National Unified Operational Prediction Capability

**NUOPC Layer Reference**

ESMF v6.3.0rp1

_CSC Committee Members_

July 11, 2014
# Contents

1. **Description** 3

2. **Design and Implementation Notes** 3
   2.1 Generic Components 3
   2.2 Field Dictionary 5
   2.3 Metadata 6
      2.3.1 Model and Mediator Component Metadata 6
      2.3.2 Connector Component Metadata 7
      2.3.3 Field Metadata 8
   2.4 Initialization 9
      2.4.1 Initialize Phase Definitions 9
      2.4.2 Data-Dependencies during Initialize 11
      2.4.3 Transfer of Grid/Mesh Objects between Components 12

3. **API** 14
   3.1 Generic Component: NUOPC Driver 14
   3.2 Generic Component: NUOPC DriverAtmOcn 16
   3.3 Generic Component: NUOPC DriverAtmOcnMed 18
   3.4 Generic Component: NUOPC ModelBase 20
   3.5 Generic Component: NUOPC Model 21
   3.6 Generic Component: NUOPC Mediator 23
   3.7 Generic Component: NUOPC Connector 25
   3.8 Utility Class: NUOPC RunSequence 28
      3.8.1 NUOPC RunElementAdd 28
      3.8.2 NUOPC RunElementAddComp 29
      3.8.3 NUOPC RunElementAddLink 29
      3.8.4 NUOPC RunElementPrint 30
      3.8.5 NUOPC RunSequenceAdd 30
      3.8.6 NUOPC RunSequenceDeallocate 31
      3.8.7 NUOPC RunSequenceDeallocate 31
      3.8.8 NUOPC RunSequenceIterate 31
      3.8.9 NUOPC RunSequencePrint 32
      3.8.10 NUOPC RunSequencePrint 32
      3.8.11 NUOPC RunSequenceSet 33
   3.9 Utility Routines 34
      3.9.1 NUOPC ClockCheckSetClock 34
      3.9.2 NUOPC ClockInitialize 34
      3.9.3 NUOPC ClockPrintCurrTime 35
      3.9.4 NUOPC ClockPrintStartime 35
      3.9.5 NUOPC ClockPrintStopTime 35
      3.9.6 NUOPC CplCompAreServicesSet 36
      3.9.7 NUOPC CplCompAttributeAdd 36
      3.9.8 NUOPC CplCompAttributeGet 37
      3.9.9 NUOPC CplCompAttributeSet 37
      3.9.10 NUOPC FieldAttributeAdd 38
      3.9.11 NUOPC FieldAttributeGet 39
      3.9.12 NUOPC FieldAttributeSet 39
      3.9.13 NUOPC FieldBundleUpdateTime 40
      3.9.14 NUOPC FieldDictionaryAddEntry 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.9.15 NUOPC_FieldDictionaryGetEntry</td>
<td>40</td>
</tr>
<tr>
<td>3.9.16 NUOPC_FieldDictionaryHasEntry</td>
<td>41</td>
</tr>
<tr>
<td>3.9.17 NUOPC_FieldDictionarySetup</td>
<td>41</td>
</tr>
<tr>
<td>3.9.18 NUOPC_FieldIsAtTime</td>
<td>42</td>
</tr>
<tr>
<td>3.9.19 NUOPC_FillCplList</td>
<td>42</td>
</tr>
<tr>
<td>3.9.20 NUOPC_GridCompAreServicesSet</td>
<td>42</td>
</tr>
<tr>
<td>3.9.21 NUOPC_GridCompAttributeAdd</td>
<td>43</td>
</tr>
<tr>
<td>3.9.22 NUOPC_GridCompCheckSetClock</td>
<td>43</td>
</tr>
<tr>
<td>3.9.23 NUOPC_GridCompSetClock</td>
<td>44</td>
</tr>
<tr>
<td>3.9.24 NUOPC_GridCompSetServices</td>
<td>44</td>
</tr>
<tr>
<td>3.9.25 NUOPC_GridCreateSimpleXY</td>
<td>45</td>
</tr>
<tr>
<td>3.9.26 NUOPC_IsCreated</td>
<td>45</td>
</tr>
<tr>
<td>3.9.27 NUOPC_StateAdvertiseField</td>
<td>46</td>
</tr>
<tr>
<td>3.9.28 NUOPC_StateAdvertiseFields</td>
<td>47</td>
</tr>
<tr>
<td>3.9.29 NUOPC_StateBuildStdList</td>
<td>47</td>
</tr>
<tr>
<td>3.9.30 NUOPC_StateIsAllConnected</td>
<td>48</td>
</tr>
<tr>
<td>3.9.31 NUOPC_StateIsAtTime</td>
<td>48</td>
</tr>
<tr>
<td>3.9.32 NUOPC_StateIsFieldConnected</td>
<td>48</td>
</tr>
<tr>
<td>3.9.33 NUOPC_StateIsUpdated</td>
<td>49</td>
</tr>
<tr>
<td>3.9.34 NUOPC_StateRealizeField</td>
<td>49</td>
</tr>
<tr>
<td>3.9.35 NUOPC_StateSetTimestamp</td>
<td>50</td>
</tr>
<tr>
<td>3.9.36 NUOPC_StateUpdateTimestamp</td>
<td>50</td>
</tr>
<tr>
<td>3.9.37 NUOPC_TimePrint</td>
<td>51</td>
</tr>
<tr>
<td>4 Standardized Component Dependencies</td>
<td>52</td>
</tr>
<tr>
<td>4.1 Fortran components that are statically built into the executable</td>
<td>53</td>
</tr>
<tr>
<td>4.2 Fortran components that are provided as shared libraries</td>
<td>56</td>
</tr>
<tr>
<td>4.3 Components that are loaded during run-time as shared objects</td>
<td>57</td>
</tr>
<tr>
<td>4.4 Components that depend on components</td>
<td>58</td>
</tr>
<tr>
<td>4.5 Components written in C/C++</td>
<td>60</td>
</tr>
<tr>
<td>5 NUOPC Layer Compliance</td>
<td>63</td>
</tr>
<tr>
<td>5.1 The Compliance Checker</td>
<td>63</td>
</tr>
<tr>
<td>5.2 The Component Explorer</td>
<td>65</td>
</tr>
</tbody>
</table>
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:

```fortran
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 generic components also define a public internal state type by the standard name InternalState. It is recommended that the following syntax is used when accessing a generic component (here with internal state):

