before the Driver uses it: `label_ModifyInitializePhaseMap`.
  - *Optional specialization* to set run sequence: `label_SetRunSequence`.
  - Drive the initialize sequence for the child components, compatible with up to IPDv05, as documented in section 7, through IPDv05p3.
- `InitializePhaseMap`: IPDv02p3 (NUOPC PROVIDED)
  - Continue to drive the initialize sequence for the child components, compatible with up to IPDv05, as documented in section 7, through IPDv05p7.
- `InitializePhaseMap`: IPDv02p5 (NUOPC PROVIDED)
  - Continue to drive the initialize sequence for the child components, compatible with up to IPDv05, as documented in section 7, through IPDv05p8.
- `InternalInitializePhaseMap`: IPDv05p1 (NUOPC PROVIDED)
  - Request that fields in export and import State of child components are mirrored onto the driver's own import and export States.
  - This includes transferring the associated Grid, Mesh, or LocStream objects.
- `InternalInitializePhaseMap`: IPDv05p2 (NUOPC PROVIDED)
  - Reset the request of field mirroring.
- `InternalInitializePhaseMap`: IPDv05p3 (NUOPC PROVIDED)
  - Add the `REMAPMETHOD=redist` option to all entries in `CplList` attribute on all Connectors to/from the driver itself.

  - *Optional specialization* to modify the `CplList` attribute on all of the Connectors: `label_ModifyCplLists`.

- `InternalInitializePhaseMap`: IPDv05p4 (NUOPC PROVIDED)
  - Check that all connected fields in the driver's own import and export State have a producer connection.
- `InternalInitializePhaseMap`: IPDv05p6 (NUOPC PROVIDED)
  - Complete the allocation of all the fields in the driver's own import and export State.
- `InternalInitializePhaseMap`: IPDv05p8 (NUOPC PROVIDED)
  - Set the `InitializeDataComplete` consistent with the data-dependency protocol.

RUN:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - If the incoming clock is valid, set the internal stop time to one time step interval on the incoming clock.
  - Drive the time stepping loop, from current time to stop time, incrementing by time step.
    * For each time step iteration the Model and Connector components Run() methods are being called according to the run sequence.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - Execute the Finalize() methods of all Connector components in order.
  - Execute the Finalize() methods of all Model components in order.
  - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.
  - Destroy all Model components and their import and export states.
  - Destroy all Connector components.
  - Internal clean-up.

7.2 NUOPC_ModelBase IPD implementation

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 7 for a precise definition). The default implementation sets the following mapping:
    * IPDv00p1 = 1: (REQUIRED, IMPLEMENTOR PROVIDED)
    * IPDv00p2 = 2: (REQUIRED, IMPLEMENTOR PROVIDED)
    * IPDv00p3 = 3: (REQUIRED, IMPLEMENTOR PROVIDED)
    * IPDv00p4 = 4: (REQUIRED, IMPLEMENTOR PROVIDED)

RUN:

- phase 1: (NUOPC PROVIDED)
  - SPECIALIZATION REQUIRED/PROVIDED: `label_SetRunClock` to check and set the internal Clock against the incoming Clock.
    - `IF` (Phase specific specialization present): Execute the phase specific specialization.
    - `ELSE`: Execute the phase independent specialization. PROVIDED: By default check that internal Clock and incoming Clock agree on current time and that the time step of the incoming Clock is a multiple of the internal Clock time step. Under these conditions set the internal stop time to one time step interval of the incoming Clock. Otherwise exit with error, flagging an incompatibility.
  - SPECIALIZATION REQUIRED/PROVIDED: `label_CheckImport` to check Fields in the import State.
    - `IF` (Phase specific specialization is present): Execute the phase specific specialization.
    - `ELSE`: Execute the phase independent specialization. PROVIDED: By default check that all import Fields are at the current time of the internal Clock.
  - Time stepping loop: starting at current time, running to stop time of the internal Clock.
    - Timestamp the Fields in the export State according to the current time of the internal Clock.
    - SPECIALIZATION REQUIRED: `label_Advance` to execute model or mediation code.
    - SPECIALIZATION OPTIONAL: `label_AdvanceClock` to advance the current time of the internal Clock. By default (without specialization) advance the current time of the internal Clock according to the time step of the internal Clock.
  - SPECIALIZATION OPTIONAL: `label_TimestampExport` to timestamp Fields in the export State.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.

