NASA/CR-20250007049

Model-based Systems Analysis & Engineering: Standard Evaluator

Joerg Gablonsky
The Boeing Company, Tukwila, Washington

Jeffrey Musiak
The Boeing Company, South Burlington, Vermont

Micah Goldade
The Boeing Company, Anchorage, Alaska

Eduardo Ocampo
The Boeing Company, Huntington Beach, California

Ranald M Engelbeck
The Boeing Company, Everett, Washington

Sean Wakayama
The Boeing Company, Huntington Beach, California

Alexander Carrere
The Boeing Company, Huntsville, Alabama

Shamsheer Chauhan
Collins Aerospace, Ann Arbor. Michigan

Melinda Refford
Collins Aerospace, Charlotte, North Carolina

August 2025

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION.
  Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM.
  Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT.
  Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION.
  Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.

- SPECIAL PUBLICATION.
  Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION.
  English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

NASA/CR-20250007049

Model-based Systems Analysis & Engineering: Standard Evaluator

Joerg Gablonsky
The Boeing Company, Tukwila, Washington

Jeffrey Musiak
The Boeing Company, South Burlington, Vermont

Micah Goldade
The Boeing Company, Anchorage, Alaska

Eduardo Ocampo
The Boeing Company, Huntington Beach, California

Ranald M Engelbeck
The Boeing Company, Everett, Washington

Sean Wakayama
The Boeing Company, Huntington Beach, California

Alexander Carrere
The Boeing Company, Huntsville, Alabama

Shamsheer Chauhan
Collins Aerospace, Ann Arbor. Michigan

Melinda Refford
Collins Aerospace, Charlotte, North Carolina

Prepared under Contract 80GRC023CA045

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

August 2025

Acknowledgments

This project and report reflect the combined efforts of the Boeing MBSA&E team. Funding for this effort was provided by the Sustainable Flight National Partnership Program, through the Advanced Air Transport Technology Project. The work was performed under NASA Contract 80GRC023CA045. Eric Hendricks was the NASA MBSA&E Program Manager. Ty Marien was the Contract Technical Representative. Irian Ordaz was the NASA sub-task lead for this work. Special thanks for the many discussions, feedback, and code that was shared with the team by Kenneth Moore, Robb Falck, and Eliot Aretskin-Hariton.

This work was sponsored by the Advanced Air Vehicles Program
at the NASA Glenn Research Center.

Level of Review: This material has been technically reviewed by expert reviewer(s).

This report is available in electronic form at https://www.sti.nasa.gov/ and https://ntrs.nasa.gov/

NASA STI Program/Mail Stop 050
NASA Langley Research Center
Hampton, VA 23681-2199

1. Abstract

The standard evaluator enables the full description of a discipline analyses used in a Multidisciplinary Analysis and Optimization (MDAO) assembly including any relevant options. It also enables assembling an MDAO assembly from a textual description, it's state from a HDF5 file, and optimization definition from a third textual description. These descriptions are independent of any specific MDAO framework, even though the current implementation that was developed as part of the contract is leveraging the NASA OpenMDAO framework (J. S. Gray, 2019).

This enables replacing components inside an assembly in a programmatic and simple way, for example to replace a group of computationally expensive analyses with a surrogate model or a simplified analysis. It also allows to create an assembly via a textual description and paves the way to build a GUI for creating and editing the assembly.

This, together with the ability to define the optimization problems to be solved in an MDAO problem via a textual and standardized format, allows the capture and version control of MDAO studies. This becomes a key enabler of integration with the digital thread and modern development processes.

This paper describes the need for programmatically defining options for components in an MDAO assembly, especially when wanting to run components that are hosted remotely and not within the same computing environment / memory space of the MDAO framework.

We further describe the independent assembly definition that was developed and implemented as a Python library and show multiple examples of using this definition and the library.

2. Contents

3. Nomenclature

CI/CD: Continuous Integration/Continuous Development – Aims to streamline and accelerate the software development lifecycle

CLIN: Contract Line Item Number - A specific task or deliverable defined in a government contract.

GitHub: A web-based DevOps lifecycle tool that provides a Git-repository manager with wiki, issue-tracking, and CI/CD pipeline features.

HDF5: Hierarchical Data Format version 5 - A file format designed to store and organize large amounts of numerical data.

JSON: JavaScript Object Notation - A lightweight data interchange format based on JavaScript object syntax.

MBE: Model-Based Engineering - An approach that uses models as the primary artifacts of the engineering process.