```
use NUOPC_DriverXYZ, only: &
    DriverXYZ_SS => SetServices, &
    DriverXYZ_IS => InternalState
```

A generic component is used by user code to implement a specialized version of the component. The user code therefore also must implement a public SetServices routine. The first thing this routine must do is call into the SetServices routine provided by the generic component. It is through this step that the specialized component inherits from the generic component.

There are three mechanisms through which user code specializes generic components.

1. The specializing user code must set entry points for standard component methods not implemented by the generic component. 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 is done in the SetServices routine right after calling into the generic SetServices method.

2. Some generic components require that specific methods are attached to the component. 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 provide access to an internal state type. The documentation of a generic component indicates which internal state members are used for specialization, and how they are expected to be set. Setting internal state members often requires the availability of other pieces of information. It may happen in the SetServices routine, but more often inside a specialized standard entry point or an attachable method.

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 NUOPC Generic Components. There are two trees, one is rooted in ESMF_GridComp, while the other is rooted in ESMF_CplComp.
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 Attribute. It also associates each registered StandardName with canonical Units, a default LongName, and a default ShortName.

The NUOPC Layer provides a number of default entries in the Field Dictionary, shown in the table below. The StandardName Attribute of all default entries complies with the Climate and Forecast (CF) conventions as documented at [http://cf-pcmdi.llnl.gov/](http://cf-pcmdi.llnl.gov/).

Currently it is typically that a user of the NUOPC Layer extends the Field Dictionary by calling the NUOPC_FieldDictionaryAddEntry() interface to add additional entries. It is our intention to grow the number of default entries over time, and to more strongly leverage the NUOPC Field Dictionary to ensure metadata interoperability between codes that use the NUOPC Layer.

Besides the StandardName Attribute, the NUOPC Layer currently only uses the Units entry to verify that Fields are given in their canonical units. The plan is to extend this to support unit conversion in the future. The default LongName and default ShortName associations are provided as a convenience to the implementor of NUOPC compliant components; the NUOPC Layer itself does not base any decisions on these two Attributes.

<table>
<thead>
<tr>
<th>StandardName</th>
<th>Units</th>
<th>LongName</th>
<th>ShortName</th>
</tr>
</thead>
<tbody>
<tr>
<td>air_pressure_at_sea_level</td>
<td>Pa</td>
<td>Air Pressure at Sea Level</td>
<td>pmsl</td>
</tr>
</tbody>
</table>

![Diagram of NUOPC Generic Component inheritance structure](image-url)
### Metadata

#### 2.3.1 Model and Mediator Component Metadata

The Model and Mediator Component metadata is implemented as an ESMF Attribute Package:

- **Convection**: NUOPC
- **Purpose**: General
- **Includes**:
  - CIM Model Component Simulation Description (see for example the Component Attribute packages section in the ESMF v5.2.0rp2 documentation)
- **Description**: Model/Mediator component description and nesting metadata.
<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Controlled Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verbosity</td>
<td>String value controlling the verbosity of INFO messages.</td>
<td>high, low</td>
</tr>
<tr>
<td>InitializePhaseMap</td>
<td>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.</td>
<td>IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z &gt; 0 and Y, Z &lt; 10</td>
</tr>
<tr>
<td>NestingGeneration</td>
<td>Integer value enumerating nesting level.</td>
<td>0, 1, 2, ...</td>
</tr>
<tr>
<td>Nestling</td>
<td>Integer value enumerating siblings within the same generation.</td>
<td>0, 1, 2, ...</td>
</tr>
<tr>
<td>InitializeDataComplete</td>
<td>String value indicating whether all initialize data dependencies have been satisfied.</td>
<td>false, true</td>
</tr>
<tr>
<td>InitializeDataProgress</td>
<td>String value indicating whether progress is being made resolving initialize data dependencies.</td>
<td>false, true</td>
</tr>
</tbody>
</table>

### 2.3.2 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)
- Description: Basic component description and connection metadata.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Controlled Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verbosity</td>
<td>String value controlling the verbosity of INFO messages.</td>
<td>high, low</td>
</tr>
<tr>
<td>InitializePhaseMap</td>
<td>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.</td>
<td>IPDvXXpY=Z, where XX = two-digit revision number, e.g. 01, Y = logical NUOPC phase number, Z = actual ESMF phase number, with Y, Z &gt; 0 and Y, Z &lt; 10</td>
</tr>
<tr>
<td>CplList</td>
<td>List of StandardNames of the connected Fields.</td>
<td>N/A</td>
</tr>
</tbody>
</table>
2.3.3 Field Metadata

The Field metadata is implemented as an ESMF Attribute Package:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Controlled Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected</td>
<td>Connected status.</td>
<td>false, true</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>Nine integer values representing ESMF Time object.</td>
<td>N/A</td>
</tr>
<tr>
<td>ProducerConnection</td>
<td>String value indicating connection details.</td>
<td>open, targeted, connected</td>
</tr>
<tr>
<td>ConsumerConnection</td>
<td>String value indicating connection details.</td>
<td>open, targeted, connected</td>
</tr>
<tr>
<td>Updated</td>
<td>String value indicating updated status during initialization.</td>
<td>false, true</td>
</tr>
<tr>
<td>TransferOfferGeomObject</td>
<td>String value indicating a component’s intention to transfer the underlying Grid or Mesh on which an advertised Field object is defined.</td>
<td>will provide, can provide, cannot provide</td>
</tr>
<tr>
<td>TransferActiveGeomObject</td>
<td>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.</td>
<td>provide, accept</td>
</tr>
</tbody>
</table>
2.4 Initialization

2.4.1 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. The following tables document the meaning of each initialization phase for the two different child component groups for the different IPD versions. The phases are listed in the prescribed sequence used by the driver.

<table>
<thead>
<tr>
<th>IPDv00 label</th>
<th>Child Group</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPDv00p1</td>
<td>model, mediator, driver</td>
<td>Advertise the import and export Fields.</td>
</tr>
<tr>
<td>IPDv00p1</td>
<td>connector</td>
<td>Construct the CplList Attribute on the connector.</td>
</tr>
<tr>
<td>IPDv00p2</td>
<td>model, mediator, driver</td>
<td>Realize the import and export Fields.</td>
</tr>
<tr>
<td>IPDv00p2</td>
<td>connector</td>
<td>Set the Connected Attribute on each import and export Field. Precompute the RouteHandle.</td>
</tr>
<tr>
<td>IPDv00p3</td>
<td>model, mediator, driver</td>
<td>Check compatibility of the Fields' Connected status.</td>
</tr>
<tr>
<td>IPDv00p4</td>
<td>model, mediator, driver</td>
<td>Handle Field data initialization. Time stamp the export Fields.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPDv01 label</th>
<th>Child Group</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPDv01p1</td>
<td>model, mediator, driver</td>
<td>Advertise the import and export Fields.</td>
</tr>
<tr>
<td>IPDv01p1</td>
<td>connector</td>
<td>Construct the CplList Attribute on the connector.</td>
</tr>
<tr>
<td>IPDv01p2</td>
<td>model, mediator, driver</td>
<td>unspecified</td>
</tr>
<tr>
<td>IPDv01p2</td>
<td>connector</td>
<td>Set the Connected Attribute on each import and export Field.</td>
</tr>
<tr>
<td>IPDv01p3</td>
<td>model, mediator, driver</td>
<td>Realize the import and export Fields.</td>
</tr>
<tr>
<td>IPDv01p3</td>
<td>connector</td>
<td>Precompute the RouteHandle.</td>
</tr>
<tr>
<td>IPDv01p4</td>
<td>model, mediator, driver</td>
<td>Check compatibility of the Fields' Connected status.</td>
</tr>
<tr>
<td>IPDv01p5</td>
<td>model, mediator, driver</td>
<td>Handle Field data initialization. Time stamp the export Fields.</td>
</tr>
<tr>
<td>IPDv02 label</td>
<td>Child Group</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>IPDv02p1</td>
<td>model, mediator, driver</td>
<td>Advertise the import and export Fields.</td>
</tr>
<tr>
<td>IPDv02p1</td>
<td>connector</td>
<td>Construct the CplList Attribute on the connector.</td>
</tr>
<tr>
<td>IPDv02p2</td>
<td>model, mediator, driver</td>
<td>unspecified</td>
</tr>
<tr>
<td>IPDv02p2</td>
<td>connector</td>
<td>Set the Connected Attribute on each import and export Field.</td>
</tr>
<tr>
<td>IPDv02p3</td>
<td>model, mediator, driver</td>
<td>Realize the import and export Fields.</td>
</tr>
<tr>
<td>IPDv02p3</td>
<td>connector</td>
<td>Precompute the RouteHandle.</td>
</tr>
<tr>
<td>IPDv02p4</td>
<td>model, mediator, driver</td>
<td>Check compatibility of the Fields' Connected status.</td>
</tr>
<tr>
<td>IPDv02p5</td>
<td>model, mediator, driver</td>
<td>Handle Field data initialization. Times-tamp the export Fields.</td>
</tr>
</tbody>
</table>

A loop is entered over all those model, mediator, driver Components that use IPDv02 and have unsatisfied data dependencies, repeating the following two steps:

<table>
<thead>
<tr>
<th>Run ()</th>
<th>connector</th>
<th>Loop over all Connectors that connect to the Component that is currently indexed by the outer loop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPDv02p5</td>
<td>model, mediator, driver</td>
<td>Handle Field data initialization. Time stamp the export Fields.</td>
</tr>
</tbody>
</table>

Repeat these two steps until all data dependencies have been satisfied, or a dead-lock situation is detected.
<table>
<thead>
<tr>
<th>IPDv03 label</th>
<th>Child Group</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPDv03p1</td>
<td>model, mediator, driver</td>
<td>Advertise the import and export Fields, setting TransferOfferGeomObject.</td>
</tr>
<tr>
<td>IPDv03p1</td>
<td>connector</td>
<td>Construct the CplList Attribute on the connector.</td>
</tr>
<tr>
<td>IPDv03p2</td>
<td>model, mediator, driver</td>
<td>unspecified</td>
</tr>
<tr>
<td>IPDv03p2</td>
<td>connector</td>
<td>Set the Connected Attribute on each import and export Field. Set the TransferActionGeomObject attribute.</td>
</tr>
<tr>
<td>IPDv03p3</td>
<td>model, mediator, driver</td>
<td>Realize the import and export Fields that have TransferActionGeomObject equal to &quot;provide&quot;.</td>
</tr>
<tr>
<td>IPDv03p3</td>
<td>connector</td>
<td>Transfer the Grid/Mesh objects (only DistGrid) for Field pairs that have a provider and an acceptor side.</td>
</tr>
<tr>
<td>IPDv03p4</td>
<td>model, mediator, driver</td>
<td>Optionally modify the decomposition and distribution information of the accepted Grid/Mesh by replacing the DistGrid.</td>
</tr>
<tr>
<td>IPDv03p4</td>
<td>connector</td>
<td>Transfer the full Grid/Mesh objects (with coordinates) for Field pairs that have a provider and an acceptor side.</td>
</tr>
<tr>
<td>IPDv03p5</td>
<td>model, mediator, driver</td>
<td>Realize all Fields that have TransferActionGeomObject equal to &quot;accept&quot; on the transferred Grid/Mesh objects.</td>
</tr>
<tr>
<td>IPDv03p5</td>
<td>connector</td>
<td>Precompute the RouteHandle.</td>
</tr>
<tr>
<td>IPDv03p6</td>
<td>model, mediator, driver</td>
<td>Check compatibility of the Fields' Connected status.</td>
</tr>
<tr>
<td>IPDv03p7</td>
<td>model, mediator, driver</td>
<td>Handle Field data initialization. Time stamp the export Fields.</td>
</tr>
</tbody>
</table>

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()  
Loop over all Connectors that connect to the Component that is currently indexed by the outer loop.

IPDv03p7  
Handle Field data initialization. Time stamp the export Fields.

Repeat these two steps until all data dependencies have been satisfied, or a dead-lock situation is detected.

2.4.2 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 type 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 2.4.1. 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 section 2.3.1.

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 2.3.3) 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.

### 2.4.3 Transfer of Grid/Mesh 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. Here often the Mediator carries out computations on a Model’s native grid. It is both cumbersome and error prone to re-defined the same physical grid in two different components. The Initialize Phase Definition version 3 (IPDv03), defined in section 2.4.1, supports the transfer of ESMF Grid and Mesh objects between Model and/or Mediator components during initialization.

The NUOPC Layer transfer protocol for GeomObjects (i.e. ESMF Grids and Meshes) 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 or Mesh object in the Model/Mediator initialize method for phase IPDv03p3.

Fields that hold "accept" for the value of the TransferActionGeomObject attribute require two additional ne-
negotiation steps. By IPDv03p4 the Model/Mediator component can access the transferred Grid/Mesh on the Fields that have the "accept" value. However, only the DistGrid, i.e. the decomposition and distribution information of the Grid/Mesh 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. The DistGrid that is set on the Grid/Mesh objects when leaving the Model/Mediator phase IPDv03p4 will consequently be used by the generic Connector to fully transfer the Grid/Mesh 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.

<table>
<thead>
<tr>
<th>TransferOfferGeomObject A</th>
<th>TransferOfferGeomObject B</th>
<th>Outgoing setting by generic Connector</th>
</tr>
</thead>
</table>
| "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!                 |
3 API

3.1 Generic Component: NUOPC_Driver

MODULE:

module NUOPC_Driver

DESCRIPTION:
Driver component that drives Model, Mediator, and Connector components. The default is to use explicit time stepping. For every Driver time step the same sequence of Model, Mediator, and Connector Run methods are called. The run sequence is fully customizable.

SUPER:

ESMF_GridComp

USE DEPENDENCIES:

use ESMF

SETSERVICES:

subroutine routine_SetServices(gcomp, rc)
    type(ESMF_GridComp) :: gcomp
    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 2.4.1 for a precise definition), with the following mapping:
    * IPDv00p1 = phase 1: (REQUIRED, NUOPC PROVIDED)

• phase 1: (NUOPC PROVIDED, suitable for: IPDv00p1)
  – Allocate and initialize the internal state.
  – If the internal clock is not yet set, set the default internal clock to be a copy of the incoming clock, if the incoming clock is valid.
  – Required specialization to set number of Model+Mediator components, modelCount, in the internal state: label_SetModelCount.
  – Allocate internal storage according to modelCount.
  – Optional specialization to provide Model, Mediator, and Connector petList members in the internal state: label_SetModelPetList.
  – Create Model and Mediator components with their import and export States.
  – Attach standard NUOPC Model Component metadata.
  – Create Connector components.
  – Attach standard NUOPC Connector Component metadata.
  – Initialize the default run sequence.
- **Required specialization** to set component services: `label_SetModelServices`.
  * Call into `SetServices()` for all Model, Mediator, and Connector components.
  * Optionally replace the default clock.
  * Optionally replace the default run sequence.

- Execute Initialize phase=0 for all Model, Mediator, and Connector components. This is the method where each component is required to initialize its `InitializePhaseMap` Attribute.

- **Optional specialization** to analyze and modify components' `InitializePhaseMap` Attribute before the Driver uses it: `label_ModifyInitializePhaseMap`.

- Implement the initialize sequence for the child components, compatible with up to IPD version 02, as documented in section 2.4.1.

**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.
  - 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)*
  - **Optional specialization** to finalize driver component: `label_Finalize`.
  - Execute all Connector components’ Finalize() methods in order.
  - Execute all Model components’ Finalize() methods in order.
  - Destroy all Model components and their import and export states.
  - Destroy all Connector components.
  - Deallocate the run sequence.
  - Deallocate the internal state.

**INTERNALSTATE:**