7.3 NUOPC_Model IPD implementation

INITIALIZE:

- phase 0: Set Initialize Phase Definition Version (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 7 for a precise definition). The default implementation sets the following mapping:
    - IPDv00p1 = 1: (REQUIRED, IMPLEMENTOR PROVIDED)
      - Advertise Fields in import and export States.
    - IPDv00p2 = 2: (REQUIRED, IMPLEMENTOR PROVIDED)
      - Realize the advertised Fields in import and export States.
    - IPDv00p3 = 3: (REQUIRED, NUOPC PROVIDED)
      - Check compatibility of the Fields' Connected status.
    - IPDv00p4 = 4: (REQUIRED, NUOPC PROVIDED)
      - Handle Field data initialization. Time stamp the export Fields.

- IPDv00p1, IPDv01p1, IPDv02p1, IPDv03p1, IPDv04p1, IPDv05p1: Advertise fields in import and export States (REQUIRED, IMPLEMENTOR PROVIDED)

  – Advertise fields in import/export states using one of the two `NUOPC_Advertise` methods (**??**, **??**). The methods require Standard Names for each field, and the Standard Names must appear in the NUOPC Field Dictionary or a runtime error is generated. `NUOPC_Advertise` accepts a `TransferOfferGeomObject` argument which may be one of:

    * "will provide" (default) - The field will provide its own geometric object (i.e., Grid, Mesh, or LocStream)
    * "can provide"- The field can provide its own geometric object, but only if the connected field in the other component will not provide it
    * "cannot provide" - The field cannot provide its own geometric object. It must accept a geometric object from a connected field.

    See section 2.4.7 for more details about transferring geometric objects between NUOPC components. Memory is not allocated for advertised fields, but attributes are set on the field which can be used in later phases, especially for determining if another component can provide and/or consume the advertised field.

- IPDv00p2, IPDv01p3, IPDv02p3, IPDv03p3, IPDv04p3, IPDv05p4: Realize field *providing* a geometric object (REQUIRED*, IMPLEMENTOR PROVIDED)

  – Realize connected import and export fields that have their TransferActionGeomObject attribute set to "provide", i.e., that will provide their own geometric object (i.e., Grid, Mesh, or LocStream). "provide" is the default value of TransferActionGeomObject. Realize means an ESMF_Field object is created on a geometric object and memory for the field is allocated or referenced.

    The `NUOPC_Realize` methods (**??**, **??**, **??**, **??**, **??**) are used to realize fields. Only previously advertised fields can be realized and the field's name is used to search the state for the previously advertised field.

    *Note: This phase is not required if all fields are *accepting* a geometric object.

- IPDv03p4, IPDv04p4, IPDv05p5: Modify decomposition of accepted geometric object (OPTIONAL, IMPLEMENTOR PROVIDED)

  – Optionally modify the decomposition information of any accepted geometric object by replacing the DistGrid. In the case of the Grid geometric object, this can be accomplished by retrieving the Grid (and its DistGrid) from the Field, creating a new DistGrid with modified decomposition, creating a new Grid on the new (modified) DistGrid, and then using `ESMF_FieldEmptySet` to replace the existing Grid with the new one.

    This phase is useful when accepting a Grid from another component, but when the PET counts differ between components. In this case, a new decomposition needs to be set based on the current processor count.

- IPDv03p5, IPDv04p5, IPDv05p6: Realize fields *accepting* a geometric object (REQUIRED*, IMPLEMENTOR PROVIDED)

  – Realize connected import and export fields that have their TransferActionGeomObject attribute set to "accept", i.e., that will accept a geometric object from a connected field in another component. If the generic `NUOPC_Connector` is used, at this point the full geometric object has already been set in the field and only a call to `ESMF_FieldEmptyComplete` is required to allocate memory for the field.