MBSA: Model-Based Systems Analysis - The application of modeling to support system analysis activities.

MDAO: Multi-Disciplinary Analysis and Optimization – Analyzing and optimizing problems leveraging multiple different disciplines working in conjunction and influencing each other.

NASA: National Aeronautics and Space Administration - The United States government agency responsible for the civilian space program and aerospace research.

OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization developed by NASA.

Pydantic: A data validation and settings management library using Python type annotations.

4. List of Figures

5. Introduction

National Aeronautics & Space Administration (NASA) Aeronautics engaged with industry, academic, and other agencies through the Sustainable Flight National Partnership (SFNP) to accomplish the aviation community's goal of net-zero carbon emissions by 2050. The Advanced Air Transport Technology (AATT) Project was a contributing member of the SFNP and was tasked with integrating simulation data, ground test data, and flight demonstration data for system-level vision-vehicle performance and technology assessments. The effort toward collating and integrating technology and vehicle data across the SFNP led by the AATT Project was dubbed Model-Based Systems Analysis & Engineering (MBSA&E). NASA sponsored a set of projects to stimulate innovation and develop industry partnerships supporting the MBSA&E effort. The Standard Evaluator project described in this report was one of six projects performed by Boeing.

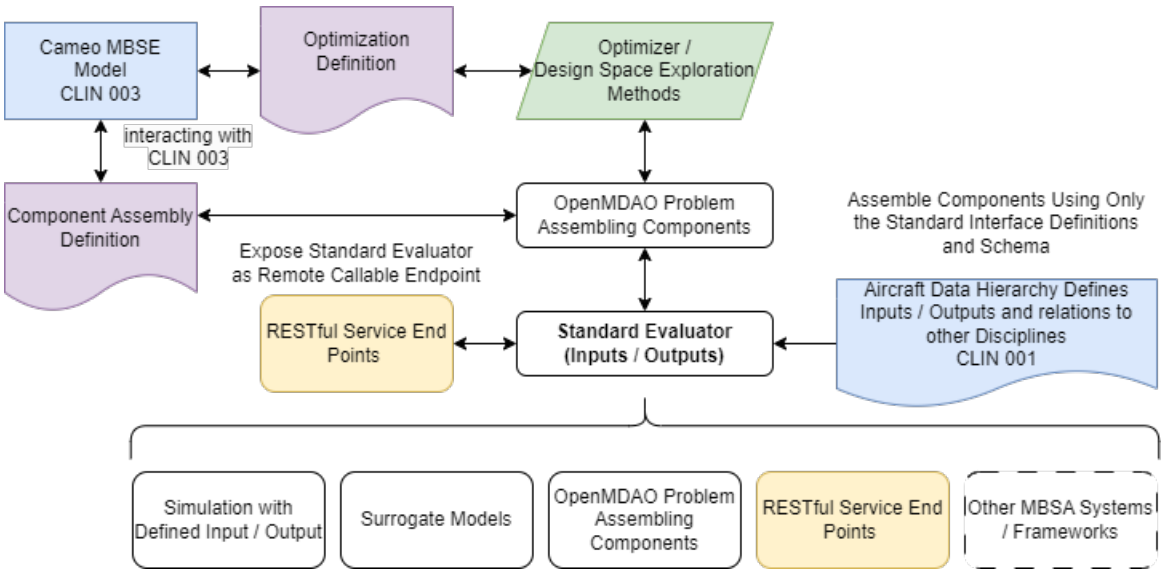

Figure 1 Overview of the functionalities implemented and their context

Figure 1 shows the vision that was outlined at the beginning of the project, and highlights the two main capabilities implemented in the standard-evaluator Python open-source library:

- Component Assembly Definition
- Optimization Definition

Both of these are MDAO framework independent definitions that are implemented via Pydantic classes and allow the storing of the information in JSON format, and through that enable version control of both the assembly of a Multi-Disciplinary Analysis and Optimization (MDAO) and the actual optimization definitions used in a study. The library developed in this project leverages OpenMDAO, but other frameworks like Ansys ModelCenter, Dassault 3DX, or Siemens Simcenter HEEDS could also use this information. Conversion to the CMDOWS format (Imco van Gent, 2018) should be possible but has also not been explored.

While not implemented under the current contract the component assembly information and optimization definition can be exchanged with Cameo Models leveraging the capabilities delivered in the CLIN003 project, and the definition of inputs and outputs and relationships between disciplines could be managed by the Aircraft Data hierarchy developed under CLIN001.