```fortran
label_InternalState

type type_InternalState
  type(type_InternalStateStruct), pointer :: wrap
end type

type type_InternalStateStruct
  integer :: modelCount
  type(type_PetList), pointer :: modelPetLists(:)
  type(type_PetList), pointer :: connectorPetLists(:, :)
  !--- private members ----------------------------------------
  type(ESMF_GridComp), pointer :: modelComp(:)
  type(ESMF_State), pointer :: modelIS(:), modelES(:)
  type(ESMF_CplComp), pointer :: connectorComp(:, :)
  type(NUOPC_RunSequence), pointer :: runSeq(:) ! size may increase dynamic.
  integer :: runPhaseToRunSeqMap(10)
  type(ESMF_Clock) :: driverClock ! clock of the parent
```

15
3.2 Generic Component: NUOPC_DriverAtmOcn

module NUOPC_DriverAtmOcn

DESCRIPTION:
This is a specialization of the NUOPC_Driver generic component, driving a coupled Atmosphere-Ocean model. The default is to use explicit time stepping. Each driver time step, the same sequence of Atmosphere, Ocean and connector Run methods are called. The run sequence is fully customizable for cases where explicit time stepping is not suitable.

SUPER:
NUOPC_Driver

USE DEPENDENCIES:
use ESMF

SETSERVICES:
subroutine routine_SetServices(gcomp, rc)
    type(ESMF_GridComp) :: gcomp
    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 2.4.1 for a precise definition), with the following mapping:
    * IPDv00p1 = phase 1: (REQUIRED, NUOPC PROVIDED)
- phase 1: (NUOPC PROVIDED, suitable for: IPDv00p1)
  - Allocate and initialize the internal state.
  - If the internal clock is not yet set, set the default internal clock to be a copy of the incoming clock, if the incoming clock is valid.
  - Set the number of model components to 2.
  - Allocate internal storage according to modelCount = 2.
  - Optional specialization to provide Model and Connector petList members in the internal state: label_SetModelPetList.
  - Create atm and ocn Model components with their import and export States.
  - Attach standard NUOPC Model Component metadata.
- Create atm2ocn and ocn2atm Connector components.
- Attach standard NUOPC Connector Component metadata.
- Initialize the default run sequence.
- **Required specialization** to set component services: `label_SetModelServices`.
  * Call into `SetServices()` for the atm, ocn, atm2ocn, and ocn2atm components.
  * Optionally replace the default clock.
  * Optionally replace the default run sequence.
- Execute Initialize phase=0 for all Model, and Connector components. This is the method where each component is required to initialize its `InitializePhaseMap` attribute.
- Implement the initialize sequence for the child components, compatible with up to IPD version 02, as documented in section 2.4.1.

**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.
  - Time stepping loop, from current time to stop time, incrementing by time step.
  - For each time step iteration, the Run() methods for atm, ocn, atm2ocn, and ocn2atm are being called according to the run sequence.

**FINALIZE:**

- **phase 1: (REQUIRED, NUOPC PROVIDED)**
  - Optional specialization to finalize driver component: `label_Finalize`.
  - Execute Finalize() for atm2ocn and ocn2atm.
  - Execute Finalize() for atm and ocn.
  - Destroy atm and ocn and their import and export States.
  - Destroy atm2ocn and ocn2atm.
  - Deallocate the run sequence.
  - Deallocate the internal state.

**INTERNALSTATE:**