    The `NUOPC_Realize` methods (**??**, **??**, **??**, **??**, **??**) are used to realize fields. Only previously advertised fields can be realized and the field's name is used to search the state for the previously advertised field.

    *Note: This phase is not required if all fields are *providing* a geometric object.

- IPDv00p3, IPDv01p4, IPDv02p4, IPDv03p6, IPDv04p6, IPDv05p7: Verify import fields connected and set clock (NUOPC PROVIDED)

  – If the model internal clock is found to be not set, then set the model internal clock as a copy of the incoming clock.

  - *Optional specialization* to set the internal clock and/or alarms: `label_SetClock`.
  - Check compatibility, ensuring all advertised import Fields are connected.
- IPDv00p4, IPDv01p5: Initialize export fields (NUOPC PROVIDED)
  - *Optional specialization* to initialize export Fields: `label_DataInitialize`
  - Time stamp Fields in export State for compatibility checking.
- IPDv02p5, IPDv03p7, IPDv04p7, IPDv05p8: Initialize export fields (NUOPC PROVIDED)
  - *Optional specialization* to initialize export Fields: `label_DataInitialize`
  - Timestamp Fields in export State for compatibility checking.
  - Set Component metadata used to resolve initialize data dependencies.

RUN:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - SPECIALIZATION REQUIRED/PROVIDED: `label_SetRunClock` to check and set the internal Clock against the incoming Clock.
    - `IF` (Phase specific specialization present): Execute the phase specific specialization.
    - `ELSE`: Execute the phase independent specialization. PROVIDED: By default check that internal Clock and incoming Clock agree on current time and that the time step of the incoming Clock is a multiple of the internal Clock time step. Under these conditions set the internal stop time to one time step interval of the incoming Clock. Otherwise exit with error, flagging an incompatibility.
  - SPECIALIZATION REQUIRED/PROVIDED: `label_CheckImport` to check Fields in the import State.
    - `IF` (Phase specific specialization is present): Execute the phase specific specialization.
    - `ELSE`: Execute the phase independent specialization. PROVIDED: By default check that all import Fields are at the current time of the internal Clock.
  - Time stepping loop: starting at current time, running to stop time of the internal Clock.
    - Timestamp the Fields in the export State according to the current time of the internal Clock.
    - SPECIALIZATION REQUIRED: `label_Advance` to execute model code.
    - SPECIALIZATION OPTIONAL: `label_AdvanceClock` to advance the current time of the internal Clock. By default (without specialization) advance the current time of the internal Clock according to the time step of the internal Clock.
  - SPECIALIZATION OPTIONAL/PROVIDED: `label_TimestampExport` to timestamp Fields in the export State.
    - `IF` (Phase specific specialization present): Execute the phase specific specialization.
    - `ELSE`: Execute the phase independent specialization. PROVIDED: Timestamp all Fields in the export State according to the current time of the internal Clock, which now is identical to the stop time of the internal Clock.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.

7.3.1 Initialize Phase Specialization - label_SetClock

OPTIONAL, IMPLEMENTOR PROVIDED
Called from: IPDv00p3, IPDv01p4, IPDv02p4, IPDv03p6, IPDv04p6, IPDv05p7

The specialization method can change aspects of the internal clock, which defaults to a copy of the incoming parent clock. For example, the timeStep size may be changed and/or Alarms may be set on the clock.

The method `NUOPC_CompSetClock(comp, externalClock, stabilityTimeStep)` (**??**) can be used to set the internal clock as a copy of externalClock, but with a timeStep that is less than or equal to the stabilityTimeStep. At the same time it ensures that the timeStep of the external clock is a multiple of the timeStep of the internal clock. If the stabilityTimeStep argument is not provided then the internal clock will simply be set as a copy of the external clock.

7.3.2 Initialize Phase Specialization - label_DataInitialize

OPTIONAL, IMPLEMENTOR PROVIDED
Called from: IPDv00p4, IPDv01p5, IPDv02p5, IPDv03p7, IPDv04p7, IPDv05p8

The specialization method should initialize field data in the export state. Fields in the export state will be timestamped automatically by the calling phase for all fields that have the "Updated" attribute set to "true".

