FORCE

Release 0.5.0

unknown

Sep 28, 2020

CONTENTS

This site contains documentation for the Business Decision Support System (BDSS), implemented under the Formulations and Computational Engineering (FORCE) project within Horizon 2020 (NMBP-23-2016/721027).

CHAPTER

ONE

USER MANUAL

1.1 Introduction

The Business Decision System is the CLI support for the evaluation of Pareto front computations. It is a single executable `force_bdss` that interprets a workflow specification file, normally generated via the GUI workflow manager.

By itself, the executable and the code implementing it provides no functionality. All functionality comes from external plugins, extending the API to provide new entities, specifically:

- Multi Criteria Optimizer (MCO)
- DataSources, which can be a simulator or just a database
- Notification Listeners, like a remote database which retrieve data during the computation
- UI Hooks, which permit to define additional operations which will be executed at specific moments in the UI lifetime (before and after execution of the BDSS, before saving the workflow)

Plugin support requires compliance to the Force BDSS api for plugins. Extensions are registered via setuptools entry points.

Execution of the force bdss executable is simple. Invoke with:

```
force_bdss workflow.json
```

1.2 Installation

The BDSS, the Workflow Manager and all plugins can be cloned from the Force 2020 github respositories. For the BDSS and Workflow Manager,

```
git clone https://github.com/force-h2020/force-bdss
git clone https://github.com/force-h2020/force-wfmanager
```

1.2.1 Enthought Deployment Manager

The BDSS, the Workflow Manager and plugins must be installed through the Enthought Deployment Manager (EDM), a python virtual environment and package manager. For new users it is worth examining EDM's documentation.

To install EDM, follow the instructions specific to your operating system ,here.

1.2.2 The Bootstrap Environment

Once EDM is installed create a 'bootstrap' environment from which you can install the BDSS, Workflow Manager and plugins,

```
edm install -e bootstrap -y click setuptools
```

Note that 'bootstrap' can be replaced by any name to the same effect. Now you can enter `bootstrap` with,

```
edm shell -e bootstrap
```

and your shell prompt is prefixed with `(bootstrap)`.

1.2.3 The BDSS Runtime Environment

Although repositories (BDSS, etc) are installed *from* the `bootstrap` environment, they are installed *into* a separate environment, within which the BDSS and the Workflow Manager will actually run. Thus this environment has also to be created before installation. To do this first cd into the cloned force-bdss respository,

```
~/Force-Project (bootstrap)$ cd force-bdss
```

and then,

```
~/Force-Project/force-bdss (bootstrap)$ python -m ci build-env
```

This creates a environment called `force-pyXX`, where `XX` refers to the python version that the environment runs (e.g. `force-py36` for python 3.6) . You will now see it in the list of EDM environments,

```
(bootstrap)$ edm environments list

>> * bootstrap     cpython  3.6.9+2  win_x86_64  msvc2015  ~\.edm\envs\bootstrap
>>   force-py36    cpython  3.6.9+2  win_x86_64  msvc2015  ~.edm\envs\force-pyXX
```

1.2.4 Repository Installation

From the `bootstrap` environment (not `force-pyXX`!), for each respository in turn, cd into its directory and then install it with `python -m ci install`. i.e.,

```
~/Force-Project/force-bdss (bootstrap)$ python -m ci install

~/Force-Project/force-bdss (bootstrap)$ cd ../force-wfmanager
~/Force-Project/force-wfmanager (bootstrap)$ python -m ci install

...etc
```

CHAPTER

TWO

DEVELOPER MANUAL

2.1 FORCE BDSS Design

The BDSS is an Envisage/Task application. it uses tasks to manage the plugin system, with stevedore to manage the additions.

