National Unified Operational Prediction Capability

NUOPC Layer Reference

ESMF v7.0.1

Content Standards Committee (CSC) Members

December 13, 2016

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

Contents

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 straight forward 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 flavor: `Driver`, `Model`, `Mediator`, or `Connector`. This is optionally followed by a string of additional descriptive terms. The four flavors 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 the entries of an imported generic component module in the local scope as part of the `use` association. This prevents name clashes.

```
use NUOPC_<GenericComp>, only: &
  <GenericComp>_SS      => SetServices, &
  <GenericComp>_labelA  => labelA
```

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 before doing anything else. It is through this mechanism that the deriving component *inherits* functionality that is implemented in the generic component. The example shows how a specific model component is implemented to derive from the generic `NUOPC_Model`:

```
use NUOPC_Model, only: &
  model_SS => SetServices

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

  ! derive this specific "model" component from generic NUOPC_Model
  call NUOPC_CompDerive(model, model_SS, rc=rc)

  ! specializing code for "model" to follow

end subroutine
```

There are three mechanisms through which user code specializes generic components. The first two methods specialize through call-back mechanisms into user implemented routines. The third method specializes by setting the values of parameters implemented by the generic component.

1. The specializing user code sets entry points for standard component methods not implemented by the generic component by calling `NUOPC_CompSetEntryPoint()`. Methods (and phases) that need to be implemented are clearly documented in the generic component description. The user code may further overwrite standard methods already implemented by the generic component code. However, this should rarely be necessary, and may indicate that there is a better fitting generic component available. Finally, some generic components come with generic routines that are suitable candidates for the standard component methods, yet require that the specializing code registers them as appropriate. Setting entry points for standard component methods should be done in the `SetServices()` routine right after `NUOPC_CompDerive()` is called.

2. Some generic components require that specific methods are attached to the component by calling `NUOPC_CompSpecialize()`. If a generic component uses specialization through attachable methods, the specific method labels (i.e. the names by which these methods are registered) and the purpose of the method are clearly documented. In some cases attachable methods are optional. This is clearly documented. Further, some generic components attach a default method to a label, which then is used for all phases. This default can be overwritten with a phase specific attachable method. Attaching methods to the component should be done in the `SetServices` routine right after setting entry points for the standard component methods.

3. Some generic components offer methods that allow parameter specialization. The options of this specialization type are documented as part of the `Set()` methods.

Components that inherit 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`.

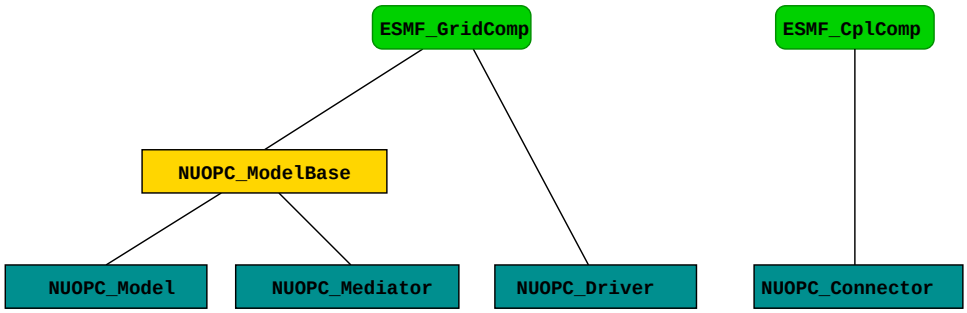

Figure 1: The NUOPC Generic Component inheritance structure. The tree on the left is rooted in `ESMF_GridComp`, while the tree on the right is rooted in `ESMF_CplComp`. The ESMF data types are shown in green. The four main NUOPC Generic Component flavors are shown in dark blue boxes. The yellow box shows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

2.2 Field Dictionary

The NUOPC Layer uses standard metadata on Fields to guide the decision making 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`.

The NUOPC Layer provides a number of default entries in the Field Dictionary, shown in the table below. The `StandardName` Attribute of these entries complies with the Climate and Forecast (CF) conventions as documented at http://cfconventions.org/Data/cf-standard-names/docs/guidelines.html.

Currently it is common practice that a user of the NUOPC Layer extends the Field Dictionary by calling the `NUOPC_FieldDictionaryAddEntry()` interface to add additional entries. It is the intention to increase the number of default entries over time, and to more strongly leverage the NUOPC Field Dictionary as a means to increase the interoperability between codes that use the NUOPC Layer.

Currently the NUOPC Layer uses the `CanonicalUnits` entry to verify that Fields are provided in their canonical units. The plan is to extend support to include unit conversion in the future.

StandardName	**CanonicalUnits**
air_pressure_at_sea_level	Pa
magnitude_of_surface_downward_stress	Pa
precipitation_flux	kg m-2 s-1
sea_surface_height_above_sea_level	m
sea_surface_salinity	1e-3
sea_surface_temperature	K
surface_eastward_sea_water_velocity	m s-1
surface_downward_eastward_stress	Pa
surface_downward_heat_flux_in_air	W m-2
surface_downward_water_flux	kg m-2 s-1
surface_downward_northward_stress	Pa
surface_net_downward_shortwave_flux	W m-2
surface_net_downward_longwave_flux	W m-2
surface_northward_sea_water_velocity	m s-1

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 as an ESMF Attribute Package:

- Convention: NUOPC

- Purpose: Instance
- Includes:
  - CIM Model Component Simulation Description (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)

Attribute name	Definition	Controlled vocabulary
`Kind`	String value indicating component kind.	Driver
`Verbosity`	String value controlling the verbosity of the component. This attribute is used by the generic code layer but should also be considered by any specializing code that implements verbose diagnostic output.	0, 1, ... max
`Profiling`	String value controlling the profiling level. This attribute is used by the generic code layer but should also be considered by any specializing code that implements profiling.	0, 1, ... max
`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.
`InternalInitializePhaseMap`	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, ...
`InitializeDataComplete`	String value indicating whether all initialize data dependencies have been satisfied.	false, true
`InitializeDataProgress`	String value indicating whether progress is being made resolving initialize data dependencies.	false, true

2.3.2 Model Component Metadata

The Model component metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance
- Includes:
  - CIM Model Component Simulation Description (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)

Attribute name	Definition	Controlled vocabulary
`Kind`	String value indicating component kind.	Model
`Verbosity`	String value controlling the verbosity of the component. This attribute is used by the generic code layer but should also be considered by any specializing code that implements verbose diagnostic output.	0, 1, ... max
`Profiling`	String value controlling the profiling level. This attribute is used by the generic code layer but should also be considered by any specializing code that implements profiling.	0, 1, ... max
`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.
`InternalInitializePhaseMap`	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, ...
`InitializeDataComplete`	String value indicating whether all initialize data dependencies have been satisfied.	false, true
`InitializeDataProgress`	String value indicating whether progress is being made resolving initialize data dependencies.	false, true

2.3.3 Mediator Component Metadata

The Mediator component metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance
- Includes:
  - CIM Model Component Simulation Description (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)

Attribute name	Definition	Controlled vocabulary
`Kind`	String value indicating component kind.	Mediator
`Verbosity`	String value controlling the verbosity of the component. This attribute is used by the generic code layer but should also be considered by any specializing code that implements verbose diagnostic output.	0, 1, ... max
`Profiling`	String value controlling the profiling level. This attribute is used by the generic code layer but should also be considered by any specializing code that implements profiling.	0, 1, ... max
`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.
`InternalInitializePhaseMap`	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, ...
`InitializeDataComplete`	String value indicating whether all initialize data dependencies have been satisfied.	false, true
`InitializeDataProgress`	String value indicating whether progress is being made resolving initialize data dependencies.	false, true

2.3.4 Connector Component Metadata

The Connector Component metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: General
- Includes:
  - ESG General (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)

Attribute name	Definition	Controlled vocabulary
`Kind`	String value indicating component kind.	Connector
`Verbosity`	String value controlling the verbosity of the component. This attribute is used by the generic code layer but should also be considered by any specializing code that implements verbose diagnostic output.	0, 1, ... max
`Profiling`	String value controlling the profiling level. This attribute is used by the generic code layer but should also be considered by any specializing code that implements profiling.	0, 1, ... max
`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.

2.3.5 State Metadata

The State metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance

- Includes:
  - *no other packages at the moment*
- Description: Namespace implementation for import and export.

Attribute name	Definition	Controlled vocabulary
`Namespace`	String value holding the namespace of all the objects contained in the State.	*no restriction*

2.3.6 Field Metadata

The Field metadata is implemented as an ESMF Attribute Package:

- Convention: NUOPC
- Purpose: Instance
- Includes:
  - ESG General
- Description: Basic Field description with connection and time stamp metadata.

Attribute name	Definition	Controlled vocabulary
`Connected`	Connected status.	false, true
`TimeStamp`	Nine integer values representing ESMF Time object.	N/A
`ProducerConnection`	String value indicating connection details.	open, targeted, connected
`ConsumerConnection`	String value indicating connection details.	open, targeted, connected
`Updated`	String value indicating updated status during initialization.	false, true
`TransferOfferGeomObject`	String value indicating a component's intention to transfer the underlying Grid or Mesh on which an advertised Field object is defined.	will provide, can provide, cannot provide
`TransferActionGeomObject`	String value indicating the action a component is supposed to take with respect to transferring the underlying Grid or Mesh on which an advertised Field object is defined.	provide, accept

2.4 Initialization

2.4.1 Phase Maps and Component Labels

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. The NUOPC layer adds an abstraction layer that allows phases to be referenced by label.

For complex scenarios, e.g. multiple versions of multi-stage initialize sequences, the use of an integer based phase index quickly becomes confusing. The NUOPC Layer addresses this issue by 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.

Currently the NUOPC Layer leverages the `InitializePhaseMap` during the initialization loop that is implemented by the generic `NUOPC_Driver`. It looks for phase map entries according to the initialize phase definition outlined in section 2.4.2. 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 within the NUOPC Layer.

Within NUOPC, components under a driver are also referenced by label. Every 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 Initialize Phase Definitions

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 same sequence in which the driver calls them.

IPDv00 label	Component	Meaning
IPDv00p1	driver-self	*unspecified by NUOPC*
IPDv00p1	models, mediators, drivers	Advertise their import and export Fields.
IPDv00p1	connectors	Construct their CplList Attribute.
IPDv00p2	driver-self	*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-self	*unspecified by NUOPC*
IPDv00p3	models, mediators, drivers	Check for compatibility of their Fields' Connected status.
IPDv00p4	driver-self	*unspecified by NUOPC*
IPDv00p4	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields.