The other capability developed during the contract was a prototype to enable the full description of a discipline MDAO/MBSA assembly including any relevant options. This ensures that options are well defined and can be programmatically discovered compared to the current approach where what options are available or required to instantiate a discipline or change its behavior is burdensome.

When soliciting proposals for this contract NASA expressed the goal to have the ability to do studies where you can easily change fidelity levels of simulations, a functionality enabled via the standard-evaluator library. The independent definition of components allows the development of wrappers that expose the same simulation to multiple frameworks instead of the current state where simulations need to be wrapped repeatedly for different integration frameworks.

Furthermore, assembling simulations and components via programming can be limiting for some user communities, which prefer graphical user interfaces (GUI). The assembly definition allows developing such a GUI in different programming languages. Finally, the assembly definition together with utility routines in the standard-evaluator library provides the ability to exchange simulations of different fidelities.

A team from DLR (German Aerospace Center) Institute of System Architectures in Aeronautics build a system they called MDAx that tries to address some of the GUI aspects of creating workflows (A. Page Risueño, 2020), but choose the XDSM standard to store the assembly information. Exploring how to combine the work done by DLR and in this work is future work.

Another related effort is the CMDOWS proposed standard to store and exchange MDO systems. It has similar goals to the work done under this contract but does not approach the assembly in an object-oriented approach as a nested system as what was implemented in the standard-evaluator library (Imco van Gent, 2018). It should be possible to write converters from one format into the other and back.

6. Programmatically defining options

Components of a MDAO assembly have two different types of inputs to them:

- **Options** which are set when the component is initialized
- **Inputs** that define the values used to calculate the outputs of the component and are expected to be changed every time the component is executed.

Examples of options are dimensions of inputs, names of files needed to run an analysis, or values that are expected to be fixed during execution.

While it sometimes makes sense to store values that are fixed during a study in the options, it is best practice to avoid doing this. Instead, it is better to set those as inputs with bounds that are equal so that the component can be reused in a study where this input might become a design variable.

When building reusable components, it is important to programmatically define options. For example, the NASA Aviary library (Carl Recine, 2024) contains a large number of components implementing components for MDAO of airplanes, many of which have options that drive the behavior of a component.

If a user wants to use a component, for example that is part of a library like Aviary, the user needs to understand what options are required to instantiate the component when adding it to an MDAO assembly. There are a few different ways of finding out how to use the components.

1. **Looking at the source code.** This is a very human centered approach, and often can be very time consuming, especially when the component is complex. It might require multiple attempts by the user to figure out what the options are, and how to set them. Finding default values is not always easy and knowing what valid inputs for the option are requires deep understanding of the component. It is nearly impossible for an automated process to do this which is a requirement to develop a Graphical User Interface (GUI) to create assembles of components.
2. **Looking at the documentation of the code.** This is another human centered approach, with most of the drawbacks of looking at the source code. While more automation is possible with this approach, it requires good discipline by developers of components to ensure the documentation is correct. While there are some good tools to automate the creation of documentation, when code gets updated, there is a high risk of the documentation not being correct.

3. **Providing programmatic way of describing the options, including default values, ranges, and explanations.** This approach is designed to work with automation as well as with a human user and avoids the need to access to source code of the component. This is especially useful for components or simulations that are proprietary or when the component is executed via remote calls. It also enables the development of a GUI to build assemblies of components.

In this work we created a Python class that is an example implementation of an API to enable a developer of a component to programmatically define options, and for a user of a component to inquire about the options.