2.1.1 Application

The main class is the `BDSSApplication`, an `envisage.Application` subclass, which is in charge of loading the plugins, and also adding the relevant core plugins to make the whole system run. Specifically it loads:

- The `FactoryRegistryPlugin`, which is where all external plugins will put their classes.
- **Depending on the –evaluate switch, a relevant execution plugin:**
  - `OptimizeOperation`: Invokes the MCO.
  - `EvaluateOperation`: performs a single point evaluation, that is, executes the pipeline only once.

Note: the design requiring the `--evaluate` switch assumed a "Dakota" model of execution (external process controlled by Dakota). In the current Enthought Example plugin we use both the `--evaluate` strategy and direct control, where all the calculation is performed without spawning additional processes other than the initial `force_bdss`.

The `BDSSApplication` contains the following structure:

At the application level, there are three main attributes of type:

- `WorkflowFile`: contains the `Workflow` object, as well as the ability to read, write from file
- `IOperation`: determines the operation that will be performed by the BDSS
- `IFactoryRegistry`: contains references to all `IFactory` classes that are contributed by currently installed plugins. This object is constructed from by the `BDSSApplication`, using the Envisage plugins installed in the local environment, but is actually used by `WorkflowReader` to instantiate serialised `Workflow` objects from file.

Upon start up, the `BDSSApplication` performs the following process:

2.1.2 Operations

The `IOperation` interface defines the requirements for an operation that can be performed by the BDSS. The `BaseOperation` class provides this interface and contains useful routines that are used to set up both internal and external communication routes. Currently we have 2 subclasses of this base class: `EvaluateOperation` and `OptimizeOperation`.

The `OptimizeOperation` (the default operation performed by the `force_bdss`) is designed to work alongside a `BaseMCO` subclass, that defines an optimization method to be performed on a `Workflow` instance.

The `EvaluateOperation` (invoked by using the `--evaluate` flag with the `force_bdss` command line application) is designed to work alongside a `BaseMCOCommunicator` subclass that determines how to send and receive MCO parameters and KPIs.

2.1.3 Class Diagrams

Factory Classes

Factories are used in the BDSS to modularise the code and therefore allow additional components to be contributed using plugins.

The `BaseFactory` class fulfills an `IFactory` interface, and is therefore able to be contributed and subsequently located by the `BaseExtensionPlugin` and `FactoryRegistryPlugin` classes respectively

Each individual `BaseFactory` subclass also provides an interface that inherits from `IFactory`.

Model Classes

Each `BaseModel` class is designed to act as both a serializable and visual representation of a `Workflow` object. It contains any information that is exposed to the user and, since it inherits from `HasTraits`, the UI components are provided by the TraitsUI library.

The `BaseModel` classes all share a common API for serialization, event handling and workflow verification:

- `notify(BaseDriverEvent)`: Takes an `BaseDriverEvent` instance and assigns it to the `event` attribute.
- `verify()`: Performs a set of checks on the model attributes and returns a list of `VerifierError` instances describing any issues.
- `serialize()`: Returns a dictionary object containing basic python values able to be written to file as a JSON.

The `BaseModel.verify` and `BaseModel.serialize` methods are expected to be overridden by any subclass to suit the needs of the developer, whereas the `BaseModel.notify` method is designed to be used as a setter method for the `event` attribute. Further information is available on both event handling and verification pathways in the BDSS.

Runtime Classes

The factory classes act as creators of `BaseModel` instances, as well as other objects that are only used during an MCO run. The following sections take a closer look at each of these objects in turn

MCO

The `BaseMCOFactory` fulfills the `IMCOFactory` interface. It is able to construct both `BaseMCO` and `BaseMCOModel` subclasses and also contains references to a list of objects that fulfill the `IMCOParameterFactory` interface. Likewise, the `BaseMCOParameterFactory` provides this interface and constructs `BaseMCOParameter` subclasses. Consequently, each MCO must declare a set of parameter types that it is able to use.