IPDv01 label	Component	Meaning
IPDv01p1	driver-self	*unspecified by NUOPC*
IPDv01p1	models, mediators, drivers	Advertise their import and export Fields.
IPDv01p1	connectors	Construct their CplList Attribute.
IPDv01p2	driver-self	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-self	*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-self	*unspecified by NUOPC*
IPDv01p4	models, mediators, drivers	Check for compatibility of their Fields' Connected status.
IPDv01p5	driver-self	*unspecified by NUOPC*
IPDv01p5	models, mediators, drivers	Handle Field data initialization. Timestamp their export Fields.

IPDv02 label	Component	Meaning
IPDv02p1	driver-self	*unspecified by NUOPC*
IPDv02p1	models, mediators, drivers	Advertise their import and export Fields.

IPDv02p1	connectors	Construct their CplList Attribute.
IPDv02p2	driver-self	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-self	*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-self	*unspecified by NUOPC*
IPDv02p4	models, mediators, drivers	Check for compatibility of their Fields' Connected status.
IPDv02p5	driver-self	*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-self	*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-self	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-self	*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-self	*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-self	*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-self	*unspecified by NUOPC*
IPDv03p6	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv03p7	driver-self	*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-self	*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-self	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-self	*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-self	*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-self	*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-self	*unspecified by NUOPC*
IPDv04p6	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv04p7	driver-self	*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-self	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-self	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-self	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-self	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-self	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-self	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-self	*unspecified by NUOPC*
IPDv05p7	models, mediators, drivers	Check compatibility of their Fields' Connected status.
IPDv05p8	driver-self	*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 statisfied, or a dead-lock situation is detected.

2.4.3 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 exact handshakes between these components is outlined as part of the Initialize Phase Definition in section 2.4.2.

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 2.4.4. 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.

2.4.4 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 2.4.3, 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 mis-match. 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.

2.4.5 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 internal `IPDv04p2` phase (see 2.4.2) 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**
`srcMaskValues`	List of integer values that defines the mask values.	multi	List of integers.
`dstMaskValues`	List of integer values that defines the mask values.	multi	List of integers.
`remapmethod`	Redistribution or interpolation to compute the regridding weights.	single	`redist`, `bilinear`(default), `patch`, `nearest_stod`, `nearest_dtos`, `conserve`
`polemethod`	Extrapolation method around the pole(s).	single	`none`(default), `allavg`, `npntavg`=*"integer indicating number of points"*
`unmappedaction`	The action to take when unmapped destination elements are encountered.	single	`ignore`(default), `error`
`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`
`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
`dumpWeightsFlag`	Enable or disable dumping of the interpolation weights into a file.	single	`true`, `false`(default)

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 is implemented as part of Initialize Phase Definition version 2 (IPDv02) as defined in section 2.4.2. Participating components communicate their current status to the NUOPC Layer via Field and Component metadata. Participants are those components that contain an `IPDv02p5` assignment in their `InitializePhaseMap` Attribute according to sections 2.3.1, 2.3.2, 2.3.3, and 2.3.4.

Every time a component's `IPDv02p5` initialization phase is called it is responsible for setting the `InitializeDataComplete` and `InitializeDataProgress` Attributes according to its current status before returning. For convenience, the NUOPC Layer provides a generic implementation of an `IPDv02p5` phase initialize method for Models and Mediators (available as ESMF Initialize phase 5). This generic implementation takes care of setting the `InitializeDataProgress` Attribute automatically. It does so by inspecting the `Updated` Field Attribute (see section 2.3.6) on all the Fields in the component's exportState. The generic `IPDv02p5` implementation must be specialized by attaching a method for specialization point `label_DataInitialize`. This specialization method 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". 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 `IPDv02p5` method. Doing so ensures that all the currently available Fields are passed to the component before it tries to access them during `IPDv02p5`. 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 with an `IPDv02p5` initialization phase 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. The Initialize Phase Definition version 3 (IPDv03) and above, defined in section 2.4.2, support the transfer of `ESMF_Grid`, and `ESMF_Mesh`, and `ESMF_LocStream` objects between Model and/or Mediator components during initialization.

The NUOPC Layer transfer protocol for GeomObjects (i.e. ESMF Grids, Meshes, or LocStreams) 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 `IPDv03p1`.

The generic Connector uses the intents from both sides and constructs a response according to the table below. The response is provided by the Connector during `IPDv03p2` by setting 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 initialize method for phase `IPDv03p3`.

Fields that hold "accept" for the value of the `TransferActionGeomObject` attribute require two additional negotiation steps. By `IPDv03p4` 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 `IPDv03p4` 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 `IPDv03p5`, where they can be used to realize the respective Field objects. Realizing typically just requires the `ESMF_FieldEmptyComplete()` call to be made. 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 Mirroring

In some cases it is helpful for a NUOPC component to automatically mirror or match the set of fields advertised in another component. 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. This is currently used in the NUOPC Component

Explorer: when driving a child NUOPC Model with required import fields, the Component Explorer uses the field mirroring capability to advertise in the driver-self export State the exact set of fields advertised in the child NUOPC Model. This ensures that the entire Initialize Phase Sequence will complete (because all dependencies are satisfied) and all phases can be exercised by the Component Explorer.

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.

The field mirror capability is supported in the Initialize Phase Definition version 5 (IPDv05) and higher, defined in section 2.4.2. During `IPDv05p1`, driver-self, models, mediators, and drivers advertise fields in their import and export States. At this point, these components can also optionally set the `FieldTransferPolicy` Attribute in their import and export States. The default value "TransferNone" indicates that no fields should be mirrored. The other option, "TransferAll", indicates that fields should be mirrored in the State of a connected component.

During `IPDv05p1`, each Connector consider the `FieldTransferPolicy` Attribute on both its import and export States. If *both* States have a `FieldTransferPolicy` of "TransferAll", then fields are transferred between the States 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. Intuitively, 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. Then, the field to be mirrored is advertised in the accepting State using a call to `NUOPC_Advertise()` such that the mirrored field shares the same Standard Name.

At this point the Initialization Sequence continues as usual. Since fields to be mirrored have been advertised with matching Standard Names, the field pairing algorithm will now match them in the usual way thereby establishing a connection between the original and mirrored fields.

3 API

3.1 Generic Component: NUOPC_Driver

MODULE:

```
module NUOPC_Driver
```

DESCRIPTION:
Component that drives 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
```

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 2.4.2 for a precise definition). The default implementation sets the following mapping:
    - IPDv00p1 = 1: (REQUIRED, NUOPC PROVIDED)
- IPDv00p1 (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.
  - *Optional specialization* to set run sequence: `label_SetRunSequence`.
  - 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`.
  - Drive the initialize sequence for the child components, compatible with up to IPDv04, as documented in section 2.4.2.

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.

3.1.1 NUOPC_DriverAddComp - Add a GridComp child to a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverAddComp()
subroutine NUOPC_DriverAddGridComp(driver, compLabel, &
  compSetServicesRoutine, petList, comp, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                         :: driver
character(len=*),    intent(in)             :: compLabel
interface
  subroutine compSetServicesRoutine(gridcomp, rc)
    use ESMF
    implicit none
    type(ESMF_GridComp)        :: gridcomp ! must not be optional
    integer, intent(out)       :: rc       ! must not be optional
  end subroutine
end interface
integer,             intent(in),  optional :: petList(:)
type(ESMF_GridComp), intent(out), optional :: comp
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Create and add a GridComp (i.e. Model, Mediator, or Driver) as a child component to a Driver. The component is created on the provided `petList`, or by default across all of the Driver PETs.

The specified `SetServices()` routine is called back immediately after the new child component has been created internally. Very little around the component is set up at that time (e.g. component attributes are not available). The routine should therefore be very light weight, with the sole purpose of setting the entry points of the component – typically by deriving from a generic component followed by the appropriate specilizations.

The `compLabel` must uniquely identify the child component within the context of the Driver component.

If the `comp` argument is specified, it will reference the newly created component on return.

3.1.2 NUOPC_DriverAddComp - Add a GridComp child from shared object to a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverAddComp()
subroutine NUOPC_DriverAddGridCompSO(driver, compLabel, &
  sharedObj, petList, comp, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                        :: driver
  character(len=*),    intent(in)            :: compLabel
  character(len=*),    intent(in),  optional :: sharedObj
  integer,             intent(in),  optional :: petList(:)
  type(ESMF_GridComp), intent(out), optional :: comp
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Create and add a GridComp (i.e. Model, Mediator, or Driver) as a child component to a Driver. The component is created on the provided `petList`, or by default across all of the Driver PETs.

The `SetServices()` routine in the `sharedObj` is called back immediately after the new child component has been created internally. Very little around the component is set up at that time (e.g. component attributes are not available). The routine should therefore be very light weight, with the sole purpose of setting the entry points of the component – typically by deriving from a generic component followed by the appropriate specilizations.

The `compLabel` must uniquely identify the child component within the context of the Driver component.

If the `comp` argument is specified, it will reference the newly created component on return.

3.1.3 NUOPC_DriverAddComp - Add a CplComp child to a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverAddComp()
subroutine NUOPC_DriverAddCplComp(driver, srcCompLabel, dstCompLabel, &
  compSetServicesRoutine, petList, comp, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp)                         :: driver
     character(len=*),    intent(in)             :: srcCompLabel
     character(len=*),    intent(in)             :: dstCompLabel
     interface
       subroutine compSetServicesRoutine(cplcomp, rc)
         use ESMF
         implicit none
         type(ESMF_CplComp)         :: cplcomp  ! must not be optional
         integer, intent(out)       :: rc       ! must not be optional
       end subroutine
     end interface
     integer, target,     intent(in),  optional :: petList(:)
     type(ESMF_CplComp),  intent(out), optional :: comp
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Create and add a CplComp (i.e. Connector) as a child component to a Driver. The component is created on the provided `petList`, or by default across the union of PETs of the components indicated by `srcCompLabel` and `dstCompLabel`.

The specified `SetServices()` routine is called back immediately after the new child component has been created internally. Very little around the component is set up at that time (e.g. component attributes are not available). The routine should therefore be very light weight, with the sole purpose of setting the entry points of the component – typically by deriving from a generic component followed by the appropriate specilizations.

The `compLabel` must uniquely identify the child component within the context of the Driver component.

If the `comp` argument is specified, it will reference the newly created component on return.

3.1.4 NUOPC_DriverAddRunElement - Add RunElement for Model, Mediator, or Driver

INTERFACE:

```
   ! Private name; call using NUOPC_DriverAddRunElement()
   subroutine NUOPC_DriverAddRunElementMPL(driver, slot, compLabel, &
     phaseLabel, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp)                         :: driver
     integer,             intent(in)             :: slot
     character(len=*),    intent(in)             :: compLabel
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),    intent(in),  optional :: phaseLabel
     logical,             intent(in),  optional :: relaxedflag
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Add an element associated with a Model, Mediator, or Driver component to the run sequence of the Driver. The component must have been added to the Driver, and associated with `compLabel` prior to this call.