```plaintext
label_InternalState

type type_InternalState
   type(type/InternalStateStruct), pointer :: wrap
end type

type type/InternalStateStruct
   integer, pointer :: atmPetList(:)
   integer, pointer :: ocnPetList(:)
   type(ESMF_GridComp) :: atm
   type(ESMF_GridComp) :: ocn
   type(ESMF_State) :: atmIS, atmES
   type(ESMF_State) :: ocnIS, ocnES
   integer, pointer :: atm2ocnPetList(:)
   integer, pointer :: ocn2atmPetList(:)
   type(ESMF_CplComp) :: atm2ocn, ocn2atm
   type(NUOPC_RunSequence), pointer :: runSeq(:)
end type
```

17
3.3 Generic Component: NUOPC_DriverAtmOcnMed

MODULE:

module NUOPC_DriverAtmOcnMed

DESCRIPTION:
This is a specialization of the NUOPC_Driver generic component, driving a coupled Atmosphere-Ocean-Mediator model. The default is to use explicit time stepping. Each driver time step, the same sequence of Atmosphere, Ocean, Mediator, and the connector Run methods are called. The run sequence is fully customizable for cases where explicit time stepping is not suitable.

SUPER:
NUOPC_Driver

USE DEPENDENCIES:
use ESMF

SETSERVICES:

subroutine routine_SetServices(gcomp, rc)
type(ESMF_GridComp) :: gcomp
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 2.4.1 for a precise definition), with the following mapping:
    * IPDv00p1 = phase 1: (REQUIRED, NUOPC PROVIDED)

• phase 1: (NUOPC PROVIDED, suitable for: IPDv00p1)
  − Allocate and initialize the internal state.
  − If the internal clock is not yet set, set the default internal clock to be a copy of the incoming clock, if the incoming clock is valid.
  − Set the number of model components to 3.
  − Allocate internal storage according to modelCount = 3.
  − Optional specialization to provide Model and Connector petList members in the internal state:
    label_SetModelPetList.
  − Create atm, ocn, and med components with their import and export States.
  − Attach standard NUOPC Model Component metadata.
  − Create atm2med, ocn2med, med2atm, and med2ocn Connector components.
  − Attach standard NUOPC Connector Component metadata.
  − Initialize the default run sequence.
  − Required specialization to set component services: label_SetModelServices.
    * Call into SetServices() for the atm, ocn, med, atm2med, ocn2med, med2atm, and med2ocn components.
* Optionally replace the default clock.
* Optionally replace the default run sequence.

- Execute Initialize phase=0 for all Model, Mediator, and Connector components. This is the method where each component is required to initialize its InitializePhaseMap Attribute.
- Implement the initialize sequence for the child components, compatible with up to IPD version 02, as documented in section [2.4.1](#).

**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.
  - Time stepping loop, from current time to stop time, incrementing by time step.
  - For each time step iteration, the Run() methods for atm, ocn, med, atm2med, ocn2med, med2atm, and med2ocn are being called according to the run sequence.

**FINALIZE:**
- phase 1: (REQUIRED, NUOPC PROVIDED)
  - Optional specialization to finalize driver component: label_Finalize.
  - Execute Finalize() for atm2med, ocn2med, med2atm, and med2ocn.
  - Destroy atm, ocn, and med and their import and export States.
  - Destroy atm2med, ocn2med, med2atm, and med2ocn.
  - Deallocate the run sequence.
  - Deallocate the internal state.

**INTERNALSTATE:**

```fortran
label_InternalState

type type_InternalState
  type(type_InternalStateStruct), pointer :: wrap
end type

type type_InternalStateStruct
  integer, pointer :: atmPetList(:)
  integer, pointer :: ocnPetList(:)
  integer, pointer :: medPetList(:)
  type(ESMF_GridComp) :: atm
  type(ESMF_GridComp) :: ocn
  type(ESMF_GridComp) :: med
  type(ESMF_State) :: atmIS, atmES
  type(ESMF_State) :: ocnIS, ocnES
  type(ESMF_State) :: medIS, medES
  integer, pointer :: atm2medPetList(:)
  integer, pointer :: ocn2medPetList(:)
  integer, pointer :: med2atmPetList(:)
  integer, pointer :: med2ocnPetList(:)
  type(ESMF_CplComp) :: atm2med, ocn2med
  type(ESMF_CplComp) :: med2atm, med2ocn
  type(NUOPC_RunSequence), pointer :: runSeq(:)
end type
```

19
### 3.4 Generic Component: NUOPC_ModelBase

**MODULE:**

```module NUOPC_ModelBase```
```
DESCRIPTION:
Model component with a default *explicit* time dependency. Each time the `Run` method is called the model integrates one `timeStep` forward on the provided `Clock`. The `Clock` must be advanced between `Run` calls. The component’s `Run` method flags incompatibility if the current time of the incoming `Clock` does not match the current time of the model.

**SUPER:**

`ESMF_GridComp`

**USE DEPENDENCIES:**

`use ESMF`

**SETSERVICES:**

```subroutine routine_SetServices(gcomp, rc)```
```
```type(ESMF_GridComp) :: gcomp```
```
```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 2.4.1 for a precise definition), with the following mapping:
    * IPDv00p1 = phase 1: *(REQUIRED, IMPLEMENTOR PROVIDED)*
    * IPDv00p2 = phase 2: *(REQUIRED, IMPLEMENTOR PROVIDED)*
    * IPDv00p3 = phase 3: *(REQUIRED, IMPLEMENTOR PROVIDED)*
    * IPDv00p4 = phase 4: *(REQUIRED, IMPLEMENTOR PROVIDED)*

**RUN:**

- phase 1: *(REQUIRED, NUOPC PROVIDED)*
  - Allocate internal state memory.
  - Assign the `driverClock` member in the internal state as an alias to the incoming `Clock`.
  - **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.

  - Deallocate the internal state memory.

FINALIZE:
  • phase I: (REQUIRED, NUOPC PROVIDED)
    - Optionally overwrite the provided NOOP with model finalization code.

INTERNALSTATE:

  label_InternalState

  type type_InternalState
  
  type(type_InternalStateStruct), pointer :: wrap
  end type

  type type_InternalStateStruct
  
  type(ESMF_Clock) :: driverClock
  end type

3.5 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 provided Clock. The Clock must be advanced between Run calls. The component’s Run method flags incompatibility if the current time of the incoming Clock does not match the current time of the model.

SUPER:

  NUOPC_ModelBase

USE DEPENDENCIES:

  use ESMF

SERVICES:

  subroutine routine_SetServices(gcomp, rc)
  
  type(ESMF_GridComp) :: gcomp
  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 2.4.1 for a precise definition), with the following mapping:
    * IPDv00p1 = phase 1: (REQUIRED, IMPLEMENTOR PROVIDED)
      - Advertise Fields in import and export States.
    * IPDv00p2 = phase 2: (REQUIRED, IMPLEMENTOR PROVIDED)
      - Realize the advertised Fields in import and export States.
    * IPDv00p3 = phase 3: (REQUIRED, NUOPC PROVIDED)
      - Check compatibility of the Fields' Connected status.
    * IPDv00p4 = phase 4: (REQUIRED, NUOPC PROVIDED)
      - Handle Field data initialization. Time stamp the export Fields.
- phase 3: (NUOPC PROVIDED, suitable for: IPDv00p3, IPDv01p4, IPDv02p4)
  - 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.
- phase 4: (NUOPC PROVIDED, suitable for: IPDv00p4, IPDv01p5)
  - Optional specialization to initialize export Fields: label_DataInitialize
  - Time stamp Fields in export State for compatibility checking.
- phase 5: (NUOPC PROVIDED, suitable for: IPDv02p5)
  - 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)
  - Allocate internal state memory.
  - Assign the driverClock member in the internal state as an alias to the incoming Clock.
  - 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.
  – Deallocate the internal state memory.

FINALIZE:

  ● phase 1: (REQUIRED, NUOPC PROVIDED)
  
  – Optionally overwrite the provided NOOP with model finalization code.

INTERNALSTATE:

  label_InternalState

  type type_InternalState
  
  type(type_InternalStateStruct), pointer :: wrap
  end type

  type type_InternalStateStruct
  
  type(ESMF_Clock) :: driverClock
  end type

3.6 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 routine_SetServices(gcomp, rc)
  
  type(ESMF_GridComp) :: gcomp
  integer, intent(out) :: rc

INITIALIZE:
• phase 0: (REQUIRED, NUOPC PROVIDED)
  – Initialize the \texttt{InitializePhaseMap} Attribute according to the NUOPC Initialize Phase Definition (IPD) version 00 (see section 2.4.1 for a precise definition), with the following mapping:
    * IPDv00p1 = phase 1: (REQUIRED, IMPLEMENTOR PROVIDED)
      · Advertise Fields in import and export States.
    * IPDv00p2 = phase 2: (REQUIRED, IMPLEMENTOR PROVIDED)
      · Realize the advertised Fields in import and export States.
    * IPDv00p3 = phase 3: (REQUIRED, NUOPC PROVIDED)
      · Check compatibility of the Fields’ Connected status.
    * IPDv00p4 = phase 4: (REQUIRED, NUOPC PROVIDED)
      · Handle Field data initialization. Time stamp the export Fields.

• phase 3: (NUOPC PROVIDED, suitable for: IPDv00p3, IPDv01p4, IPDv02p4)
  – Set the Mediator internal clock as a copy of the incoming clock.
  – Check compatibility, ensuring all advertised import Fields are connected.

• phase 4: (NUOPC PROVIDED, suitable for: IPDv00p4, IPDv01p5)
  – \textit{Optional specialization} to initialize export Fields: \texttt{label\_DataInitialize}
  – Time stamp Fields in import and export States for compatibility checking.