The `BaseMCOModel` class provides user input required by a `BaseMCO` class during runtime. It also contains references to, and methods used to broadcast the MCO-related `BaseDriverEvent` subclasses: `MCOStartEvent`, `MCOProgressEvent` and `MCOFinishEvent`. The variables designated as KPIs are represented by a list of `KPISpecification` objects, which also provide scaling instructions in order to convert them into comparable unit-less values. The `BaseMCO` class' only job is to provide the the implementation of any optimization algorithm that will be performed during the `BaseMCO.run` method. This method takes in one argument, which must fulfill the `IEvaluator` interface.

Any object that provides this interface must contain a reference to the corresponding `BaseMCOModel`, as well as an implementation of an `IEvaluator.evaluate` method which is used to return the state of the system to be optimized for a given list of input parameters. It is therefore expected that the `BaseMCO.run` method will perform some iteration over `IEvaluate.evaluate`, passing in possible MCO parameter values and obtaining the corresponding KPI values as output.

Data Sources

The `BaseDataSourceFactory` fulfills the `IDataSourceFactory` interface. It is able to construct both `BaseDataSource` and `BaseDataSourceModel` subclasses.

The `BaseDataSourceModel` class provides user input required by a `BaseDataSource` class during runtime. It is also used in the backend to broadcast any Data Source-related events: `DataSourceStartEvent` and `DataSourceProgressEvent` and `DataSourceFinishEvent`.

The `BaseDataSource` class has methods that require implementation: `BaseDataSource.run` and `BaseDataSource.slots` The `run` method contains a black box of code that will be performed during evaluation of the workflow. It expects a reference to the `BaseDataSourceModel` as well as a list of `DataValue` instances that contain extra parameters generated during runtime. The `slots` method is used to communicate the expected length and format of both the input and output lists of `DataValues` associated with the `run` method, mainly so that this can be displayed in the UI before an MCO run is called.

Notification Listeners

The `BaseNotificationListenerFactory` fulfills the `INotificationListenerFactory` interface. It is able to construct both `BaseNotificationListener` and `BaseNotificationListenerModel` subclasses

The `BaseNotificationListenerModel` class provides user input required by a `BaseNotificationListener` class during runtime.

2.1.4 Workflow

The `Workflow` contains a full description of the system to be optimised, the optimisation procedure and any listener objects that are expected to react on events that are created during runtime.

Workflow Specification

The "Workflow" object of the application needs to contain a representation of three entities:

- Multi Criteria Optimizer (MCO)
- Data Sources
- Notification Listeners

Additionally, messages can be sent and received between the CLI `force_bdss` application and the `force_wfmanager` GUI via a subset of notification listener, known as a 'UI Hook'.

There are a few core assumptions about each of these entities:

- The MCO provides numerical values and injects them into a pipeline made of multiple layers. Each layer is composed of multiple Data Sources. The MCO can execute this pipeline directly, or indirectly by invoking the `force_bdss` with the option `--evaluate`. This invocation will produce, given a vector in the input space, a vector in the output space.
- The Data Sources are entities that are arranged in layers. Each Data Source has inputs and outputs, called slots. These slots may depend on the configuration options of the specific data source. The pipeline is created by binding data sources outputs on the layers above with the data sources inputs of a given layer. Each output can be designated as a KPI and thus be transmitted back to the MCO for optimization.
- A Notification Listener listens to the state of the MCO (i.e. Started/New step of the computation/Finished) and can perform tasks accordingly. For example, it could connect to a remote database which is filled with the MCO results during the computation (e.g. the GUI `force_wfmanager` has a notification listener in order to fill the result table).
- UI Hooks permit to define additional operations which will be executed at specific moments in the UI lifetime (before and after execution of the BDSS, before saving the workflow). Those operations won't be executed by the command line interface of the BDSS.