If `phaseLabel` was not specified, the first entry in the `RunPhaseMap` attribute of the referenced component will be used to determine the run phase of the added element.

By default an error is returned if no component is associated with the specified `compLabel`. This error can be suppressed by setting `relaxedflag=.true.`, and no entry will be added to the run sequence.

The `slot` number identifies the run sequence time slot in case multiple sequences are available. Slots start counting from 1.

3.1.5 NUOPC_DriverAddRunElement - Add RunElement for Connector

INTERFACE:

```
   ! Private name; call using NUOPC_DriverAddRunElement()
   subroutine NUOPC_DriverAddRunElementCPL(driver, slot, srcCompLabel, &
     dstCompLabel, phaseLabel, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp)                         :: driver
     integer,             intent(in)             :: slot
     character(len=*),    intent(in)             :: srcCompLabel
     character(len=*),    intent(in)             :: dstCompLabel
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),    intent(in),  optional :: phaseLabel
     logical,             intent(in),  optional :: relaxedflag
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Add an element associated with a Connector component to the run sequence of the Driver. The component must have been added to the Driver, and associated with `srcCompLabel` and `dstCompLabel` prior to this call.

If `phaseLabel` was not specified, the first entry in the `RunPhaseMap` attribute of the referenced component will be used to determine the run phase of the added element.

By default an error is returned if no component is associated with the specified `compLabel`. This error can be suppressed by setting `relaxedflag=.true.`, and no entry will be added to the run sequence.

The `slot` number identifies the run sequence time slot in case multiple sequences are available. Slots start counting from 1.

3.1.6 NUOPC_DriverAddRunElement - Add RunElement that links to another slot

INTERFACE:

```
! Private name; call using NUOPC_DriverAddRunElement()
subroutine NUOPC_DriverAddRunElementL(driver, slot, linkSlot, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                       :: driver
  integer,             intent(in)           :: slot
  integer,             intent(in)           :: linkSlot
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Add an element to the run sequence of the Driver that links to the time slot indicated by `linkSlot`.

3.1.7 NUOPC_DriverEgestRunSequence - Egest the run sequence as FreeFormat

INTERFACE:

```
subroutine NUOPC_DriverEgestRunSequence(driver, freeFormat, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                          :: driver
  type(NUOPC_FreeFormat), intent(out)          :: freeFormat
  integer,                intent(out), optional :: rc
```

DESCRIPTION:

Egest the run sequence stored in the driver as a FreeFormat object. It is the caller's responsibility to destroy the created freeFormat object.

3.1.8 NUOPC_DriverGetComp - Get a GridComp child from a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverGetComp()
subroutine NUOPC_DriverGetGridComp(driver, compLabel, comp, petList, &
  relaxedflag, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                       :: driver
  character(len=*),    intent(in)           :: compLabel
  type(ESMF_GridComp), intent(out), optional :: comp
  integer,             pointer,     optional :: petList(:)
  logical,             intent(in),  optional :: relaxedflag
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Query the Driver for a GridComp (i.e. Model, Mediator, or Driver) child component that was added under `compLabel`.

If provided, the `petList` argument will be associated with the petList that was used to create the referenced component.

By default an error is returned if no component is associated with the specified `compLabel`. This error can be suppressed by setting `relaxedflag=.true.`, and unassociated arguments will be returned.

3.1.9 NUOPC_DriverIngestRunSequence - Ingest the run sequence from FreeFormat

INTERFACE:

```
subroutine NUOPC_DriverIngestRunSequence(driver, freeFormat, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                          :: driver
  type(NUOPC_FreeFormat), intent(in)           :: freeFormat
  integer,                intent(out), optional :: rc
```

DESCRIPTION:

Ingest the run sequence from a FreeFormat object.

3.1.10 NUOPC_DriverGetComp - Get a CplComp child from a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverGetComp()
subroutine NUOPC_DriverGetCplComp(driver, srcCompLabel, dstCompLabel, &
  comp, petList, relaxedflag, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                          :: driver
  character(len=*),    intent(in)              :: srcCompLabel
  character(len=*),    intent(in)              :: dstCompLabel
  type(ESMF_CplComp),  intent(out), optional   :: comp
  integer,             pointer    , optional   :: petList(:)
  logical,             intent(in),  optional   :: relaxedflag
  integer,             intent(out), optional   :: rc
```

DESCRIPTION:

Query the Driver for a CplComp (i.e. Connector) child component that was added under `compLabel`.

If provided, the `petList` argument will be associated with the petList that was used to create the referenced component.

By default an error is returned if no component is associated with the specified `compLabel`. This error can be suppressed by setting `relaxedflag=.true.`, and unassociated arguments will be returned.

3.1.11 NUOPC_DriverGetComp - Get all the GridComp child components from a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverGetComp()
subroutine NUOPC_DriverGetAllGridComp(driver, compList, petLists, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                         :: driver
type(ESMF_GridComp), pointer, optional      :: compList(:)
type(type_petList),  pointer, optional      :: petLists(:)
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Get all the GridComp (i.e. Model, Mediator, or Driver) child components from a Driver. The incoming `compList` and `petLists` arguments must be unassociated. On return it becomes the responsibility of the caller to deallocate the associated `compList` and `petLists` arguments

---

3.1.12 NUOPC_DriverGetComp - Get all the CplComp child components from a Driver

INTERFACE:

```
! Private name; call using NUOPC_DriverGetComp()
subroutine NUOPC_DriverGetAllCplComp(driver, compList, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                         :: driver
type(ESMF_CplComp),  pointer                :: compList(:)
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Get all the CplComp (i.e. Connector) child components from a Driver. The incoming `compList` and `petLists` arguments must be unassociated. On return it becomes the responsibility of the caller to deallocate the associated `compList` and `petLists` arguments

---

3.1.13 NUOPC_DriverNewRunSequence - Replace the run sequence in a Driver

INTERFACE:

```
subroutine NUOPC_DriverNewRunSequence(driver, slotCount, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                       :: driver
integer,             intent(in)           :: slotCount
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Replace the current run sequence of the Driver with a new one that has `slotCount` slots. Each slot uses its own clock for time keeping.

3.1.14 NUOPC_DriverPrint - Print internal Driver information

INTERFACE:

```
subroutine NUOPC_DriverPrint(driver, orderflag, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                       :: driver
logical,             intent(in),  optional :: orderflag
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Print internal Driver information. If `orderflag` is provided and set to `.true.`, the output is ordered from lowest to highest PET. Setting this flag makes the method collective.

3.1.15 NUOPC_DriverSetRunSequence - Set internals of RunSequence slot

INTERFACE:

```
! Private name; call using NUOPC_DriverSetRunSequence()
subroutine NUOPC_DriverSetRunSequence(driver, slot, clock, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                        :: driver
  integer,             intent(in)            :: slot
  type(ESMF_Clock),    intent(in)            :: clock
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set the `clock` in the run sequence under `slot` of the Driver.

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
```

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 2.4.2 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.
    - 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`.

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
```

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 2.4.2 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)
  - 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: (NUOPC PROVIDED)
  - *Optional specialization* to initialize export Fields: `label_DataInitialize`
  - Time stamp Fields in export State for compatibility checking.
- IPDv02p5: (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.
    * Advance the current time of the internal Clock according to the time step of the internal Clock.
  - 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`.

3.3.1 NUOPC_ModelGet - Get info from a Model

INTERFACE:

```
subroutine NUOPC_ModelGet(model, driverClock, modelClock, &
  importState, exportState, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                        :: model
type(ESMF_Clock),    intent(out), optional :: driverClock
type(ESMF_Clock),    intent(out), optional :: modelClock
type(ESMF_State),    intent(out), optional :: importState
type(ESMF_State),    intent(out), optional :: exportState
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Access Model information.

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
```

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 2.4.2 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.
    - * 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.

3.4.1 NUOPC_MediatorGet - Get info from a Mediator

INTERFACE:

```
subroutine NUOPC_MediatorGet(mediator, driverClock, mediatorClock, &
  importState, exportState, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                         :: mediator
type(ESMF_Clock),    intent(out), optional :: driverClock
type(ESMF_Clock),    intent(out), optional :: mediatorClock
type(ESMF_State),    intent(out), optional :: importState
type(ESMF_State),    intent(out), optional :: exportState
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Access Mediator information.

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 2.4.3, 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
```

INITIALIZE:

- phase 0: (REQUIRED, NUOPC PROVIDED)
  - Initialize the `InitializePhaseMap` Attribute according to the NUOPC Initialize Phase Definition (IPD) version 04 (see section 2.4.2 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.

3.5.1 NUOPC_ConnectorGet - Get parameters from a Connector

INTERFACE:

```
subroutine NUOPC_ConnectorGet(connector, srcFields, dstFields, rh, state, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                          :: connector
type(ESMF_FieldBundle), intent(out), optional :: srcFields
type(ESMF_FieldBundle), intent(out), optional :: dstFields
type(ESMF_RouteHandle), intent(out), optional :: rh
type(ESMF_State),       intent(out), optional :: state
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Get parameters from the `connector` internal state.

3.5.2 NUOPC_ConnectorSet - Set parameters in a Connector

INTERFACE:

```
subroutine NUOPC_ConnectorSet(connector, srcFields, dstFields, rh, state, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                          :: connector
type(ESMF_FieldBundle), intent(in),  optional :: srcFields
type(ESMF_FieldBundle), intent(in),  optional :: dstFields
type(ESMF_RouteHandle), intent(in),  optional :: rh
type(ESMF_State),       intent(in),  optional :: state
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Set parameters in the `connector` internal state.

3.6 General Generic Component Methods

3.6.1 NUOPC_CompAreServicesSet - Check if SetServices was called

INTERFACE:

```
! Private name; call using NUOPC_CompAreServicesSet()
function NUOPC_GridCompAreServicesSet(comp, rc)
```

RETURN VALUE:

```
logical :: NUOPC_GridCompAreServicesSet
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)           :: comp
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if SetServices has been called for `comp`. Otherwise return `.false.`.