• phase 5: (NUOPC PROVIDED, suitable for: IPDv02p5)
  – \textit{Optional specialization} to initialize export Fields: \texttt{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)
  – Allocate internal state memory.
  – Assign the \texttt{driverClock} member in the internal state as an alias to the incoming Clock.
  – \texttt{SPECIALIZATION REQUIRED/PROVIDED: label\_SetRunClock} to check and set the internal Clock against the incoming Clock.
    * \texttt{IF} (Phase specific specialization present): Execute the phase specific specialization.
    * \texttt{ELSE}: Execute the phase independent specialization. \texttt{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.
  – \texttt{SPECIALIZATION REQUIRED/PROVIDED: label\_CheckImport} to check Fields in the import State.
    * \texttt{IF} (Phase specific specialization is present): Execute the phase specific specialization.
    * \texttt{ELSE}: Execute the phase independent specialization. \texttt{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.
    * \texttt{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.
  – \texttt{SPECIALIZATION OPTIONAL: label\_TimestampExport} to timestamp Fields in the export State.
Deallocate the internal state memory.

FINALIZE:

• phase 1: (REQUIRED, NUOPC PROVIDED)
  – Optionally overwrite the provided NOOP with Mediator finalization code.

INTERNALSTATE:

label_InternalState
type type_InternalState
  type(type_InternalStateStruct), pointer :: wrap
deend type
type type_InternalStateStruct
  type(ESMF_Clock) :: driverClock
deend type

3.7 Generic Component: NUOPC_Connector

MODULE:

module NUOPC_Connector
deendmodule

DESCRIPTION:
Connector component that uses a default bilinear regrid method during Run to transfer data from the connected import Fields to the connected export Fields.

SUPER:

ESMF_CplComp

USE DEPENDENCIES:

use ESMF

SETSERVICES:

subroutine routine_SetServices(cplcomp, rc)
  type(ESMF_CplComp) :: cplcomp
  integer, intent(out) :: rc
deend subroutine

INITIALIZE:

• phase 0: (REQUIRED, NUOPC PROVIDED)
  – Initialize the InitializePhaseMap Attribute according to the NUOPC Initialize Phase Definition (IPD) version 01 (see section 2.4.1 for a precise definition), with the following mapping:
    * IPDv01p1 = phase 1: (REQUIRED, NUOPC PROVIDED)
* IPDv01p2 = phase 2: (REQUIRED, NUOPC PROVIDED)
* IPDv01p3 = phase 3: (REQUIRED, NUOPC PROVIDED)

• phase 1: (NUOPC PROVIDED, suitable for: IPDv01p1, IPDv02p1)
  - 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.

• phase 2: (NUOPC PROVIDED, suitable for: IPDv01p2, IPDv02p2)
  - 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.

• phase 3: (NUOPC PROVIDED, suitable for: IPDv01p3, IPDv02p3)
  - 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 a Connector operation: label_ReleaseRouteHandle.
  - By default (if label_ReleaseRouteHandle was not provided) release the precomputed Connector RouteHandle.
  - Destroy the internal state members.
– Deallocate the internal state.

**INTERNALSTATE:**

```fortran
label_InternalState

type type_InternalState
   type(type_InternalStateStruct), pointer :: wrap
end type

type type_InternalStateStruct
   type(ESMF_FieldBundle) :: srcFields
   type(ESMF_FieldBundle) :: dstFields
   type(ESMF_RouteHandle) :: rh
   type(ESMF_State) :: state
end type
```
3.8 Utility Class: NUOPC_RunSequence

The NUOPC_RunSequence class 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.

3.8.1 NUOPC_RunElementAdd - Add a RunElement to the end of a RunSequence

INTERFACE:

subroutine NUOPC_RunElementAdd(runSeq, i, j, phase, rc)

ARGUMENTS:

 type(NUOPC_RunSequence), intent(inout), target :: runSeq
 integer, intent(in) :: i, j, phase
 integer, optional, intent(out) :: rc
DESCRIPTION:

Add a new RunElement at the end of an existing RunSequence. The RunElement is set to the values provided for \(i\), \(j\), \(phase\).

3.8.2 NUOPC_RunElementAddComp - Add a RunElement for a Component to the end of a RunSequence

INTERFACE:

```fortran
subroutine NUOPC_RunElementAddComp(runSeq, i, j, phase, rc)
ARGUMENTS:
    type(NUOPC_RunSequence), intent(inout), target :: runSeq
    integer, intent(in) :: i
    integer, intent(in), optional :: j
    integer, intent(in), optional :: phase
    integer, optional, intent(out) :: rc
```

DESCRIPTION:

Add a new RunElement for a Component to the end of an existing RunSequence. The RunElement is set to the values provided for \(i\), \(j\), \(phase\), or as determined by their defaults.

The arguments are:

- **runSeq** An existing NUOPC_RunSequence object.
- **i** Element \(i\) index. This index must be > 0. Corresponds to the Model or Mediator component index if \(j < 0\). Corresponds to src side of a Connector if \(j >= 0\).
- **[j]** Element \(j\) index. Defaults to -1.
- **[phase]** Element phase index. Defaults to 1.
- **rc** Return code; equals ESMF_SUCCESS if there are no errors.

3.8.3 NUOPC_RunElementAddLink - Add a RunElement for a Link to the end of a RunSequence

INTERFACE:

```fortran
subroutine NUOPC_RunElementAddLink(runSeq, slot, rc)
ARGUMENTS:
```

ARGUMENTS:
type(NUOPC_RunSequence), intent(inout), target :: runSeq
integer, intent(in) :: slot
integer, optional, intent(out) :: rc

DESCRIPTION:

Add a new RunElement for a link to the end of an existing RunSequence.
The arguments are:

runSeq  An existing NUOPC_RunSequence object.
slot  Run sequence slot to be linked to. Must be > 0.
rc  Return code; equals ESMF_SUCCESS if there are no errors.

3.8.4  NUOPC_RunElementPrint - Print info about a RunElement object

INTERFACE:

subroutine NUOPC_RunElementPrint(runElement, rc)

ARGUMENTS:

type(NUOPC_RunElement), intent(in) :: runElement
integer, optional, intent(out) :: rc

DESCRIPTION:

Write information about runElement into the default log file.

3.8.5  NUOPC_RunSequenceAdd - Add more RunSequences to a RunSequence vector

INTERFACE:

subroutine NUOPC_RunSequenceAdd(runSeq, addCount, rc)

ARGUMENTS:

type(NUOPC_RunSequence), pointer :: runSeq(:)
integer, intent(in) :: addCount
integer, optional, intent(out) :: rc
DESCRIPTION:

The incoming RunSequence vector runSeq is extended by addCount more RunSequence objects. The existing RunSequence objects are copied to the front of the new vector before the old vector is deallocated.

3.8.6 NUOPC_RunSequenceDeallocate - Deallocate an entire RunSequence vector

INTERFACE:

! Private name; call using NUOPC_RunSequenceDeallocate()
subroutine NUOPC_RunSequenceArrayDeall(runSeq, rc)

ARGUMENTS:

type(NUOPC_RunSequence), pointer :: runSeq(:)
integer, optional, intent(out) :: rc

DESCRIPTION:

Deallocate all of the RunElements in all of the RunSequence defined in the runSeq vector.

3.8.7 NUOPC_RunSequenceDeallocate - Deallocate a single RunSequence object

INTERFACE:

! Private name; call using NUOPC_RunSequenceDeallocate()
subroutine NUOPC_RunSequenceSingleDeall(runSeq, rc)

ARGUMENTS:

type(NUOPC_RunSequence), intent(inout) :: runSeq
integer, optional, intent(out) :: rc

DESCRIPTION:

Deallocate all of the RunElements in the RunSequence defined by runSeq.

3.8.8 NUOPC_RunSequenceIterate - Iterate through a RunSequence

INTERFACE:
function NUOPC_RunSequenceIterate(runSeq, runSeqIndex, runElement, rc)

RETURN VALUE:

logical :: NUOPC_RunSequenceIterate

ARGUMENTS:

type(NUOPC_RunSequence), pointer :: runSeq(:)
integer, intent(in) :: runSeqIndex
type(NUOPC_RunElement), pointer :: runElement
integer, optional, intent(out) :: rc

DESCRIPTION:

Iterate through the RunSequence that is in position runSeqIndex in the runSeq vector. If runElement comes in unassociated, the iteration starts from the beginning. Otherwise this call takes one forward step relative to the incoming runElement, returning the next RunElement in runElement. In either case, the logical function return value is .true. if the end of iteration has not been reached by the forward step, and .false. if the end of iteration has been reached. The returned runElement is only valid for a function return value of .true..

3.8.9 NUOPC_RunSequencePrint - Print info about a single RunSequence object

INTERFACE:

! Private name; call using NUOPC_RunSequencePrint()
subroutine NUOPC_RunSequenceSinglePrint(runSeq, rc)

ARGUMENTS:

type(NUOPC_RunSequence), intent(in) :: runSeq
integer, optional, intent(out) :: rc

DESCRIPTION:

Write information about runSeq into the default log file.