`Workflow` Class

The specifications above can be performed by the `Workflow` class, a concrete implementation of a pipeline for the BDSS:

- The `Workflow.mco_model` attribute refers to a `BaseMCOModel` instance that defines user-inputted parameters for the MCO. It contains a list of `BaseMCOParameter` instances that define the search space, as well as a list of `KPISpecification` instances referring to variables that will be optimised. It also defined the types of `MCOStartEvent`, `MCOProgressEvent` and `MCOFinishEvent` classes that will be emitted during the MCO run time.
- The `Workflow.execution_layers` attribute contains a list of `ExecutionLayer` instances that represent each Data Source layer. Within each element, the `ExecutionLayer.data_sources` attribute contains a list of `BaseDataSourceModel` instances that define the input and output slots for each Data Source. The `ExecutionLayer.execute_layer` method can be called to calculate all `DataValue` objects that are generated by that layer.
- The `Workflow.notification_listeners` attribute contains a list of `BaseNotificationListener` instances that define user-inputted parameters for each notification listener that will be active during the MCO run.

During an Force BDSS run, the `Workflow` object is used to initialise the MCO and also perform each calculation of the system state for a given set of input parameter values. This is carried out by the following steps:

1. Pass the `Workflow` instance into the `BaseMCO.run` method as the `evaluator` argument.
2. For every iteration in the MCO method, `Workflow.evaluate` method can then be invoked for a list of values referring to the state of each `BaseMCOParameter`. The return values represent the corresponding state of each `KPISpecification`.
3. Both parameter and KPI values can then be broadcast to any `BaseNotificationListener` instances active using the `Workflow.mco_model.notify_progress_event` method.
4. Iterate steps 2 and 3 until all evaluations have been performed and the MCO has finished.

Workflow JSON Files

A `Workflow` object can be instantiated from an appropriately formatted workflow JSON file. Typically the structure of this JSON represents a serialised version of each object contained within the `Workflow`. Currently the `WorkflowReader` supports two file versions: 1 and 1.1. There are only minor differences between both versions:

1. `Workflow.mco_model` attribute data stored under `mco` key in version 1 vs `mco_model` key in 1.1
2. `Workflow.execution_layers` attribute data represented as a list of lists in version 1 vs a list of dictionaries in version 1.1. In version 1, each element in the outer list implicitly represents an execution layer, whilst each element in the the inner list represents the serialised status of a `DataSourceModel` instance. In version 1.1, we explicitly include the status of each `ExecutionLayer` instance in the outer list, and therefore each dictionary element is also expected to contain a `data_sources` key with a list of `DataSourceModel` statuses.

The `WorkflowWriter` will produce JSON files that conform to the latest available version (currently 1.1) by default.

2.1.5 Event Handling

The `BaseDriverEvent` class is the main object used to relay messages internally between components of the `BDSSApplication`. It can also be serialized in order to be passed between external programs (i.e. the `force_wfmanager`) as a JSON.

Any events that are created during runtime are propagated through the `Workflow` object hierarchy, up to the `BaseOperation` class that is being performed by the `BDSSApplication`, before finally being broadcast to all `BaseNotificationListener` classes present. Consequently, the `BaseModel` class contains an `event` attribute, as well as a `notify` method that is used to set it. Listeners can then detect changes in any `BaseModel.event` subclass, and process the new event accordingly.

Any actions that are required to be performed upon notification of a specific `BaseDriverEvent` subclass are handled by the `BaseNotificationListener.deliver` method.

In this way, we can also use `BaseDriverEvents` to control the progress of an MCO run, since after every broadcast, the `OptimizeOperation` checks its run time status to see whether the event has triggered a MCO 'pause' or 'stop' signal.

Additionally, the `BaseDataSource._run` method shadows the `BaseDataSource.run` method in order to signal the beginning and end of a data source execution. By doing so, we are able to pause and stop and MCO run between each `run` method invocation, which represents a black box operation.

2.1.6 Verification

Any `Workflow` instance can be verified by invoking the `verify_workflow` function. This returns a list of `VerifierError` instances, generated by iterating through the `Workflow` hierarchy and calling each bound `verify` method in turn.