3.6.2 NUOPC_CompAreServicesSet - Check if SetServices was called

INTERFACE:

```
! Private name; call using NUOPC_CompAreServicesSet()
function NUOPC_CplCompAreServicesSet(comp, rc)
```

RETURN VALUE:

```
logical :: NUOPC_CplCompAreServicesSet
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(in)           :: comp
integer,            intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if SetServices has been called for `comp`. Otherwise return `.false.`.

3.6.3 NUOPC_CompAttributeAdd - Add NUOPC GridComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeAdd()
subroutine NUOPC_GridCompAttributeAdd(comp, attrList, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                       :: comp
character(len=*),   intent(in)            :: attrList(:)
integer,            intent(out), optional :: rc
```

DESCRIPTION:

Add Attributes to the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

3.6.4 NUOPC_CompAttributeAdd - Add NUOPC CplComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeAdd()
subroutine NUOPC_CplCompAttributeAdd(comp, attrList, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                        :: comp
character(len=*),   intent(in)            :: attrList(:)
integer,            intent(out), optional :: rc
```

DESCRIPTION:

Add Attributes to the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

3.6.5 NUOPC_CompAttributeEgest - Egest NUOPC GridComp Attributes in FreeFormat

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeEgest()
subroutine NUOPC_GridCompAttributeEge(comp, freeFormat, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp),    intent(in)            :: comp
type(NUOPC_FreeFormat), intent(out)           :: freeFormat
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Egest the Attributes of the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance") as a FreeFormat object. It is the caller's responsibility to destroy the created `freeFormat` object.

3.6.6 NUOPC_CompAttributeEgest - Egest NUOPC CplComp Attributes in FreeFormat

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeEgest()
subroutine NUOPC_CplCompAttributeEge(comp, freeFormat, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp),     intent(in)            :: comp
type(NUOPC_FreeFormat), intent(out)           :: freeFormat
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Egest the Attributes of the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance") as a FreeFormat object. It is the caller's responsibility to destroy the created `freeFormat` object.

3.6.7 NUOPC_CompAttributeGet - Get a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_GridCompAttributeGet(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)            :: comp
character(*),        intent(in)            :: name
character(*),        intent(out)           :: value
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Access the Attribute `name` inside of `comp` using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.8 NUOPC_CompAttributeGet - Get a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_CplCompAttributeGet(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp),  intent(in)           :: comp
character(*),        intent(in)           :: name
character(*),        intent(out)          :: value
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Access the Attribute `name` inside of `comp` using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.9 NUOPC_CompAttributeGet - Get a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_GridCompAttributeGetI(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)           :: comp
character(*),        intent(in)           :: name
integer,             intent(out)          :: value
integer,             intent(out), optional :: rc
```

DESCRIPTION:

Access the Attribute `name` inside of `comp` using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.10 NUOPC_CompAttributeGet - Get a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_CplCompAttributeGetI(comp, name, value, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),  intent(in)           :: comp
  character(*),        intent(in)           :: name
  integer,             intent(out)          :: value
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Access the Attribute `name` inside of `comp` using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.11 NUOPC_CompAttributeGet - Get a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_GridCompAttributeGetSL(comp, name, valueList, itemCount, &
  rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(in)           :: comp
  character(*),        intent(in)           :: name
  character(*),        intent(out), optional :: valueList(:)
  integer,             intent(out), optional :: itemCount
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Access the Attribute `name` inside of `comp` using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.12 NUOPC_CompAttributeGet - Get a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_CplCompAttributeGetSL(comp, name, valueList, itemCount, &
  rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp), intent(in)            :: comp
  character(*),       intent(in)            :: name
  character(*),       intent(out), optional :: valueList(:)
  integer,            intent(out), optional :: itemCount
  integer,            intent(out), optional :: rc
```

DESCRIPTION:

Access the Attribute `name` inside of `comp` using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.13 NUOPC_CompAttributeGet - Get a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_GridCompAttributeGetTK(comp, name, typekind, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),      intent(in)            :: comp
  character(*),             intent(in)            :: name
  type(ESMF_TypeKind_Flag), intent(out)           :: typekind
  integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Query the `typekind` of the Attribute `name` inside of `comp` using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.14 NUOPC_CompAttributeGet - Get a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeGet()
subroutine NUOPC_CplCompAttributeGetTK(comp, name, typekind, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp),       intent(in)            :: comp
character(*),             intent(in)            :: name
type(ESMF_TypeKind_Flag), intent(out)           :: typekind
integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Query the `typekind` of the Attribute `name` inside of `comp` using the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.15 NUOPC_CompAttributeIngest - Ingest free format NUOPC GridComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeIngest()
subroutine NUOPC_GridCompAttributeIng(comp, freeFormat, addFlag, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp),    intent(in)            :: comp
type(NUOPC_FreeFormat), intent(in)            :: freeFormat
logical,                intent(in),  optional :: addFlag
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Ingest the Attributes from a FreeFormat object onto the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

If `addFlag` is `.false.` (default), an error will be returned if an attribute is to be ingested that was not previously added to the `comp` object. If `addFlag` is `.true.`, all missing attributes will be added by this method automatically as needed.

3.6.16 NUOPC_CompAttributeIngest - Ingest free format NUOPC CplComp Attributes

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeIngest()
subroutine NUOPC_CplCompAttributeIng(comp, freeFormat, addFlag, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp),     intent(in)            :: comp
type(NUOPC_FreeFormat), intent(in)            :: freeFormat
logical,                intent(in),  optional :: addFlag
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Ingest the Attributes from a FreeFormat object onto the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

If `addFlag` is `.false.` (default), an error will be returned if an attribute is to be ingested that was not previously added to the `comp` object. If `addFlag` is `.true.`, all missing attributes will be added by this method automatically as needed.

3.6.17 NUOPC_CompAttributeSet - Set a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_GridCompAttributeSetS(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                   :: comp
character(*), intent(in)              :: name
character(*), intent(in)              :: value
integer,      intent(out), optional   :: rc
```

DESCRIPTION:

Set the Attribute `name` inside of `comp` on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.18 NUOPC_CompAttributeSet - Set a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_CplCompAttributeSetS(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                    :: comp
character(*), intent(in)              :: name
character(*), intent(in)              :: value
integer,      intent(out), optional   :: rc
```

DESCRIPTION:

Set the Attribute `name` inside of `comp` on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.19 NUOPC_CompAttributeSet - Set a NUOPC GridComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_GridCompAttributeSetI(comp, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                   :: comp
character(*), intent(in)              :: name
integer,      intent(in)              :: value
integer,      intent(out), optional   :: rc
```

DESCRIPTION:

Set the Attribute `name` inside of `comp` on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.20 NUOPC_CompAttributeSet - Set a NUOPC CplComp Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_CplCompAttributeSetI(comp, name, value, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp)                    :: comp
  character(*), intent(in)              :: name
  integer,      intent(in)              :: value
  integer,      intent(out), optional   :: rc
```

DESCRIPTION:

Set the Attribute `name` inside of `comp` on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.21 NUOPC_CompAttributeSet - Set a NUOPC GridComp List Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_GridCompAttributeSetSL(comp, name, valueList, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                   :: comp
  character(*), intent(in)              :: name
  character(*), intent(in)              :: valueList(:)
  integer,      intent(out), optional   :: rc
```

DESCRIPTION:

Set the Attribute `name` inside of `comp` on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.22 NUOPC_CompAttributeSet - Set a NUOPC CplComp List Attribute

INTERFACE:

```
! Private name; call using NUOPC_CompAttributeSet()
subroutine NUOPC_CplCompAttributeSetSL(comp, name, valueList, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                    :: comp
character(*), intent(in)              :: name
character(*), intent(in)              :: valueList(:)
integer,      intent(out), optional   :: rc
```

DESCRIPTION:

Set the Attribute `name` inside of `comp` on the highest level of the standard NUOPC AttPack hierarchy (convention="NUOPC", purpose="Instance").

Return with error if the Attribute is not present or not set.

3.6.23 NUOPC_CompCheckSetClock - Check Clock compatibility and set stopTime

INTERFACE:

```
! Private name; call using NUOPC_CompCheckSetClock()
subroutine NUOPC_GridCompCheckSetClock(comp, externalClock, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp),     intent(inout)         :: comp
type(ESMF_Clock),        intent(in)            :: externalClock
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Compare `externalClock` to the internal clock of `comp` to make sure they match in their current time. Also ensure that the time step of the external clock is a multiple of the time step of the internal clock. If both conditions are satisfied then set the stop time of the internal clock so it is reached in one time step of the external clock. Otherwise leave the internal clock unchanged and return with error. The direction of the involved clocks is taking into account.

3.6.24 NUOPC_CompDerive - Derive a GridComp from a generic component

INTERFACE:

```
! Private name; call using NUOPC_CompDerive()
subroutine NUOPC_GridCompDerive(comp, genericSetServicesRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)           :: comp
interface
  subroutine genericSetServicesRoutine(gridcomp, rc)
    use ESMF
    implicit none
    type(ESMF_GridComp)       :: gridcomp ! must not be optional
    integer, intent(out)      :: rc       ! must not be optional
  end subroutine
end interface
integer,            intent(out), optional :: rc
```

DESCRIPTION:

Derive a GridComp (i.e. Model, Mediator, or Driver) from a generic component by calling into the specified `SetServices()` routine of the generic component. This is typically the first call in the `SetServices()` routine of the specializing component, and is followed by `NUOPC_CompSpecialize()` calls.

3.6.25 NUOPC_CompDerive - Derive a CplComp from a generic component

INTERFACE:

```
! Private name; call using NUOPC_CompDerive()
subroutine NUOPC_CplCompDerive(comp, genericSetServicesRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp),  intent(in)           :: comp
interface
  subroutine genericSetServicesRoutine(cplcomp, rc)
    use ESMF
    implicit none
    type(ESMF_CplComp)        :: cplcomp  ! must not be optional
    integer, intent(out)      :: rc       ! must not be optional
  end subroutine
end interface
integer,            intent(out), optional :: rc
```

DESCRIPTION:

Derive a CplComp (i.e. Connector) from a generic component by calling into the specified `SetServices()` routine of the generic component. This is typically the first call in the `SetServices()` routine of the specializing component, and is followed by `NUOPC_CompSpecialize()` calls.

3.6.26 NUOPC_CompFilterPhaseMap - Filter the Phase Map of a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompFilterPhaseMap()
subroutine NUOPC_GridCompFilterPhaseMap(comp, methodflag, acceptStringList, &
  rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                           :: comp
  type(ESMF_Method_Flag), intent(in)            :: methodflag
  character(len=*),       intent(in)            :: acceptStringList(:)
  integer,                intent(out), optional :: rc
```

DESCRIPTION:

Filter all PhaseMap entries in a GridComp (i.e. Model, Mediator, or Driver) that do *not* match any entry in the `acceptStringList`.