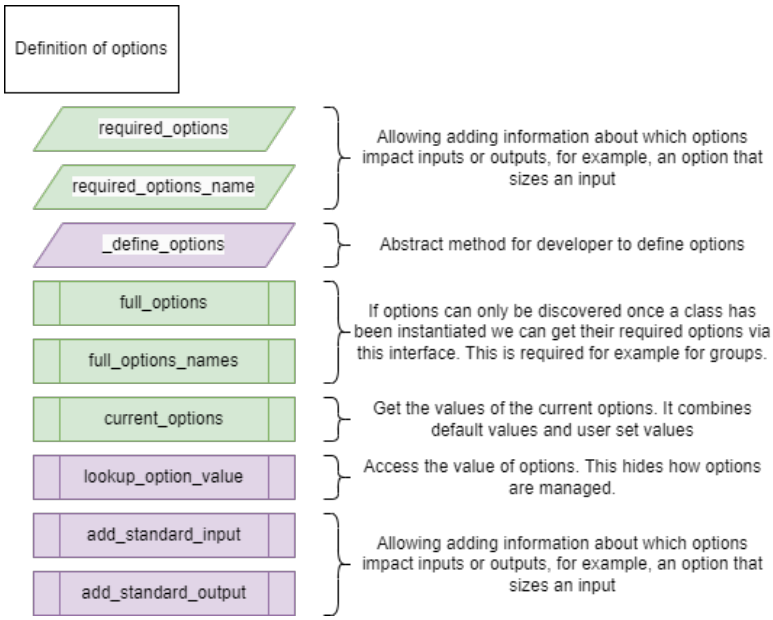

Figure 2: API to programmatically define options

Figure 2 shows the API that allows the programmatic definition of options. This is currently implemented as a separate class that can be used with dual inheritance in conjunction with OpenMDAO classes, but it could be included in OpenMDAO directly by adding this API to the *System* class.

The API defines 9 main new methods. The first three are class methods which can be called without the need to instantiate the class, thereby making it possible to get the defaults and information about the options without the need to instantiate the class (shown as trapezoids in Figure 1). A user can call the class method *required_options,* store the results in a variable, update the options from their default, and then can instantiate the component with the new options.

- **required_options** is a class method that returns a descendant of an OpenMDAO *OptionsDictionary* named *UnitsOptionsDictionary* to allow the definition of units for an option. It allows the definition of all options, their types, bounds, default values, and descriptions, among other things.
- **required_options_names** is a class method that returns just the names of all required options.
- **_define_options** is a class method that a developer of a component should overwrite to define the options for this component. Part of defining the options is to not just define what they are, but also to define bounds, units, types, descriptions etc.

The next six methods are instance methods (double boxes in the figure), that is, they can only be called when the class has been instantiated. The first three are user facing methods (green background in Figure 1), with the last three methods targeting component developers and shaded in light purple.

- **full_options** is like the *required_options* class method but can consider that some options are only discoverable after a component, and specifically a group, have been instantiated. An example is below.

- **full_option_names** returns the names of the full options.
- **current_options** returns the values of the current options as an *UnitsOptionsDictionary*.

The reason we need to have a *full_options* instance method in addition to the *required_options* class method especially for groups is that for a group implemented in OpenMDAO we only know the components inside the group once the group is instantiated, since the components are added to the group in the *setup* method. So only after the *setup* method has been called does the group know which components are inside the group. In the implementation of the **full_options** method the method iterates over all components inside the group, calls the **full_options** method of each component, and then merges all options into the options of the class.

The last three main methods are used by the developers of a component.

- **lookup_option_value** should be used inside the computational methods of a component when trying to access the value of an option. This will check that the option is defined as part of the required options for this component and hide from the developer how options are managed.
- **add_standard_input** supersedes the OpenMDAO *add_input* method and allows the developer to specify that an input depends on one or more options. This will allow more visibility into which options impact which input to the component. Note that this is not required to be done if it is too difficult to link an option to a specific input or output. This method also allows the developer to look up the value, units, or description of an input via a *_lookup_info* method if they have not been set, and the *_lookup_info* method has been implemented.
- **add_standard_output** supersedes the OpenMDAO *add_output* method, and does the same thing as *add_standard_input* for outputs.

A demonstration of this functionality can be found in the GitHub project for the standard-evaluator under the demos folder https://github.com/Boeing/standard-evaluator/blob/main/docs/source/demos/defining_options.ipynb, or in the documentation https://boeing.github.io/standard-evaluator/demos/defining_options.html.

I. Demonstration of using the programmatically defined options

This demonstration shows how to create a component and group that allow an user to programmatically define options. It shows how to access the default options of a component using the *required_options* class method, and the default options of a group using its *full_options* instance method once the group has been initialized.

The demo finishes with showing how to get the current values of all the options required to set up a group using the *current_options* instance method.