In the `BDSSApplication`, this function is called during the beginning of both `OptimizeOperation.run` and `EvaluateOperation.run` methods. If any `VerifierError` instances are returned, then an `Exception` is raised and the program will terminate. Additionally, since the `Workflow` can be verified prior to the `force_bdss` runtime, the `force_wfmanager` GUI is able to report any errors back to the user as the `Workflow` is being constructed.

2.1.7 Package Structure

As well as a command line program, the BDSS also comes with a `force_bdss` package containing objects required by plugin developers. These should be publicly accessed through the `force_bdss.api` module, but a brief explanation of the internal structure is provided below.

The `data_sources`, `mco`, `notification_listeners` and `ui_hooks` packages, and the `base_extension_plugin` class, contain all the base classes that plugin developers need to use in order to write a plugin. They have been coded to be as error tolerant as possible, and deliver robust error messages as much as possible. Further information regarding these classes can be found in the plugin development documentation

The `io` package contains the reader and writer for the model. It simply serializes the model objects and dumps them to JSON, or vice-versa. Note that the reader requires the factory registry, because you can't load entities from the file if you don't have the appropriate plugin, as only the plugin knows the model structure and can therefore take the JSON content and apply it to the model object.

The `core_plugins` contains fundamental plugins that are considered part of a "standard library", providing common data sources, MCOs and other relevant objects.

Finally, `core` contains:

- base classes for a few entities that are reused for the plugins.
- the `DataValue` entity. This is the "exchange entity" between data sources. It is a value that also contains the type, the accuracy, and so on. It can refer to anything: a float, an array, a string, etc.
- `Workflow` model object, representing the entire state of the BDSS.
- `input/output_slot_info` contain the _bound_ information for slots. A `DataSource` provides slots (see slot module) but these are not bound to a specific "variable name". The `SlotInfo` classes provide this binding.
- `execution_layer` contains the `ExecutionLayer` class, which provides the actual machinery that runs the pipeline.
- `verifier` contains a verification function that checks if the workflow can run or has errors.

2.1.8 Future directions

The future design will probably need to address the following:

- Check if the `--evaluate` strategy and design is still relevant. More MCOs are needed for reasonable conclusions.
- IWM is going to provide a strict description of types (`osp-core`, previously known as `simphony`). Currently, all type entries in the e.g. slots are simple strings as a workaround. This is supposed to change once IWM provides a comprehensive set of types.
- The project is now at a stage where plugins can be developed, and real evaluations can be performed. We can solve the current toy cases, but real cases and UI requirements may promote the need for additional requirements.

2.2 Plugin Development

Force BDSS is extensible through plugins. A plugin can be (and generally is) provided as a separate python package that makes available some new classes. Force BDSS will find these classes from the plugin at startup.

A single Plugin can provide one or more of the following entities: `MCO`, `DataSources`, `NotificationListeners`, `UIHooks`. It can optionally provide `DataViews` to be used by the `force_wfmanager` GUI.

An example plugin implementation is available at:

https://github.com/force-h2020/force-bdss-plugin-enthought-example

To implement a new plugin, you must define at least four classes:

- The `Plugin` class itself.
- One of the entities you want to implement: a `DataSource`, `NotificationListener`, `MCO`, or `UIHook`.
- A `Factory` class for the entity above: it is responsible for creating the specific entity, for example, a `DataSource`
- A `Model` class which contains configuration options for the entity. For example, it can contain login and password information so that its data source knows how to connect to a service. The `Model` is also shown visually in the `force_wfmanager` UI, so some visual aspects need to be configured as well.

The plugin is made available by having it defined in the `setup.py` file `entry_points` section, under the namespace `force.bdss.extensions`. For example:

```
entry_points={
    "force.bdss.extensions": [
        "enthought_example = "
        "enthought_example.example_plugin:ExamplePlugin",
    ]
}
```

2.2.1 The plugin

The plugin class must be

- Inheriting from `force_bdss.api.BaseExtensionPlugin`
- Implement a `id` class member, that must be set to the result of calling the function `plugin_id()`. For example:

```
id = plugin_id("enthought", "example", 0)
```

- Implement a method `get_factory_classes()` returning a list of all the classes (NOT the instances) of the entities you want to export.
- Implement the methods `get_name()`, `get_version()` and `get_description()` to return appropriate values. The `get_version()` method in particular should return the same value as in the id (in this case zero). It is advised to extract this value in a global, module level constant.