3.8.10 NUOPC_RunSequencePrint - Print info about a RunSequence vector

INTERFACE:

! Private name; call using NUOPC_RunSequencePrint()
subroutine NUOPC_RunSequenceArrayPrint(runSeq, rc)
ARGUMENTS:

```fortran
  type(NUOPC_RunSequence), pointer :: runSeq(:)
  integer, optional, intent(out) :: rc
```

DESCRIPTION:

Write information about the whole runSeq vector into the default log file.

3.8.11 NUOPC_RunSequenceSet - Set values inside a RunSequence object

INTERFACE:

```fortran
  subroutine NUOPC_RunSequenceSet(runSeq, clock, rc)
```

ARGUMENTS:

```fortran
  type(NUOPC_RunSequence), intent(inout) :: runSeq
  type(ESMF_Clock), intent(in) :: clock
  integer, optional, intent(out) :: rc
```

DESCRIPTION:

Set the Clock member in runSeq.
3.9 Utility Routines

3.9.1 NUOPC_ClockCheckSetClock - Check a Clock for compatibility

INTERFACE:

    subroutine NUOPC_ClockCheckSetClock(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:

Compares setClock to checkClock to make sure they match in their current Time. Further ensures that checkClock's timeStep is a multiple of setClock's timeStep. If both these conditions are satisfied then the stop-Time of the setClock is set one checkClock's timeStep ahead of the current Time, taking into account the direction of the Clock.

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.

3.9.2 NUOPC_ClockInitialize - Initialize a new Clock from Clock and stabilityTimeStep

INTERFACE:

    function NUOPC_ClockInitialize(externalClock, stabilityTimeStep, rc)

RETURN VALUE:

    type(ESMF_Clock) :: NUOPC_ClockInitialize

ARGUMENTS:

    type(ESMF_Clock) :: externalClock
    type(ESMF_TimeInterval), intent(in), optional :: stabilityTimeStep
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns a new Clock instance that is a copy of the incoming Clock, but potentially with a smaller timestep. The timestep is chosen so that the timestep of the incoming Clock (externalClock) is a multiple of the new Clock's timestep, and at the same time the new timestep is <= the stabilityTimeStep.
3.9.3 NUOPC_ClockPrintCurrTime - Formatted print of current time

INTERFACE:

```fortran
subroutine NUOPC_ClockPrintCurrTime(clock, string, unit, rc)
```

ARGUMENTS:

```fortran
  type(ESMF_Clock), intent(in) :: clock
  character(*)                :: string
  character(*)                :: unit
  integer                     :: rc
```

DESCRIPTION:

Writes the formatted current time of `clock` to `unit`. Prepends `string` if provided. If `unit` is present it must be an internal unit, i.e. a string variable. If `unit` is not present then the output is written to the default external unit (typically that would be stdout).

3.9.4 NUOPC_ClockPrintStartTime - Formatted print of start time

INTERFACE:

```fortran
subroutine NUOPC_ClockPrintStartTime(clock, string, unit, rc)
```

ARGUMENTS:

```fortran
  type(ESMF_Clock), intent(in) :: clock
  character(*)                :: string
  character(*)                :: unit
  integer                     :: rc
```

DESCRIPTION:

Writes the formatted start time of `clock` to `unit`. Prepends `string` if provided. If `unit` is present it must be an internal unit, i.e. a string variable. If `unit` is not present then the output is written to the default external unit (typically that would be stdout).

3.9.5 NUOPC_ClockPrintStopTime - Formatted print of stop time

INTERFACE:

```fortran
subroutine NUOPC_ClockPrintStopTime(clock, string, unit, rc)
```
ARGUMENTS:

    type(ESMF_Clock), intent(in) :: clock
    character(*), intent(in), optional :: string
    character(*), intent(out), optional :: unit
    integer, intent(out), optional :: rc

DESCRIPTION:

Writes the formatted stop time of clock to unit. Prepends string if provided. If unit is present it must be an internal unit, i.e. a string variable. If unit is not present then the output is written to the default external unit (typically that would be stdout).

3.9.6 NUOPC_CplCompAreServicesSet - Check if SetServices was called

INTERFACE:

    function NUOPC_CplCompAreServicesSet(comp, rc)

RETURN VALUE:

    logical :: NUOPC_CplCompAreServicesSet

ARGUMENTS:

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

DESCRIPTION:

Returns .true. if SetServices has been called for comp. Otherwise returns .false..

3.9.7 NUOPC_CplCompAttributeAdd - Add the NUOPC CplComp Attributes

INTERFACE:

    subroutine NUOPC_CplCompAttributeAdd(comp, rc)

ARGUMENTS:

    type(ESMF_CplComp), intent(inout) :: comp
    integer, intent(out), optional :: rc
DESCRIPTION:

Adds standard NUOPC Attributes to a Coupler Component. Checks the provided importState and exportState arguments for matching Fields and adds the list as "CplList" Attribute.

This adds the standard NUOPC Coupler Attribute package: convention="NUOPC", purpose="General" to the Field. The NUOPC Coupler Attribute package extends the ESG Component Attribute package: convention="ESG", purpose="General".

The arguments are:

comp  The ESMF_CplComp object to which the Attributes are added.

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

3.9.8  NUOPC_CplCompAttributeGet - Get a NUOPC CplComp Attribute

INTERFACE:

subroutine NUOPC_CplCompAttributeGet(comp, cplList, cplListSize, rc)

ARGUMENTS:

  type(ESMF_CplComp), intent(in)   :: comp
  character(*),   intent(out), optional :: cplList(:)
  integer,       intent(out), optional :: cplListSize
  integer,       intent(out), optional :: rc

DESCRIPTION:

Accesses the "CplList" Attribute inside of comp using the convention NUOPC and purpose General. Returns with error if the Attribute is not present or not set.

3.9.9  NUOPC_CplCompAttributeSet - Set the NUOPC CplComp Attributes

INTERFACE:

subroutine NUOPC_CplCompAttributeSet(comp, importState, exportState, rc)

ARGUMENTS:

  type(ESMF_CplComp), intent(inout) :: comp
  type(ESMF_State),    intent(in)   :: importState
  type(ESMF_State),    intent(in)   :: exportState
  integer,            intent(out), optional :: rc
DESCRIPTION:

Checks the provided importState and exportState arguments for matching Fields and sets the coupling list as "CplList" Attribute in comp.

The arguments are:

comp  The ESMF_CplComp object to which the Attributes are set.
importState  Import State.
exportState  Export State.
[rc]  Return code; equals ESMF_SUCCESS if there are no errors.

3.9.10  NUOPC_FieldAttributeAdd - Add the NUOPC Field Attributes

INTERFACE:

    subroutine NUOPC_FieldAttributeAdd(field, StandardName, Units, LongName, & ShortName, Connected, rc)

ARGUMENTS:

    type(ESMF_Field) :: field
    character(*) , intent(in) :: StandardName
    character(*) , intent(in), optional :: Units
    character(*) , intent(in), optional :: LongName
    character(*) , intent(in), optional :: ShortName
    character(*) , intent(in), optional :: Connected
    integer , intent(out), optional :: rc

DESCRIPTION:

Adds standard NUOPC Attributes to a Field object. Checks the provided arguments against the NUOPC Field Dictionary. Omitted optional information is filled in using defaults out of the NUOPC Field Dictionary.

This adds the standard NUOPC Field Attribute package: convention="NUOPC", purpose="General" to the Field. The NUOPC Field Attribute package extends the ESG Field Attribute package: convention="ESG", purpose="General".

The arguments are:

field  The ESMF_Field object to which the Attributes are added.
StandardName  The StandardName of the Field. Must be a StandardName found in the NUOPC Field Dictionary.
[Units]  The Units of the Field. Must be convertible to the canonical units specified in the NUOPC Field Dictionary for the specified StandardName. If omitted, the default is to use the canonical units associated with the StandardName in the NUOPC Field Dictionary.
[LongName]  The LongName of the Field. NUOPC does not restrict the value of this variable. If omitted, the default is to use the LongName associated with the StandardName in the NUOPC Field Dictionary.
[ShortName]  The ShortName of the Field. NUOPC does not restrict the value of this variable. If omitted, the default is to use the ShortName associated with the StandardName in the NUOPC Field Dictionary.

[Connected]  The connection status of the Field. Must be one of the NUOPC supported values: false or true. If omitted, the default is a connected status of false.

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

3.9.11  NUOPC_FieldAttributeGet - Get a NUOPC Field Attribute

INTERFACE:

    subroutine NUOPC_FieldAttributeGet(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:

Accesses the Attribute name inside of field using the convention NUOPC and purpose General. Returns with error if the Attribute is not present or not set.