3.6.27 NUOPC_CompFilterPhaseMap - Filter the Phase Map of a CplComp

INTERFACE:

```
! Private name; call using NUOPC_CompFilterPhaseMap()
subroutine NUOPC_CplCompFilterPhaseMap(comp, methodflag, acceptStringList, &
  rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp)                            :: comp
  type(ESMF_Method_Flag), intent(in)            :: methodflag
  character(len=*),       intent(in)            :: acceptStringList(:)
  integer,                intent(out), optional :: rc
```

DESCRIPTION:

Filter all PhaseMap entries in a CplComp (i.e. Connector) that do *not* match any entry in the `acceptStringList`.

3.6.28 NUOPC_CompSearchPhaseMap - Search the Phase Map of a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompSearchPhaseMap()
subroutine NUOPC_GridCompSearchPhaseMap(comp, methodflag, phaseLabel, &
  phaseIndex, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                            :: comp
type(ESMF_Method_Flag), intent(in)             :: methodflag
character(len=*),       intent(in),  optional :: phaseLabel
integer,                intent(out)            :: phaseIndex
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Search all PhaseMap entries in a GridComp (i.e. Model, Mediator, or Driver) to see if `phaseLabel` is found. Return the associated ESMF `phaseIndex`, or `-1` if not found. If `phaseLabel` is not specified, set `phaseIndex` to the first entry in the PhaseMap, or `-1` if there are no entries.

3.6.29 NUOPC_CompSearchPhaseMap - Search the Phase Map of a CplComp

INTERFACE:

```
! Private name; call using NUOPC_CompSearchPhaseMap()
subroutine NUOPC_CplCompSearchPhaseMap(comp, methodflag, phaseLabel, &
  phaseIndex, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                             :: comp
type(ESMF_Method_Flag), intent(in)             :: methodflag
character(len=*),       intent(in),  optional :: phaseLabel
integer,                intent(out)            :: phaseIndex
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Search all PhaseMap entries in a CplComp (i.e. Connector) to see if `phaseLabel` is found. Return the associated ESMF `phaseIndex`, or `-1` if not found. If `phaseLabel` is not specified, set `phaseIndex` to the first entry in the PhaseMap, or `-1` if there are no entries.

3.6.30 NUOPC_CompSetClock - Initialize and set the internal Clock of a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompSetClock()
subroutine NUOPC_GridCompSetClock(comp, externalClock, stabilityTimeStep, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp),     intent(inout)         :: comp
type(ESMF_Clock),        intent(in)            :: externalClock
type(ESMF_TimeInterval), intent(in),  optional :: stabilityTimeStep
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Set the component internal clock as a copy of `externalClock`, but with a timeStep that is less than or equal to the stabilityTimeStep. At the same time ensure 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.

3.6.31 NUOPC_CompSetEntryPoint - Set entry point for a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompSetEntryPoint()
subroutine NUOPC_GridCompSetEntryPoint(comp, methodflag, phaseLabelList, &
  userRoutine, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp)                    :: comp
  type(ESMF_Method_Flag), intent(in)     :: methodflag
  character(len=*),       intent(in)     :: phaseLabelList(:)
  interface
    subroutine userRoutine(gridcomp, importState, exportState, clock, rc)
      use ESMF_CompMod
      use ESMF_StateMod
      use ESMF_ClockMod
      implicit none
      type(ESMF_GridComp)         :: gridcomp     ! must not be optional
      type(ESMF_State)            :: importState  ! must not be optional
      type(ESMF_State)            :: exportState  ! must not be optional
      type(ESMF_Clock)            :: clock        ! must not be optional
      integer, intent(out)        :: rc           ! must not be optional
    end subroutine
  end interface
  integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set an entry point for a GridComp (i.e. Model, Mediator, or Driver). Publish the new entry point in the correct `PhaseMap` component attribute.

3.6.32 NUOPC_CompSetEntryPoint - Set entry point for a CplComp

INTERFACE:

```
! Private name; call using NUOPC_CompSetEntryPoint()
subroutine NUOPC_CplCompSetEntryPoint(comp, methodflag, phaseLabelList, &
  userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: comp
type(ESMF_Method_Flag), intent(in)      :: methodflag
character(len=*),       intent(in)      :: phaseLabelList(:)
interface
  subroutine userRoutine(cplcomp, importState, exportState, clock, rc)
    use ESMF_CompMod
    use ESMF_StateMod
    use ESMF_ClockMod
    implicit none
    type(ESMF_CplComp)          :: cplcomp      ! must not be optional
    type(ESMF_State)            :: importState  ! must not be optional
    type(ESMF_State)            :: exportState  ! must not be optional
    type(ESMF_Clock)            :: clock        ! must not be optional
    integer, intent(out)        :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set an entry point for a CplComp (i.e. Connector). Publish the new entry point in the correct `PhaseMap` component attribute.

3.6.33 NUOPC_CompSetInternalEntryPoint - Set internal entry point for a GridComp

INTERFACE:

```
! Private name; call using NUOPC_CompSetInternalEntryPoint()
subroutine NUOPC_GridCompSetIntEntryPoint(comp, methodflag, phaseLabelList, &
  userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: comp
type(ESMF_Method_Flag), intent(in)      :: methodflag
character(len=*),       intent(in)      :: phaseLabelList(:)
interface
  subroutine userRoutine(gridcomp, importState, exportState, clock, rc)
    use ESMF_CompMod
    use ESMF_StateMod
    use ESMF_ClockMod
    implicit none
    type(ESMF_GridComp)         :: gridcomp     ! must not be optional
    type(ESMF_State)            :: importState  ! must not be optional
    type(ESMF_State)            :: exportState  ! must not be optional
    type(ESMF_Clock)            :: clock        ! must not be optional
    integer, intent(out)        :: rc           ! must not be optional
  end subroutine
```

```
    end interface
    integer,          intent(out), optional :: rc
```

DESCRIPTION:

Set an *internal* entry point for a GridComp (i.e. Driver). Only Drivers currently utilize internal entry points. Internal entry points allow user specialization on the driver level during initialization and run sequencing.

3.6.34 NUOPC_CompSetServices - Try to find and call SetServices in a shared object

INTERFACE:

```
  ! Private name; call using NUOPC_CompSetServices()
  recursive subroutine NUOPC_GridCompSetServices(comp, sharedObj, userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),     intent(inout)         :: comp
    character(len=*),        intent(in),  optional :: sharedObj
    integer,                 intent(out), optional :: userRc
    integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Try to find a routine called "`SetServices`" in the `sharedObj` file and execute the routine. An attempt is made to find a routine that is close in name to "`SetServices`", allowing for compiler name mangling, i.e. upper and lower case, as well as trailing underscores.

3.6.35 NUOPC_CompSpecialize - Specialize a derived GridComp

INTERFACE:

```
  ! Private name; call using NUOPC_CompSpecialize()
  subroutine NUOPC_GridCompSpecialize(comp, specLabel, specPhaseLabel, &
    specRoutine, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp)                     :: comp
    character(len=*), intent(in)            :: specLabel
    character(len=*), intent(in),  optional :: specPhaseLabel
    interface
      subroutine specRoutine(gridcomp, rc)
        use ESMF
```

```
      implicit none
      type(ESMF_GridComp)        :: gridcomp ! must not be optional
      integer, intent(out)       :: rc       ! must not be optional
    end subroutine
  end interface
  integer,          intent(out), optional :: rc
```

DESCRIPTION:

Specialize a derived GridComp (i.e. Model, Mediator, or Driver). If `specPhaseLabel` is specified, the specialization only applies to the associated phase. Otherwise the specialization applies to all phases.

3.6.36 NUOPC_CompSpecialize - Specialize a derived CplComp

INTERFACE:

```
! Private name; call using NUOPC_CompSpecialize()
subroutine NUOPC_CplCompSpecialize(comp, specLabel, specPhaseLabel, &
  specRoutine, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp)                     :: comp
  character(len=*), intent(in)           :: specLabel
  character(len=*), intent(in),  optional :: specPhaseLabel
  interface
    subroutine specRoutine(cplcomp, rc)
      use ESMF
      implicit none
      type(ESMF_CplComp)         :: cplcomp  ! must not be optional
      integer, intent(out)       :: rc       ! must not be optional
    end subroutine
  end interface
  integer,          intent(out), optional :: rc
```

DESCRIPTION:

Specialize a derived CplComp (i.e. Connector). If `specPhaseLabel` is specified, the specialization only applies to the associated phase. Otherwise the specialization applies to all phases.

3.7 Field Dictionary Methods

3.7.1 NUOPC_FieldDictionaryAddEntry - Add an entry to the NUOPC Field dictionary

INTERFACE:

```
subroutine NUOPC_FieldDictionaryAddEntry(standardName, canonicalUnits, rc)
```

ARGUMENTS:

```
     character(*),                intent(in)           :: standardName
     character(*),                intent(in)           :: canonicalUnits
     integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Add an entry to the NUOPC Field dictionary. If necessary the dictionary is first set up.

3.7.2 NUOPC_FieldDictionaryEgest - Egest NUOPC Field dictionary into FreeFormat

INTERFACE:

```
   subroutine NUOPC_FieldDictionaryEgest(freeFormat, rc)
```

ARGUMENTS:

```
     type(NUOPC_FreeFormat), intent(out)           :: freeFormat
     integer,                intent(out), optional :: rc
```

DESCRIPTION:

Egest the contents of the NUOPC Field dictionary into a FreeFormat object. It is the caller's responsibility to destroy the created `freeFormat` object.

3.7.3 NUOPC_FieldDictionaryGetEntry - Get information about a NUOPC Field dictionary entry

INTERFACE:

```
   subroutine NUOPC_FieldDictionaryGetEntry(standardName, canonicalUnits, rc)
```

ARGUMENTS:

```
     character(*),                intent(in)           :: standardName
     character(*),                intent(out), optional :: canonicalUnits
     integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Return the canonical units that the NUOPC Field dictionary associates with the `standardName`.

3.7.4 NUOPC_FieldDictionaryHasEntry - Check whether the NUOPC Field dictionary has a specific entry

INTERFACE:

```
function NUOPC_FieldDictionaryHasEntry(standardName, rc)
```

RETURN VALUE:

```
logical :: NUOPC_FieldDictionaryHasEntry
```

ARGUMENTS:

```
character(*),                intent(in)            :: standardName
integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the NUOPC Field dictionary has an entry with the specified `standardName`, `.false.` otherwise.