2.2.2 The Factory

The factory must inherit from the appropriate factory for the given type. For example, to create a DataSource, the factory must inherit from `BaseDataSourceFactory`. It then needs these methods to be redefined

- `get_identifier()`: must returns a unique string, e.g. a uuid or a memorable string that must be unique across your plugins, present and future.
- `get_name()`: a memorable, user presentable name for the data source.
- `get_description()`: a user presentable description.
- `get_model_class()`: Must return the Model class.
- `get_data_source_class()`: Must return the data source class.

2.2.3 The Model class

The model class must inherit from the appropriate Base model class, depending on the entity, for example `BaseDataSourceModel` in case of a data source.

This class then must be treated as a Traits class, where you can use traits to define the type of data it holds. Pay particular attention to those data that can modify the slots. For those, add a `changes_slots=True` metadata tag to the trait. This will notify the UI that the new slots need to be recomputed and presented to the user. Failing to do so will have unexpected consequences. Example:

```
class MyModel(BaseDataSourceModel):
    normal_option = String()
    option_changing_slots = String(changes_slots=True)
```

Typically, options that change slots are those options that modify the behavior of the computational engine, thus requiring more or less input (input slots) or producing more or less output (output slots).

Many `BaseModel` subclasses also include a `verify` method, which is called before an MCO run starts to ensure that the execution will be successful. This verification step can also be triggered in the WfManager UI even before an MCO run is submitted. For `BaseDataSourceModel` subclasses it is automatically performed whenever the slots objects are updated, however developers can also include the `verify=True` metadata on any additional trait that requires verification. Including this in example above:

```
class MyModel(BaseDataSourceModel):
    normal_option = String(verify=True)
    option_changing_slots = String(changes_slots=True)
```

You can also define a UI view with traitsui (`import traitsui.api`). This is recommended as the default view arranges the options in random order. To do so, have a `default_traits_view()` method:

```
def default_traits_view():
    return View(
        Item("normal_option"),
        Item("option_changing_slots")
    )
```

2.2.4 The DataSource class

This is the "business end" of the data source, and where things are done. The class must be derived from `BaseDataSource`), and reimplement the appropriate methods:

- `run()`: where the actual computation takes place, given the configuration options specified in the model (which is received as an argument). It is strongly advised that the `run()` method is stateless.
- `slots()`: must return a 2-tuple of tuples. Each tuple contains instances of the `Slot` class. Slots are the input and output entities of the data source. Given that this information depends on the configuration options, `slots()` accepts the model and must return the appropriate values according to the model options.

2.2.5 The MCO class

Like the data source, the MCO needs a model (derived from `BaseMCOModel`), a factory (derived from `BaseMCOFactory`) and a MCO class (derived from `BaseMCO`). Additional entities must be also provided:

- `MCOCommunicator`: this class is responsible for handling communication between the MCO and the spawned process when the MCO is using a "subprocess" model, that is, the MCO invokes the force_bdss in evaluation mode to compute a single point.
- `parameters`: We assume that different MCOs can support different parameter types for the generated variables. Currently, only the "range" type is commonly handled.

The factory then must be added to the plugin `get_factory_classes()` list.

The factory must define the following methods:

```
def get_identifier(self):
def get_name(self):
def get_description(self):
def get_model_class(self):
```

as in data source factory. The following:

```
def get_optimizer_class(self):
def get_communicator_class(self):
```

Must return classes of the MCO and the MCOCommunicator. Finally:

```
def get_parameter_factory_classes(self):
```

Must return a list of classes of the parameter factories.