7.3.3 Run Phase Specialization - label_SetRunClock

REQUIRED, NUOPC PROVIDED
Called from: default run phase

A specialization method to check and set the internal clock against the incoming clock. This method is called by the default run phase.

If not overridden, the default method will check that the internal clock and incoming clock agree on the current time and that the time step of the incoming clock is a multiple of the internal clock time step. Under these conditions set the internal stop time to one time step interval of the incoming clock. Otherwise exit with error, flagging an incompatibility.

7.3.4 Run Phase Specialization - label_CheckImport

REQUIRED, NUOPC PROVIDED
Called from: default run phase

A specialization method to verify import fields before advancing in time. If not overridden, the default method verifies that all import fields are at the current time of the internal clock.

7.3.5 Run Phase Specialization - label_Advance

REQUIRED, IMPLEMENTOR PROVIDED
Called from: default run phase

A specialization method that advances the model forward in time by one timestep of the internal clock. This method will be called iteratively by the default run phase until reaching the stop time on the internal clock.

7.3.6 Run Phase Specialization - label_TimestampExport

REQUIRED, NUOPC PROVIDED
Called from: default run phase

A specialization method to set the timestamp on export fields after the model has advanced. If not overridden, the default method sets the timestamp on all export fields to the stop time on the internal clock (which is also now the current model time).

7.3.7 Finalize Phase Specialization - label_Finalize

OPTIONAL, IMPLEMENTOR PROVIDED
Called from: default finalize phase

An optional specialization method for custom finalization code and deallocations of user data structures.

7.4 NUOPC_Mediator IPD implementation

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 7 for a precise definition). The default implementation sets the following mapping:
    - IPDv00p1 = 1: (REQUIRED, IMPLEMENTOR PROVIDED)
      - Advertise Fields in import and export States.
    - IPDv00p2 = 2: (REQUIRED, IMPLEMENTOR PROVIDED)
      - Realize the advertised Fields in import and export States.
    - IPDv00p3 = 3: (REQUIRED, NUOPC PROVIDED)
      - Check compatibility of the Fields' Connected status.
    - IPDv00p4 = 4: (REQUIRED, NUOPC PROVIDED)
      - Handle Field data initialization. Time stamp the export Fields.
- IPDv00p3, IPDv01p4, IPDv02p4: (NUOPC PROVIDED)
  - Set the Mediator internal clock as a copy of the incoming clock.
  - Check compatibility, ensuring all advertised import Fields are connected.
- IPDv00p4, IPDv01p5: (NUOPC PROVIDED)
  - *Optional specialization* to initialize export Fields: `label_DataInitialize`
  - Time stamp Fields in import and export States for compatibility checking.

- IPDv02p5: (NUOPC PROVIDED)
  - *Optional specialization* to initialize export Fields: `label_DataInitialize`
  - Time stamp Fields in export State for compatibility checking.
  - Set Component metadata used to resolve initialize data dependencies.

RUN:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - SPECIALIZATION REQUIRED/PROVIDED: `label_SetRunClock` to check and set the internal Clock against the incoming Clock.
    - `IF` (Phase specific specialization present): Execute the phase specific specialization.
    - `ELSE`: Execute the phase independent specialization. PROVIDED: By default check that internal Clock and incoming Clock agree on current time and that the time step of the incoming Clock is a multiple of the internal Clock time step. Under these conditions set the internal stop time to one time step interval of the incoming Clock. Otherwise exit with error, flagging an incompatibility.
  - SPECIALIZATION REQUIRED/PROVIDED: `label_CheckImport` to check Fields in the import State.
    - `IF` (Phase specific specialization is present): Execute the phase specific specialization.
    - `ELSE`: Execute the phase independent specialization. PROVIDED: By default check that all import Fields are at the current time of the internal Clock.
  - Time stepping loop: starting at current time, running to stop time of the internal Clock.
    - Timestamp the Fields in the export State according to the current time of the internal Clock.
    - SPECIALIZATION REQUIRED: `label_Advance` to execute mediation code.
    - SPECIALIZATION OPTIONAL: `label_AdvanceClock` to advance the current time of the internal Clock. By default (without specialization) advance the current time of the internal Clock according to the time step of the internal Clock.
  - SPECIALIZATION OPTIONAL/PROVIDED: `label_TimestampExport` to timestamp Fields in the export State.
    - `IF` (Phase specific specialization present): Execute the phase specific specialization.
    - `ELSE`: Execute the phase independent specialization. PROVIDED: Timestamp all Fields in the export State according to the current time of the internal Clock when *entering* the RUN method.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.