3.7.5 NUOPC_FieldDictionaryMatchSyno - Check whether the NUOPC Field dictionary considers the standard names synonyms

INTERFACE:

```
function NUOPC_FieldDictionaryMatchSyno(standardName1, standardName2, rc)
```

RETURN VALUE:

```
logical :: NUOPC_FieldDictionaryMatchSyno
```

ARGUMENTS:

```
character(*),                intent(in)            :: standardName1
character(*),                intent(in)            :: standardName2
integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the NUOPC Field dictionary considers `standardName1` and `standardName2` synonyms, `.false.` otherwise. An entry with standard name of `standardName1` must exist in the field dictionary, or else an error will be returned. However, `standardName2` need not correspond to an existing entry. If `standardName2` does not correspond to an existing entry in the field dictionary, the value of .false. will be returned.

3.7.6 NUOPC_FieldDictionarySetSyno - Set synonyms in the NUOPC Field dictionary

INTERFACE:

```
subroutine NUOPC_FieldDictionarySetSyno(standardNames, rc)
```

ARGUMENTS:

```
character(*),                intent(in)            :: standardNames(:)
integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Set all of the elements of the `standardNames` argument to be considered synonyms by the field dictionary. Every element in `standardNames` must correspond to the standard name of already existing entries in the field dictionary, or else an error will be returned.

3.7.7 NUOPC_FieldDictionarySetup - Setup the NUOPC Field dictionary

INTERFACE:

```
subroutine NUOPC_FieldDictionarySetup(rc)
```

ARGUMENTS:

```
integer,      intent(out), optional   :: rc
```

DESCRIPTION:

Setup the NUOPC Field dictionary.

3.8 Free Format Methods

3.8.1 NUOPC_FreeFormatAdd - Add lines to a FreeFormat object

INTERFACE:

```
subroutine NUOPC_FreeFormatAdd(freeFormat, stringList, rc)
```

ARGUMENTS:

```
type(NUOPC_FreeFormat),           intent(inout) :: freeFormat
character(len=*),                 intent(in)    :: stringList(:)
integer,                optional, intent(out)   :: rc
```

DESCRIPTION:

Add lines to a FreeFormat object. The capacity of `freeFormat` is increased by at list the size of `stringList`, but potentially more. The elements in `stringList` are added to the end of `freeFormat`.

3.8.2 NUOPC_FreeFormatCreate - Create a FreeFormat object

INTERFACE:

```
! call using generic interface: NUOPC_FreeFormatCreate
function NUOPC_FreeFormatCreateDefault(stringList, capacity, rc)
```

RETURN VALUE:

```
type(NUOPC_FreeFormat) :: NUOPC_FreeFormatCreateDefault
```

ARGUMENTS:

```
character(len=*), optional, intent(in)  :: stringList(:)
integer,          optional, intent(in)  :: capacity
integer,          optional, intent(out) :: rc
```

DESCRIPTION:

Create a new FreeFormat object. If `stringList` is provided, then the newly created object will hold the provided strings and the count is that of `size(stringList)`. If `capacity` is provided, it is used to set the capacity of the newly created FreeFormat object. Providing a `capacity` that is smaller than `size(stringList)` triggers an error.

3.8.3 NUOPC_FreeFormatCreate - Create a FreeFormat object from Config

INTERFACE:

```
! call using generic interface: NUOPC_FreeFormatCreate
function NUOPC_FreeFormatCreateRead(config, label, relaxedflag, rc)
```

RETURN VALUE:

```
type(NUOPC_FreeFormat) :: NUOPC_FreeFormatCreateRead
```

ARGUMENTS:

```
type(ESMF_Config)                          :: config
character(len=*),      intent(in)          :: label
logical,               intent(in),  optional :: relaxedflag
integer,               intent(out), optional :: rc
```

DESCRIPTION:

Create a new FreeFormat object from ESMF_Config object. The `config` object must exist, and `label` must reference a table attribute within `config`.

By default an error is returned if `label` is not found in `config`. This error can be suppressed by setting `relaxedflag=.true.`, and an empty FreeFormat object will be returned.

3.8.4 NUOPC_FreeFormatDestroy - Destroy a FreeFormat object

INTERFACE:

```
subroutine NUOPC_FreeFormatDestroy(freeFormat, rc)
```

ARGUMENTS:

```
type(NUOPC_FreeFormat),           intent(inout) :: freeFormat
integer,                optional, intent(out)   :: rc
```

DESCRIPTION:

Destroy a FreeFormat object. All internal memory is deallocated.

3.8.5 NUOPC_FreeFormatGet - Get information from a FreeFormat object

INTERFACE:

```
subroutine NUOPC_FreeFormatGet(freeFormat, lineCount, capacity, stringList, rc)
```

ARGUMENTS:

```
type(NUOPC_FreeFormat),                      intent(in)  :: freeFormat
integer,                           optional, intent(out) :: lineCount
integer,                           optional, intent(out) :: capacity
character(len=NUOPC_FreeFormatLen), optional, pointer    :: stringList(:)
integer,                           optional, intent(out) :: rc
```

DESCRIPTION:

Get information from a FreeFormat object.

3.8.6 NUOPC_FreeFormatGetLine - Get line info from a FreeFormat object

INTERFACE:

```
   subroutine NUOPC_FreeFormatGetLine(freeFormat, line, lineString, tokenCount, &
     tokenList, rc)
```

ARGUMENTS:

```
     type(NUOPC_FreeFormat),                       intent(in)  :: freeFormat
     integer,                                      intent(in)  :: line
     character(len=NUOPC_FreeFormatLen), optional, intent(out) :: lineString
     integer,                            optional, intent(out) :: tokenCount
     character(len=NUOPC_FreeFormatLen), optional, intent(out) :: tokenList(:)
     integer,                            optional, intent(out) :: rc
```

DESCRIPTION:

Get information about a specific line in a FreeFormat object.

3.8.7 NUOPC_FreeFormatLog - Write a FreeFormat object to the default Log

INTERFACE:

```
   subroutine NUOPC_FreeFormatLog(freeFormat, rc)
```

ARGUMENTS:

```
     type(NUOPC_FreeFormat),           intent(in)    :: freeFormat
     integer,                optional, intent(out)   :: rc
```

DESCRIPTION:

Write a FreeFormat object to the default Log.

3.8.8 NUOPC_FreeFormatPrint - Print a FreeFormat object

INTERFACE:

```
   subroutine NUOPC_FreeFormatPrint(freeFormat, rc)
```

ARGUMENTS:

```
    type(NUOPC_FreeFormat),           intent(in)    :: freeFormat
    integer,                optional, intent(out)   :: rc
```

DESCRIPTION:

Print a FreeFormat object.

3.9 Utility Routines

3.9.1 NUOPC_AddNamespace - Add a namespace to a State

INTERFACE:

```
  subroutine NUOPC_AddNamespace(state, namespace, nestedStateName, &
    nestedState, rc)
```

ARGUMENTS:

```
    type(ESMF_State), intent(inout)         :: state
    character(len=*), intent(in)            :: namespace
    character(len=*), intent(in),  optional :: nestedStateName
    type(ESMF_State), intent(out), optional :: nestedState
    integer,          intent(out), optional :: rc
```

DESCRIPTION:

Add a namespace to `state`. Namespaces are implemented via nested states. This creates a nested state inside of `state`. The nested state is returned as `nestedState`. If provided, `nestedStateName` will be used to name the newly created nested state. The default name of the nested state is equal to `namespace`.

The arguments are:

state The `ESMF_State` object to which the namespace is added.

namespace The namespace string.

[nestedStateName] Name of the nested state. Defaults to `namespace`.

[nestedState] Optional return of the newly created nested state.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.2 NUOPC_Advertise - Advertise a single Field in a State

INTERFACE:

```
  ! call using generic interface: NUOPC_Advertise
  subroutine NUOPC_AdvertiseField(state, StandardName, Units, &
    LongName, ShortName, name, TransferOfferGeomObject, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout)         :: state
character(*),     intent(in)            :: StandardName
character(*),     intent(in),  optional :: Units
character(*),     intent(in),  optional :: LongName
character(*),     intent(in),  optional :: ShortName
character(*),     intent(in),  optional :: name
character(*),     intent(in),  optional :: TransferOfferGeomObject
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Advertise a field in a state. This creates an empty field and adds it to `state`. The "StandardName", "Units", "LongName", "ShortName", and "TransferOfferGeomObject" attributes of the field are set according to the provided input..

The call checks the provided information against the NUOPC Field Dictionary to ensure correctness. Defaults are set according to the NUOPC Field Dictionary.

The arguments are:

state The `ESMF_State` object through which the field is advertised.

StandardName The "StandardName" attribute of the advertised field. Must be a StandardName found in the NUOPC Field Dictionary.

[Units] The "Units" attribute of the advertised field. Must be convertible to the canonical units specified in the NUOPC Field Dictionary for the specified StandardName. (Currently this is restricted to be identical to the canonical untis specified in the NUOPC Field Dictionary.) If omitted, the default is to use the canonical units associated with the StandardName in the NUOPC Field Dictionary.

[LongName] The "LongName" attribute of the advertised field. NUOPC does not restrict the value of this attribute. If omitted, the default is to use the StandardName.

[ShortName] The "ShortName" attribute of the advertised field. NUOPC does not restrict the value of this attribute. If omitted, the default is to use the StandardName.

[name] The actual name of the advertised field by which it is accessed in the state object. NUOPC does not restrict the value of this variable. If omitted, the default is to use the value of the ShortName.

[TransferOfferGeomObject] The "TransferOfferGeomObject" attribute of the advertised field. NUOPC controls the vocabulary of this attribute. Valid options are "will provide", "can provide", "cannot provide". If omitted, the default is "will provide".

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.3 NUOPC_Advertise - Advertise a list of Fields in a State

INTERFACE:

```
! call using generic interface: NUOPC_Advertise
subroutine NUOPC_AdvertiseFields(state, StandardNames, &
  TransferOfferGeomObject, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout)         :: state
character(*),     intent(in)            :: StandardNames(:)
character(*),     intent(in),  optional :: TransferOfferGeomObject
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Advertise a list of fields in a state. This creates a list of empty fields and adds it to the `state`. The "StandardName", and "TransferOfferGeomObject" attributes of all the fields are set according to the provided input. The "Units", "LongName", and "ShortName" attributes for each field are set according to the defaults documented under method 3.9.2

The call checks the provided information against the NUOPC Field Dictionary to ensure correctness.

The arguments are:

state The `ESMF_State` object through which the fields are advertised.

StandardNames A list of "StandardName" attributes of the advertised fields. Must be StandardNames found in the NUOPC Field Dictionary.

[TransferOfferGeomObject] The "TransferOfferGeomObject" attribute associated with the advertised fields. This setting applies to all the fields advertised in this call. NUOPC controls the vocabulary of this attribute. Valid options are "will provide", "can provide", "cannot provide". If omitted, the default is "will provide".