Optimizer Engines

The `force_bdss.api` package offers the `BaseOptimizerEngine` and `SpaceSampler` abstract classes, both of which are designed as utility objects for backend developers.

The `BaseOptimizerEngine` class provides a schema that can easily be reimplemented to act as an interface between the BDSS and an external optimization library. Although it is not strictly required to run an MCO, it is expected that a developer would import the object into a `BaseMCO.run` implementation, whilst providing any relevant pre and post processing of information for the specific used case that the MCO is solving. The base class must simply define the following method:

```
def optimize(self):
```

Which is expected to act as a generator, yielding values corresponding to optimised input parameters and their corresponding KPIs. A concrete implementation of this base class, the `WeightedOptimizerEngine`, is provided that uses the `SciPy` library as a backend.

The `SpaceSampler` abstract class also acts as a utility class in order to sample vectors of values from a given distribution. Implementations of this class could be used to either provide trial parameter sets to feed into an optimiser as initial points, or importance weights to apply to each KPI. The base class must define the following methods:

```
def _get_sample_point(self):
def generate_space_sample(self, *args, **kwargs):
```

Two concrete implementations of this class are provided: `UniformSpaceSampler`, which performs a grid search and `DirichletSpaceSampler`, which samples random points from the Dirichlet distribution.

2.2.6 MCO Communicator

The MCO Communicator must reimplement BaseMCOCommunicator and two methods: `receive_from_mco()` and `send_to_mco()`. These two methods can use files, stdin/stdout or any other trick to send and receive data between the MCO and the BDSS running as a subprocess of the MCO to evaluate a single point.

2.2.7 Parameter factories

MCO parameter types also require a model and a factory per each type. Right now, the only typo encountered is Range, but others may be provided in the future, by MCOs that support them.

The parameter factory must inherit from `BaseMCOParameterFactory` and reimplement:

```
def get_identifier(self):
def get_name(self):
def get_description(self):
```

as in the case of data source. Then:

```
def get_model_class(self):
```

must return a model class for the given parameter, inheriting from `BaseMCOParameter`. This model contains the data the user can set, and is relevant to the given parameter. For example, in the case of a Range, it might specify the min and max value, as well as the starting value.

2.2.8 Notification Listeners

Notification listeners are used to notify the state of the MCO to external listeners, including the data that is obtained by the MCO as it performs the evaluation. Communication to databases (for writing) and CSV/HDF5 writers are notification listeners.

The notification listener requires a model (inherit from `BaseNotificationListenerModel`), a factory (from `BaseNotificationListenerFactory`) and a notification listener (from `BaseNotificationListener`). The factory requires, in addition to:

```
def get_identifier(self):
def get_name(self):
def get_description(self):
def get_model_class(self):
```

the method:

```
get_listener_class()
 return the notification listener object class.
```

The NotificationListener class must reimplement the following methods, that are invoked in specific lifetime events of the BDSS:

```
def initialize(self):
    Called once, when the BDSS is initialized. For example, to setup the
    connection to a database, or open a file.

def finalize(self):
    Called once, when the BDSS is finalized. For example, to close the
    connection to a database, or close a file.

def deliver(self, event):
    Called every time the MCO generates an event. The event will be passed
    as an argument. Depending on the argument, the listener implements
    appropriate action. The available events are in the api module.
```

2.2.9 UI Hooks

UI Hooks are callbacks that are triggered at some events during the lifetime of the UI. It has no model. The factory must inherit from `BaseUIHooksFactory`, and must reimplement `get_ui_hooks_manager_class()` to return a class inheriting from `BaseUIHooksManager`. This class has specific methods to be reimplemented to perform operations before and after some UI operations.

Any `BaseDriverEvents` that are required to be delivered to a UI can be indicated using the `UIEventMixin` class. The `MCOStartEvent`, `MCOProgressEvent` and `MCOFinishEvent` are all examples of such objects.