7.5 NUOPC_Connector IPD implementation

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 04 (see section 7 for a precise definition). The default implementation sets the following mapping:

    * IPDv04p1a = phase : (REQUIRED, NUOPC PROVIDED)
    * IPDv04p1b = phase : (REQUIRED, NUOPC PROVIDED)
    * IPDv04p2 = phase : (REQUIRED, NUOPC PROVIDED)
    * IPDv04p3 = phase : (REQUIRED, NUOPC PROVIDED)
    * IPDv04p4 = phase : (REQUIRED, NUOPC PROVIDED)
    * IPDv04p5a = phase : (REQUIRED, NUOPC PROVIDED)
    * IPDv04p5b = phase : (REQUIRED, NUOPC PROVIDED)

- IPDv01p1, IPDv02p1: (NUOPC PROVIDED)
  - Construct a list of matching Field pairs between import and export State based on the `StandardName` Field metadata.
  - Store this list of `StandardName` entries in the `CplList` attribute of the Connector Component metadata.
- IPDv01p2, IPDv02p2: (NUOPC PROVIDED)
  - Allocate and initialize the internal state.
  - Use the `CplList` attribute to construct `srcFields` and `dstFields` FieldBundles in the internal state that hold matched Field pairs.
  - Set the `Connected` attribute to `true` in the Field metadata for each Field that is added to the `srcFields` and `dstFields` FieldBundles.
- IPDv01p3, IPDv02p3: (NUOPC PROVIDED)
  - Use the `CplList` attribute to construct `srcFields` and `dstFields` FieldBundles in the internal state that hold matched Field pairs.
  - Set the `Connected` attribute to `true` in the Field metadata for each Field that is added to the `srcFields` and `dstFields` FieldBundles.
  - *Optional specialization* to precompute a Connector operation: `label_ComputeRouteHandle`. Simple custom implementations store the precomputed communication RouteHandle in the `rh` member of the internal state. More complex implementations use the `state` member in the internal state to store auxiliary Fields, FieldBundles, and RouteHandles.
  - By default (if `label_ComputeRouteHandle` was *not* provided) precompute the Connector RouteHandle as a bilinear Regrid operation between `srcFields` and `dstFields`, with `unmappedaction` set to `ESMF_UNMAPPEDACTION_IGNORE`. The resulting RouteHandle is stored in the `rh` member of the internal state.

RUN:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - *Optional specialization* to execute a Connector operation: `label_ExecuteRouteHandle`. Simple custom implementations access the `srcFields`, `dstFields`, and `rh` members of the internal state to implement the required data transfers. More complex implementations access the `state` member in the internal state, which holds the auxiliary Fields, FieldBundles, and RouteHandles that potentially were added during the optional `label_ComputeRouteHandle` method during initialize.
  - By default (if `label_ExecuteRouteHandle` was *not* provided) execute the precomputed Connector RouteHandle between `srcFields` and `dstFields`.
  - Update the time stamp on the Fields in `dstFields` to match the time stamp on the Fields in `srcFields`.

FINALIZE:

- phase 1: (REQUIRED, NUOPC PROVIDED)
  - *Optional specialization* to release the custom Connector operation: `label_ReleaseRouteHandle`; or by default, if `label_ReleaseRouteHandle` was *not* provided, release the default Connector RouteHandle.
  - *Optional specialization* to finalize custom parts of the component: `label_Finalize`.
  - Internal clean-up.