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.4 NUOPC_AdjustClock - Adjust the timestep in a clock

INTERFACE:

```
subroutine NUOPC_AdjustClock(clock, maxTimestep, rc)
```

ARGUMENTS:

```
type(ESMF_Clock)                              :: clock
type(ESMF_TimeInterval), intent(in),  optional :: maxTimestep
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Adjust the `clock` to have a potentially smaller timestep. The timestep on the incoming `clock` object is compared to the `maxTimestep`, and reset to the smaller of the two.

The arguments are:

clock The clock to be adjusted.

[maxTimestep] Upper bound of the timestep allowed in `clock`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.5 NUOPC_CheckSetClock - Check a Clock for compatibility and set its values

INTERFACE:

```
subroutine NUOPC_CheckSetClock(setClock, checkClock, setStartTimeToCurrent, rc)
```

ARGUMENTS:

```
type(ESMF_Clock),        intent(inout)         :: setClock
type(ESMF_Clock),        intent(in)            :: checkClock
logical,                 intent(in),  optional :: setStartTimeToCurrent
integer,                 intent(out), optional :: rc
```

DESCRIPTION:

Compare `setClock` to `checkClock` to ensure they match in their current fime. Further ensure that the timeStep of `checkClock` is a multiple of the timeStep of `setClock`. If both conditions are satisfied then the stopTime of the `setClock` is set one `checkClock` timeStep ahead of the current time. The direction of the clock is considered.

By default the startTime of the `setClock` is not modified. However, if `setStartTimeToCurrent == .true.` the startTime of `setClock` is set to the currentTime of `checkClock`.

The arguments are:

setClock The `ESMF_Clock` object to be checked and set.

checkClock The reference clock object.

[setStartTimeToCurrent] If `.true.` then also set the startTime in `setClock` according to the startTime in `checkClock`. The default is `.false.`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.6 NUOPC_GetAttribute - Get the value of a NUOPC Field Attribute

INTERFACE:

```
! call using generic interface: NUOPC_GetAttribute
subroutine NUOPC_GetAttributeFieldVal(field, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)           :: field
character(*),     intent(in)           :: name
character(*),     intent(out)          :: value
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Access the attribute `name` inside of `field` using the convention `NUOPC` and purpose `Instance`. Returns with error if the attribute is not present or not set.

The arguments are:

field The `ESMF_Field` object to be queried.

name The name of the queried attribute.

value The value of the queried attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.7 NUOPC_GetAttribute - Get the typekind of a NUOPC Field Attribute

INTERFACE:

```
! call using generic interface: NUOPC_GetAttribute
subroutine NUOPC_GetAttributeFieldTK(field, name, typekind, rc)
```

ARGUMENTS:

```
type(ESMF_Field),         intent(in)           :: field
character(*),             intent(in)           :: name
type(ESMF_TypeKind_Flag), intent(out)          :: typekind
integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Query the `typekind` of the attribute `name` inside of `field` using the convention `NUOPC` and purpose `Instance`. Returns with error if the attribute is not present or not set.

The arguments are:

field The `ESMF_Field` object to be queried.

name The name of the queried attribute.

typekind The typekind of the queried attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.8 NUOPC_GetAttribute - Get the value of a NUOPC State Attribute

INTERFACE:

```
! call using generic interface: NUOPC_GetAttribute
subroutine NUOPC_GetAttributeState(state, name, value, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in)           :: state
character(*),     intent(in)           :: name
character(*),     intent(out)          :: value
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Access the attribute `name` inside of `state` using the convention `NUOPC` and purpose `Instance`. Returns with error if the attribute is not present or not set.

The arguments are:

state The `ESMF_State` object to be queried.

name The name of the queried attribute.

value The value of the queried attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.9 NUOPC_GetStateMemberLists - Build lists of information of State members

INTERFACE:

```
recursive subroutine NUOPC_GetStateMemberLists(state, StandardNameList, &
  ConnectedList, NamespaceList, itemNameList, fieldList, rc)
```

ARGUMENTS:

```
type(ESMF_State),       intent(in)           :: state
character(ESMF_MAXSTR), pointer, optional    :: StandardNameList(:)
character(ESMF_MAXSTR), pointer, optional    :: ConnectedList(:)
character(ESMF_MAXSTR), pointer, optional    :: NamespaceList(:)
character(ESMF_MAXSTR), pointer, optional    :: itemNameList(:)
type(ESMF_Field),       pointer, optional    :: fieldList(:)
integer,                intent(out), optional :: rc
```

DESCRIPTION:

Construct lists containing the StandardNames, field names, and connected status of the fields in `state`. Return this information in the list arguments. Recursively parse through nested States.

All pointer arguments present must enter this method unassociated. On return, the deallocation of an associated pointer becomes the responsibility of the caller.

The arguments are:

state The `ESMF_State` object to be queried.

[StandardNameList] If present, return a list of the "StandardName" attribute of each member.

[ConnectedList] If present, return a list of the "Connected" attribute of each member.

[NamespaceList] If present, return a list of the namespace of each member.

[itemNameList] If present, return a list of each member name.

[fieldList] If present, return a list of the member fields.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.10 NUOPC_IsAtTime - Check if a Field is at the given Time

INTERFACE:

```
! call using generic interface: NUOPC_IsAtTime
function NUOPC_IsAtTimeField(field, time, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsAtTimeField
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)            :: field
type(ESMF_Time),  intent(in)            :: time
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Returns `.true.` if `field` has a timestamp attribute that matches `time`. Otherwise returns `.false.`.

The arguments are:

field The `ESMF_Field` object to be checked.

time The time to compare against.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.11 NUOPC_IsAtTime - Check if Field(s) in a State are at the given Time

INTERFACE:

```
! call using generic interface: NUOPC_IsAtTime
function NUOPC_IsAtTimeState(state, time, fieldName, count, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsAtTimeState
```

ARGUMENTS:

```
type(ESMF_State), intent(in)            :: state
type(ESMF_Time),  intent(in)            :: time
character(*),     intent(in),  optional :: fieldName
integer,          intent(out), optional :: count
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the field(s) in `state` have a timestamp attribute that matches `time`. Otherwise return `.false.`.

The arguments are:

state The `ESMF_State` object to be checked.

time The time to compare against.

[fieldName] The name of the field in `state` to be checked. If provided, and the state does not contain a field with `fieldName`, return an error in `rc`. If not provided, check *all* the fields contained in `state` and return `.true.` if all the fields are at the correct time.

[count] If provided, the number of fields that are at time are returned. If `fieldName` is present then `count` cannot be greater than 1.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.12 NUOPC_IsConnected - Check if a Field is connected

INTERFACE:

```
! call using generic interface: NUOPC_IsConnected
function NUOPC_IsConnectedField(field, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsConnectedField
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)            :: field
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the `field` is connected. Otherwise return `.false.`.

The arguments are:

field The `ESMF_Field` object to be checked.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.13 NUOPC_IsConnected - Check if Field(s) in a State are connected

INTERFACE:

```
! call using generic interface: NUOPC_IsConnected
function NUOPC_IsConnectedState(state, fieldName, count, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsConnectedState
```

ARGUMENTS:

```
type(ESMF_State), intent(in)            :: state
character(*),     intent(in),  optional :: fieldName
integer,          intent(out), optional :: count
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the field(s) in `state` are connected. Otherwise return `.false.`.

The arguments are:

state The `ESMF_State` object to be checked.

[fieldName] The name of the field in `state` to be checked. If provided, and the state does not contain a field with `fieldName`, return an error in `rc`. If not provided, check *all* the fields contained in `state` and return `.true.` if all the fields are connected.

[count] If provided, the number of fields that are connected are returned. If `fieldName` is present then `count` cannot be greater than 1.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.14 NUOPC_IsUpdated - Check if a Field is marked as updated

INTERFACE:

```
! call using generic interface: NUOPC_IsUpdated
function NUOPC_IsUpdatedField(field, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsUpdatedField
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)           :: field
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the field has its "Updated" attribute set to "true". Otherwise return `.false.`.

The arguments are:

field The `ESMF_Field` object to be checked.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.15 NUOPC_IsUpdated - Check if Field(s) in a State are marked as updated

INTERFACE:

```
! call using generic interface: NUOPC_IsUpdated
function NUOPC_IsUpdatedState(state, fieldName, count, rc)
```

RETURN VALUE:

```
logical :: NUOPC_IsUpdatedState
```

ARGUMENTS:

```
type(ESMF_State), intent(in)           :: state
character(*),     intent(in),  optional :: fieldName
integer,          intent(out), optional :: count
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Return `.true.` if the field(s) in `state` have the "Updated" attribute set to "true". Otherwise return `.false.`.

The arguments are:

state The ESMF_State object to be checked.

[fieldName] The name of the field in state to be checked. If provided, and the state does not contain a field with fieldName, return an error in rc. If not provided, check *all* the fields contained in state and return .true. if all the fields are updated.

[count] If provided, the number of fields that are updated are returned. If fieldName is present then count cannot be greater than 1.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.16 NUOPC_NoOp - No-Operation attachable method for GridComp

INTERFACE:

```
subroutine NUOPC_NoOp(gcomp, rc)
```

ARGUMENTS:

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

DESCRIPTION:

No-Op method with an interface that matches the requirements for a attachable method for ESMF_GridComp objects.

The arguments are:

gcomp The ESMF_GridComp object to which this method is attached.

rc Return code; equals ESMF_SUCCESS if there are no errors.

3.9.17 NUOPC_Realize - Realize previously advertised Fields inside a State on a single Grid with internal allocation

INTERFACE:

```
! call using generic interface: NUOPC_Realize
subroutine NUOPC_RealizeComplete(state, grid, typekind, selection, &
  dataFillScheme, rc)
```

ARGUMENTS:

```
type(ESMF_State)                              :: state
type(ESMF_Grid), intent(in)                   :: grid
type(ESMF_TypeKind_Flag), intent(in),  optional :: typekind
character(len=*),         intent(in),  optional :: selection
character(len=*),         intent(in),  optional :: dataFillScheme
integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Realize or remove fields inside of `state` according to `selection`. All of the fields that are realized are created internally on the same `grid` object, allocating memory for as many field dimensions as there are grid dimensions.

The type and kind of the created fields is according to argument `typekind`.

Realized fields are filled with data according to the `dataFillScheme` argument.

The arguments are:

state The `ESMF_State` object in which the fields are realized.

grid The `ESMF_Grid` object on which to realize the fields.

[typekind] The `ESMF_Grid` object on which to realize the fields. The typekind of the internally created field(s). The valid options are `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`, `ESMF_TYPEKIND_R4`, and `ESMF_TYPEKIND_R8` (default).

[selection] Selection of mode of operation:

- `"realize_all"` (default)
- `"realize_connected_remove_others"`

[dataFillScheme] Realized fields will be filled according to the selected fill scheme. See **??** for fill schemes. Default is to leave the data in realized fields uninitialized.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.9.18 NUOPC_Realize - Realize a previously advertised Field in a State

INTERFACE:

```
! call using generic interface: NUOPC_Realize
subroutine NUOPC_RealizeField(state, field, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout)         :: state
type(ESMF_Field), intent(in)            :: field
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Realize a previously advertised field in `state` by replacing the advertised field with `field` of the same name.