3.9.12  NUOPC_FieldAttributeSet - Set a NUOPC Field Attribute

INTERFACE:

    subroutine NUOPC_FieldAttributeSet(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 General.
3.9.13 NUOPC_FieldBundleUpdateTime - Update the time stamp on all Fields in a FieldBundle

INTERFACE:

subroutine NUOPC_FieldBundleUpdateTime(srcFields, dstFields, rc)

ARGUMENTS:

type(ESMF_FieldBundle), intent(in) :: srcFields
type(ESMF_FieldBundle), intent(inout) :: dstFields
integer, intent(out), optional :: rc

DESCRIPTION:

Updates the time stamp on all Fields in the dstFields FieldBundle to be the same as in the dstFields FieldBundle.

3.9.14 NUOPC_FieldDictionaryAddEntry - Add an entry to the NUOPC Field dictionary

INTERFACE:

subroutine NUOPC_FieldDictionaryAddEntry(standardName, canonicalUnits, &
defaultLongName, defaultShortName, rc)

ARGUMENTS:

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

DESCRIPTION:

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

3.9.15 NUOPC_FieldDictionaryGetEntry - Get information about a NUOPC Field dictionary entry

INTERFACE:

subroutine NUOPC_FieldDictionaryGetEntry(standardName, canonicalUnits, &
defaultLongName, defaultShortName, rc)
ARGUMENTS:

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

DESCRIPTION:

Returns the canonical units, the default LongName and the default ShortName that the NUOPC Field dictionary associates with a StandardName.

3.9.16 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:

Returns .true. if the NUOPC Field dictionary has an entry with the specified StandardName, .false. otherwise.

3.9.17 NUOPC_FieldDictionarySetup - Setup the NUOPC Field dictionary

INTERFACE:

subroutine NUOPC_FieldDictionarySetup(rc)

ARGUMENTS:

integer, intent(out), optional :: rc

DESCRIPTION:

Setup the NUOPC Field dictionary.
3.9.18 NUOPC_FieldIsAtTime - Check if the Field is at the given Time

INTERFACE:

    function NUOPC_FieldIsAtTime(field, time, rc)

RETURN VALUE:

    logical :: NUOPC_FieldIsAtTime

ARGUMENTS:

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

DESCRIPTION:

Returns .true. if the Field has a timestamp that matches time. Otherwise returns .false.

3.9.19 NUOPC_FillCplList - Fill the cplList according to matching Fields

INTERFACE:

    subroutine NUOPC_FillCplList(importState, exportState, cplList, rc)

ARGUMENTS:

    type(ESMF_State), intent(in) :: importState
    type(ESMF_State), intent(in) :: exportState
    character(ESMF_MAXSTR), pointer :: cplList(:)
    integer, intent(out), optional :: rc

DESCRIPTION:

Constructs a list of matching StandardNames of Fields in the importState and exportState. Returns this list in cplList.

The pointer argument cplList must enter this method unassociated. On return, the deallocation of the potentially associated pointer becomes the caller’s responsibility.

3.9.20 NUOPC_GridCompAreServicesSet - Check if SetServices was called

INTERFACE:
function NUOPC_GridCompAreServicesSet(comp, rc)

RETURN VALUE:

logical :: NUOPC_GridCompAreServicesSet

ARGUMENTS:

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

DESCRIPTION:

Returns .true. if SetServices has been called for comp. Otherwise returns .false..

3.9.21 NUOPC_GridCompAttributeAdd - Add the NUOPC GridComp Attributes

INTERFACE:

subroutine NUOPC_GridCompAttributeAdd(comp, rc)

ARGUMENTS:

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

DESCRIPTION:

Adds standard NUOPC Attributes to a Gridded Component. This adds the standard NUOPC GridComp Attribute package: convention="NUOPC", purpose="General" to the Gridded Component. The NUOPC GridComp Attribute package extends the CIM Component Attribute package: convention="CIM 1.5", purpose="ModelComp".

3.9.22 NUOPC_GridCompCheckSetClock - Check Clock compatibility and set stopTime

INTERFACE:

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:

Compares `externalClock` to the Component internal Clock to make sure they match in their current Time. Further ensures that the external Clock’s `timeStep` is a multiple of the internal Clock’s `timeStep`. If both these conditions are satisfied then the `stopTime` of the internal Clock is set to be reachable in one `timeStep` of the external Clock, taking into account the direction of the Clock.

3.9.23 NUOPC_GridCompSetClock - Initialize and set the internal Clock of a GridComp

INTERFACE:

```fortran
subroutine NUOPC_GridCompSetClock(comp, externalClock, stabilityTimeStep, &
rc)
```

ARGUMENTS:

```fortran
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:

Sets 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 ensures that the `timeStep` of the external Clock is a multiple of the internal Clock’s `timeStep`. If the `stabilityTimeStep` argument is not provided then the internal Clock will simply be set as a copy of the `externalClock`.

3.9.24 NUOPC_GridCompSetServices - Try to find and call SetServices in a shared object

INTERFACE:

```fortran
recursive subroutine NUOPC_GridCompSetServices(comp, sharedObj, userRc, rc)
```

ARGUMENTS:

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

DESCRIPTION:
Try to find a routine called "SetServices" in the sharedObj and execute it to set the component’s services. An attempt is made to find a routine that is close in name to "SetServices", allowing compiler name mangling, i.e. upper and lower case, as well as trailing underscores.

3.9.25 NUOPC_GridCreateSimpleXY - Create a simple XY cartesian Grid

INTERFACE:

    function NUOPC_GridCreateSimpleXY(x_min, y_min, x_max, y_max, &
            i_count, j_count, rc)

RETURN VALUE:

    type(ESMF_Grid):: NUOPC_GridCreateSimpleXY

ARGUMENTS:

    real(ESMF_KIND_R8), intent(in) :: x_min, x_max, y_min, y_max
    integer, intent(in) :: i_count, j_count
    integer, intent(out), optional :: rc

DESCRIPTION:

Creates and returns a very simple XY cartesian Grid.

3.9.26 NUOPC_IsCreated - Check whether an ESMF object has been created

INTERFACE:

    ! call using generic interface: NUOPC_IsCreated
    function NUOPC_ClockIsCreated(clock, rc)

RETURN VALUE:

    logical :: NUOPC_ClockIsCreated

ARGUMENTS:

    type(ESMF_Clock) :: clock
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns .true. if the ESMF object (here clock) is in the created state, .false. otherwise.
3.9.27 NUOPC_StateAdvertiseField - Advertise a Field in a State

INTERFACE:

```fortran
subroutine NUOPC_StateAdvertiseField(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:

Advertises a Field in a State. This call checks the provided information against the NUOPC Field Dictionary. Omitted optional information is filled in using defaults out of the NUOPC Field Dictionary.

The arguments are:

state The ESMF_State object through which the Field is advertised.

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

[Units] The Units of the advertised Field. Must be convertible to the canonical units specified in the NUOPC Field Dictionary for the specified StandardName. If omitted, the default is to use the canonical units associated with the StandardName in the NUOPC Field Dictionary.

[LongName] The LongName of the advertised Field. NUOPC does not restrict the value of this variable. If omitted, the default is to use the LongName associated with the StandardName in the NUOPC Field Dictionary.

[ShortName] The ShortName of the advertised Field. NUOPC does not restrict the value of this variable. If omitted, the default is to use the ShortName associated with the StandardName in the NUOPC Field Dictionary.

[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 transfer offer for the geom object (Grid, Mesh, LocStream, XGrid) associated with the advertised Field. NUOPC controls the vocabulary of this attribute: "will provide", "can provide", "cannot provide". If omitted, the default is "will provide".

[rc] Return code; equals ESMF_SUCCESS if there are no errors.
3.9.28 NUOPC_StateAdvertiseFields - Advertise Fields in a State

INTERFACE:

    subroutine NUOPC_StateAdvertiseFields(state, StandardNames, rc)

ARGUMENTS:

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

DESCRIPTION:

Advertises Fields in a State. Defaults are set according to the NUOPC Field Dictionary.

The arguments are:

state  The ESMF_State object through which the Field is advertised.
StandardNames A list of StandardNames of the advertised Fields. Must be StandardNames found in the NUOPC Field Dictionary.
[rc] Return code; equals ESMF_SUCCESS if there are no errors.

3.9.29 NUOPC_StateBuildStdList - Build lists of Field information from a State

INTERFACE:

    recursive subroutine NUOPC_StateBuildStdList(state, stdAttrNameList, &
        stdItemNameList, stdConnectedList, stdFieldList, rc)

ARGUMENTS:

    type(ESMF_State), intent(in) :: state
    character(ESMF_MAXSTR), pointer :: stdAttrNameList(:)
    character(ESMF_MAXSTR), pointer, optional :: stdItemNameList(:)
    character(ESMF_MAXSTR), pointer, optional :: stdConnectedList(:)
    type(ESMF_Field), pointer, optional :: stdFieldList(:)
    integer, intent(out), optional :: rc

DESCRIPTION:

Constructs lists containing the StandardName, Field name, and connected status of the Fields in the state. Returns this information in the list arguments. Recursively parses through nested States.