2.2.10 Envisage Service Offers

A plugin can also define one or more custom visualization classes for the GUI application `force-wfmanager`, typically to either display data or provide a tailor-made UI for a specific user. In which case, the plugin class must inherit from `force_bdss.core_plugins.service_offer_plugin.ServiceOfferExtensionPlugin`, which is a child class of `BaseExtensionPlugin`. Any UI subclasses can then be made discoverable by `force-wfmanager` using the `envisage ServiceOffer` protocol through the `get_service_offer_factories` method:

```
def get_service_offer_factories(self):
    """A method returning a list user-made objects to be provided by this
    plugin as envisage ServiceOffer objects. Each item in the outer list is
    a tuple containing an Interface trait to be used as the ServiceOffer
    protocol and an inner list of subclass factories to be instantiated
    from said protocol.

    Returns
    -------
    service_offer_factories: list of tuples
        List of objects to load, where each tuple takes the form
        (Interface, [HasTraits1, HasTraits2..]), defining a Traits
        Interface subclass and a list of HasTraits subclasses to be
        instantiated as an envisage ServiceOffer.
    """
```

Make sure to import the module containing the data view class from inside `get_service_offer_factories`: this ensures that running BDSS without a GUI application doesn't import the graphical stack.

Custom UI classes

There are currently two types of custom UI object that may be contributed by a plugin: `IBasePlot` and `IContributedUI`. These interfaces represent requirements for any UI feature that can be used to display MCO data or a present a simplified workflow builder respectively.

Also, multiple types of plugin contributed UI objects can be imported in the same call. For instance:

```
def get_service_offer_factories(self):
    from force_wfmanager.ui import IBasePlot, IContributedUI
    from .example_custom_uis import PlotUI, ExperimentUI, AnalysisUI

    return [
        (IBasePlot, [PlotUI]),
        (IContributedUI, [ExperimentUI, AnalysisUI])
    ]
```

2.3 FORCE Project Developer Guidelines

When contributing to the FORCE project, please clone the relevant Git repository and perform all suggested changes on a new branch (there is no need to fork the repository). Some general guidelines are provided below.

2.3.1 Continuous Integration

The FORCE project uses a TravisCI runner that will build and test any code pushed to a GitHub repository. These CI checks can also be performed locally whilst developing.

Mandatory Checks

1) Contribute unit tests for any fixes or new features using the `unittest` library. Each module should contain a `test/` folder with python files contributing `unittest.TestCase` objects. You can run (from an `edm` environment) the following command to perform all unit tests within a repository:

```
python -m ci test
```

   All tests must pass in order for a pull request to be accepted.

2) Use Flake8 style guidelines for new code. These can be tested by running:

```
python -m ci flake8
```

   Note: If you enforce stricter style guidelines (such as Black), then this is fine as long as they also pass the Flake8 test.

Optional Checks

4) Some FORCE projects will also build Sphinx documentation as part of the CI. If this is failing for any reason, you can run the following command to debug locally:

```
python -m ci docs
```

5) Some FORCE projects have thresholds for code coverage that must be met when contributing new code. These will be defined within a `coverage.yml` file in the repository top-level directory. You can check the coverage by running:

```
python -m ci coverage
```

   Note: when performing this command locally, the final step will attempt to upload the report to Codecov, which will fail unless an appropriate token is present. This is unnecessary to simply view the coverage report and can be safely ignored.

2.3.2 Pull Request Review

When the CI tests are passing locally, push the branch to `origin` using:

```
git push --set-upstream origin <branch-name>
```

And create a GitHub pull request describing the changes made and the context for doing so by following the pull request template provided. Some repositories are protected and will explicitly require at least one reviewer for a pull request to be merged. However, we strongly request that ANY code is reviewed before merging.

When reviewing, try to ensure that the PR creator remains the only committer to the PR branch. A reviewer can make a new PR against the PR branch if they want to suggest extensive changes.

CHAPTER

THREE

API REFERENCE

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search