The arguments are:

state The ESMF_State object in which the fields are realized.

field The new field to put in place of the previously advertised (empty) field.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.19 NUOPC_SetAttribute - Set the value of a NUOPC Field Attribute

INTERFACE:

```
! call using generic interface: NUOPC_SetAttribute
subroutine NUOPC_SetAttributeField(field, name, value, rc)
```

ARGUMENTS:

```
  type(ESMF_Field)                    :: field
  character(*), intent(in)            :: name
  character(*), intent(in)            :: value
  integer,      intent(out), optional :: rc
```

DESCRIPTION:

Set the attribute name inside of field using the convention NUOPC and purpose Instance.

The arguments are:

field The ESMF_Field object on which to set the attribute.

name The name of the set attribute.

value The value of the set attribute.

[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.20 NUOPC_SetAttribute - Set the value of a NUOPC State Attribute

INTERFACE:

```
! call using generic interface: NUOPC_SetAttribute
subroutine NUOPC_SetAttributeState(state, name, value, rc)
```

ARGUMENTS:

```
  type(ESMF_State)                    :: state
  character(*), intent(in)            :: name
  character(*), intent(in)            :: value
  integer,      intent(out), optional :: rc
```

DESCRIPTION:

Set the attribute `name` inside of `state` using the convention `NUOPC` and purpose `Instance`.

The arguments are:

state The `ESMF_State` object on which to set the attribute.

name The name of the set attribute.

value The value of the set attribute.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

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.

3.10.1 NUOPC_Write - Write a distributed factorList to file

INTERFACE:

```
! call using generic interface: NUOPC_Write
subroutine NUOPC_FactorsWrite(factorList, fileName, rc)
```

ARGUMENTS:

```
real(ESMF_KIND_R8), pointer               :: factorList(:)
character(*),       intent(in)            :: fileName
integer,            intent(out), optional :: rc
```

DESCRIPTION:

Write the destributed `factorList` to file. Each PET calls with its local list of factors. The call then writes the distributed factors into a single file. The order of the factors in the file is first by PET, and within each PET the PET-local order is preserved. Changing the number of PETs for the same regrid operation will likely change the order of factors across PETs, and therefore files written will differ.

The arguments are:

factorList The distributed factor list.

fileName The name of the file to be written to.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.10.2 NUOPC_Write - Write Field data to file

INTERFACE:

```
! call using generic interface: NUOPC_Write
subroutine NUOPC_FieldWrite(field, fileName, overwrite, status, timeslice, &
  iofmt, relaxedflag, rc)
```

ARGUMENTS:

```
type(ESMF_Field),           intent(in)            :: field
character(*),               intent(in)            :: fileName
logical,                    intent(in),  optional :: overwrite
type(ESMF_FileStatus_Flag), intent(in),  optional :: status
integer,                    intent(in),  optional :: timeslice
type(ESMF_IOFmt_Flag),      intent(in),  optional :: iofmt
logical,                    intent(in),  optional :: relaxedflag
integer,                    intent(out), optional :: rc
```

DESCRIPTION:

Write the data in `field` to `file` under the field's "StandardName" attribute if supported by the `iofmt`.

The arguments are:

field The `ESMF_Field` object whose data is to be written.

fileName The name of the file to write to.

[overwrite] A logical flag, the default is .false., i.e., existing Field data may *not* be overwritten. If .true., the data corresponding to each field's name will be be overwritten. If the `timeslice` option is given, only data for the given timeslice may be overwritten. Note that it is always an error to attempt to overwrite a NetCDF variable with data which has a different shape.

[status] The file status. Valid options are `ESMF_FILESTATUS_NEW`, `ESMF_FILESTATUS_OLD`, `ESMF_FILESTATUS_REPLACE`, and `ESMF_FILESTATUS_UNKNOWN` (default).

[timeslice] Time slice counter. Must be positive. The behavior of this option may depend on the setting of the `overwrite` flag:

`overwrite = .false.:` If the timeslice value is less than the maximum time already in the file, the write will fail.

`overwrite = .true.:` Any positive timeslice value is valid.

By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file, however, if the file already contains a time axis for the variable, a timeslice one greater than the maximum will be written.

[iofmt] The IO format. Valid options are `ESMF_IOFMT_BIN` and `ESMF_IOFMT_NETCDF`. If not present, file names with a `.bin` extension will use `ESMF_IOFMT_BIN`, and file names with a `.nc` extension will use `ESMF_IOFMT_NETCDF`. Other files default to `ESMF_IOFMT_NETCDF`.

[relaxedflag] If `.true.`, then no error is returned even if the call cannot write the file due to library limitations. Default is `.false.`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

3.10.3 NUOPC_Write - Write the Fields within a State to NetCDF files

INTERFACE:

```
! call using generic interface: NUOPC_Write
subroutine NUOPC_StateWrite(state, fieldNameList, fileNamePrefix, overwrite, &
  status, timeslice, relaxedflag, rc)
```

ARGUMENTS:

```
type(ESMF_State),           intent(in)            :: state
character(len=*),           intent(in),  optional :: fieldNameList(:)
character(len=*),           intent(in),  optional :: fileNamePrefix
logical,                    intent(in),  optional :: overwrite
type(ESMF_FileStatus_Flag), intent(in),  optional :: status
integer,                    intent(in),  optional :: timeslice
logical,                    intent(in),  optional :: relaxedflag
integer,                    intent(out), optional :: rc
```

DESCRIPTION:

Write the data of the fields within a `state` to NetCDF files. Each field is written to an individual file using the "StandardName" attribute as NetCDF attribute.

The arguments are:

state The `ESMF_State` object containing the fields.

[fieldNameList] List of names of the fields to be written. By default write all the fields in `state`.

[fileNamePrefix] File name prefix, common to all the files written.

[overwrite] A logical flag, the default is .false., i.e., existing Field data may *not* be overwritten. If .true., the data corresponding to each field's name will be be overwritten. If the `timeslice` option is given, only data for the given timeslice may be overwritten. Note that it is always an error to attempt to overwrite a NetCDF variable with data which has a different shape.

[status] The file status. Valid options are `ESMF_FILESTATUS_NEW`, `ESMF_FILESTATUS_OLD`, `ESMF_FILESTATUS_REPLACE`, and `ESMF_FILESTATUS_UNKNOWN` (default).

[timeslice] Time slice counter. Must be positive. The behavior of this option may depend on the setting of the `overwrite` flag:

`overwrite = .false.:` If the timeslice value is less than the maximum time already in the file, the write will fail.

`overwrite = .true.:` Any positive timeslice value is valid.

By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file, however, if the file already contains a time axis for the variable, a timeslice one greater than the maximum will be written.

[relaxedflag] If `.true.`, then no error is returned even if the call cannot write the file due to library limitations. Default is `.false.`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

4 Standardized Component Dependencies

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 to find shared libraries during link-time (and during run-time unless overridden by LD_LIBRARY_PATH). Must be specified as absolute path.
- `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 straight forward. 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://sourceforge.net/p/esmfcontrib/svn/HEAD/tree/NUOPC/trunk/ComponentExplorer/
```

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

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.

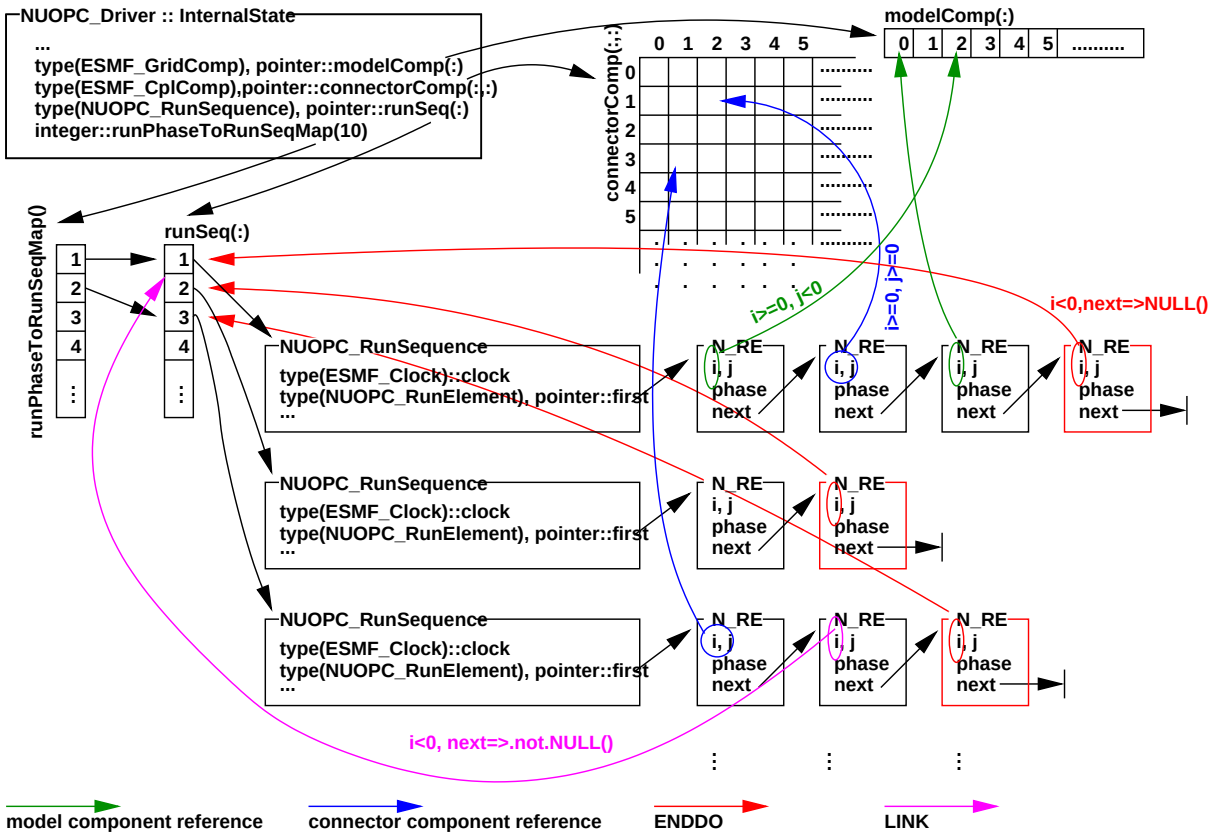

Figure 2: NUOPC_RunSequence class as it relates to the surrounding data structures.