All pointer arguments present must enter this method unassociated. On return, the deallocation of an associated pointer becomes the user responsibility.
3.9.30  NUOPC_StateIsAllConnected - Check if all the Fields in a State are connected

INTERFACE:

    function NUOPC_StateIsAllConnected(state, rc)

RETURN VALUE:

    logical :: NUOPC_StateIsAllConnected

ARGUMENTS:

    type(ESMF_State), intent(in) :: state
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns \texttt{.true.} if all the Fields in \texttt{state} are connected. Otherwise returns \texttt{.false.}.

3.9.31  NUOPC_StateIsAtTime - Check if all the Fields in a State are at the given Time

INTERFACE:

    function NUOPC_StateIsAtTime(state, time, rc)

RETURN VALUE:

    logical :: NUOPC_StateIsAtTime

ARGUMENTS:

    type(ESMF_State), intent(in) :: state
    type(ESMF_Time), intent(in) :: time
    integer, intent(out), optional :: rc

DESCRIPTION:

Returns \texttt{.true.} if all the Fields in \texttt{state} have a timestamp that matches \texttt{time}. Otherwise returns \texttt{.false.}.

3.9.32  NUOPC_StateIsFieldConnected - Test if Field in a State is connected

INTERFACE:
function NUOPC_StateIsFieldConnected(state, fieldName, rc)

RETURN VALUE:

logical :: NUOPC_StateIsFieldConnected

ARGUMENTS:

type(ESMF_State), intent(in) :: state
class(*), intent(in) :: fieldName
integer, intent(out), optional :: rc

DESCRIPTION:

Returns .true. if Fields with name fieldName contained in state is connected. Otherwise returns .false..
ARGUMENTS:

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

DESCRIPTION:

Realizes a previously advertised Field in state.

3.9.35  NUOPC_StateSetTimestamp - Set a time stamp on all Fields in a State

INTERFACE:

```plaintext```
subroutine NUOPC_StateSetTimestamp(state, clock, selective, rc)
```

ARGUMENTS:

```plaintext
type(ESMF_State), intent(inout) :: state
type(ESMF_Clock), intent(in) :: clock
logical, intent(in), optional :: selective
integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the TimeStamp Attribute according to clock on all the Fields in state.

3.9.36  NUOPC_StateUpdateTimestamp - Update the timestamp on all the Fields in a State

INTERFACE:

```plaintext```
subroutine NUOPC_StateUpdateTimestamp(state, rootPet, rc)
```

ARGUMENTS:

```plaintext
type(ESMF_State), intent(in) :: state
integer, intent(in) :: rootPet
integer, intent(out), optional :: rc
```

DESCRIPTION:

Updates the TimeStamp Attribute for all the Fields on all the PETs in the current VM to the TimeStamp Attribute held by the Field instance on the rootPet.
3.9.37  NUOPC_TimePrint - Formatted print of time information

INTERFACE:

    subroutine NUOPC_TimePrint(time, string, unit, rc)

ARGUMENTS:

    type(ESMF_Time), intent(in) :: time
    character(*),      intent(in), optional :: string
    character(*),      intent(out), optional :: unit
    integer,           intent(out), optional :: rc

DESCRIPTION:

Write a formatted time with or without string to unit. If unit is present it must be an
internal unit, i.e. a string variable. If unit is not present then the output is written to the
default external unit (typically that would be stdout).
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:

```bash
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_DEPToFront)
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:

```bash
.SUFFIXES: .f90 .F90 .c .C
%.o : %.f90
  $(ESMF_F90COMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \ $(ESMF_F90COMPITOPTS) $(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_CXXCOMPILEPATHS) $(ESMF_CXXCOMPILEPATHSLOCAL) $(ESMF_CXXCOMPILEPATHS) \ $(ESMF_CXXCOMPILECPPFLAGS) $<
%.o : %.C
  $(ESMF_CXXCOMPILER) -c $(DEP_FRONTS) $(DEP_INCS) \ $(ESMF_CXXCOMPILEOPTS) $(ESMF_CXXCOMPILEPATHS) $(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 $(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:

```makefile
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".

```makefile
.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:

```makefile
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".

```makefile
.PRECIOUS: %.so
%.mk : %.so
    @echo "# ESMF self-describing build dependency makefile fragment" > $@
    @echo "# ESMF self-describing build dependency makefile fragment" >> $@
    @echo $0
```

57
@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.

```makefile
include abc.mk
ifeq (,$(findstring .so,$(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 by 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 "%ESM_%DEP_FRONT = ABC" >> $@
@echo "%ESM_%DEP_INCPATH = `pwd`" >> $@
@echo "%ESM_%DEP_CMPL_OBJS = `pwd`/$<" >> $@
@echo "%ESM_%DEP_LINK_OBJS = `pwd`/$< $(DEP_LINK_OBJS) >> $@
@echo "%ESM_%DEP_SHRD_PATH = " $(DEP_SHRD_PATH) >> $@
@echo "%ESM_%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_FRONT :=$(DEP_FRONT) -DFRONT_XXX=$(ESM_%DEP_FRONT)
DEP_INCS :=$(DEP_INCS) $(addprefix -I, $(ESM_%DEP_INCPATH))
DEP_CMPL_OBJS :=$(DEP_CMPL_OBJS) $(ESM_%DEP_CMPL_OBJS)
DEP_LINK_OBJS :=$(DEP_LINK_OBJS) $(ESM_%DEP_LINK_OBJS)
DEP_SHRD_PATH :=$(DEP_SHRD_PATH) $(addprefix -L, $(ESM_%DEP_SHRD_PATH)) \ $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESM_%DEP_SHRD_PATH))
DEP_SHRD_LIBS :=$(DEP_SHRD_LIBS) $(addprefix -l, $(ESM_%DEP_SHRD_LIBS))

include yyy.mk
DEP_FRONT :=$(DEP_FRONT) -DFRONT_YYY=$(ESM_%DEP_FRONT)
DEP_INCS :=$(DEP_INCS) $(addprefix -I, $(ESM_%DEP_INCPATH))
DEP_CMPL_OBJS :=$(DEP_CMPL_OBJS) $(ESM_%DEP_CMPL_OBJS)
DEP_LINK_OBJS :=$(DEP_LINK_OBJS) $(ESM_%DEP_LINK_OBJS)
DEP_SHRD_PATH :=$(DEP_SHRD_PATH) $(addprefix -L, $(ESM_%DEP_SHRD_PATH)) \ $(addprefix -Wl$(COMMA)-rpath$(COMMA), $(ESM_%DEP_SHRD_PATH))
DEP_SHRD_LIBS :=$(DEP_SHRD_LIBS) $(addprefix -l, $(ESM_%DEP_SHRD_LIBS))

.PRECIOUS: %.o
%.mk : %.o
@echo "# ESMF self-describing build dependency makefile fragment" > $@
@echo >> $@
@echo "%ESM_%DEP_FRONT = ABC" >> $@
@echo "%ESM_%DEP_INCPATH = `pwd`" >> $@
@echo "%ESM_%DEP_CMPL_OBJS = `pwd`/$<" >> $@
@echo "%ESM_%DEP_LINK_OBJS = `pwd`/$< $(DEP_LINK_OBJS) >> $@
@echo "%ESM_%DEP_SHRD_PATH = " $(DEP_SHRD_PATH) >> $@
@echo "%ESM_%DEP_SHRD_LIBS = " $(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:

```bash
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:

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

# else
!! --------- Fortran block ---------
interface
  subroutine setservices_abc(gcomp, rc)
    use ESMF
```
type(ESMF_GridComp) :: gcomp
integer, intent(out) :: rc
end subroutine
end interface

#endif

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

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

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

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

```bash
.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:

```bash
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 through the ESMF component interface, and analyses 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:
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 a script is available that can be used to compile and link the explorer application specifically against a specified component. This part of the explorer leverages the standardized component dependencies discussed in section 4. This step is executed by providing the component’s mk-file to the explorer script:

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

Any issues found during this step are reported. The successful completion of this step will produce an executable called nuopcExplorerApp.

The second part of the Component Explorer is the explorer application. It can either be built using the explorer script as outlined above, or by using the makefile directly:

```
make nuopcExplorerApp
```

In the second case, the resulting nuopcExplorerApp will not be tied to a specific component, but expects a component in form of a shared object to be specified 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 a sufficient number of processes need to be specified. In terms of the common mpirun tool launching nuopcExplorerApp may look like this:

```
mpirun -np X ./nuopcExplorerApp
```
for an executable that was built against a specific component. Or like this

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

for an executable that expects a the component in form of a shared object. In both cases the output of the
nuopcExplorerApp will report what it finds during the interaction with the component. The output 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.