For brevity we only include the Python source code defining the component that is using the new features. For the full demonstration the reader is referred to the GitHub project.

```
class Paraboloid(StandardBase, om.ExplicitComponent):
    """
    Evaluates the equation f(x,y) = (|x|-3)^2 + |x|y + (y+4)^2 - 3.
    """

    @classmethod
    def _define_options(cls) -> OptionsDictionaryUnit:
        """Abstract method that allows a developer to define information about the
parameters this class uses.

        This method is called by a class method that is adding options to the
`class_options` option.

        Returns:
```

```
            OptionsDictionaryUnit -- The required options for this class, their
default values, and, if required, their units.
        """
        options = OptionsDictionaryUnit()
        options.declare("dimensions", default=40, lower=1, types=int)
        return options

    def setup(self):

        self.add_standard_input('x',
val=np.ones(self.lookup_option_value("dimensions")), options=["dimensions"],
            look_up=False,)
        self.add_input('y', val=0.0)

        self.add_output('f_xy', val=0.0)

    def setup_partials(self):
        # Finite difference all partials.
        self.declare_partials('*', '*', method='fd')

    def compute(self, inputs, outputs):
        """
        f(x,y) = (x-3)^2 + xy + (y+4)^2 - 3

        Minimum at: x = 6.6667; y = -7.3333
        """
        x = inputs['x']
        y = inputs['y']
        print(f"Dimension of x: {len(x)}")
        outputs['f_xy'] = (np.linalg.norm(x) - 3.0)**2 + np.linalg.norm(x) * y + (y +
4.0)**2 - 3.0
```

7. Independent assembly definition

A key deliverable of this CLIN was to develop an integration platform independent description of MDAO assemblies using Pydantic classes and the JSON files generated when storing instances of these classes. The description can also be loaded into automatically generated instances of the Pydantic classes.

Utilities to take an OpenMDAO problem or system and extract the relevant information in the new format were created, as were utilities to create an OpenMDAO problem from the new format.

This independent format will allow the ability to move between different integration frameworks for MDAO assemblies like Ansys ModelCenter, Dassault Systems 3DX, Siemens Simcenter HEEDS, and of course OpenMDAO. Work under this contract has only developed capabilities to leverage the format with OpenMDAO, but integrations with the other frameworks can be developed.

Having the ability to move between frameworks will avoid vendor lock-in, allow users to leverage the unique capabilities of different integration frameworks, and allow teams to leverage the same assemblies in multiple frameworks.

Of course, each framework will have their own unique capabilities, for example, the ability of OpenMDAO to leverage analytical gradients is a powerful and unique capability.

II. Element descriptions via Pydantic classes

The first classes defined enable capturing information about inputs and outputs, or elements, to a component or evaluator. Currently the following classes were defined, see the source code of the library for more details:

- **FloatVariable**: storing information about floating point elements
- **IntVariable**: storing information about integer elements
- **ArrayVariable**: storing information about multi-dimensional elements

For each element type the definition allows the capturing of the following information:

- **name** is the name of the element
- **default** optionally is the default value for this element
- **shift** optionally is a shift value for this element. This will be mostly used in conjunction with optimization problem definitions defined later.
- **scale** optionally is a shift value for this element. This will be mostly used in conjunction with optimization problem definitions defined later.
- **units** optionally allow the definition of units in string form
- **description** optionally allows a textual description of the element. This enables a more expressive explanation of an element compared to the name of the element
- **options** is a dictionary that can be used to store additional information about an element. Note that the content in this should be strings or must be a Python object that is serializable. If it is not a string this could create issues when trying to leverage the information in a non-Python application.

The Pydantic classes add checks for consistency to make sure elements are valid. The *ArrayVariable* allows he definition of multi-dimensional arrays.

The *Variable* class is a union of the three classes described in this section that allows defining an element of the more complex classes described below to be any of the three types.

III. Evaluator or Component description

Three classes were defined to describe the interface for components inside an MDAO assembly. More classes might be developed in the future, for example a class capturing information about a component that wraps an executable.

The first class is called *EvaluatorInfo* and is used for capturing the information of a basic component, or evaluator. It is the parent of all the other component information classes. The following fields are defined for it:

- **name** is the name of the evaluator
- **class_type** is a class identifier used to support inheritance in Pydantic
- **inputs** list of input elements. Must contain at least one element. All inputs and outputs are one of the *Variable* classes defined above.
- **outputs** list of output elements. Can be empty.

- **description** optionally allows a textual description of the element. This enables a more expressive explanation of a component compared to its name. To define mathematical symbols, use markdown syntax.
- **cite** optionally allows the listing of relevant citations that should be referenced when publishing work that uses this class.
- **tool** optionally stores the name of the tool exposed in the component. When used in conjunction with OpenMDAO the information about the Python class and module that defines the component is stored in this field which allows recreation of the component in a Python interpreter.
- **evaluator_identifier** optionally allows to define a unique identifier for the evaluator Uniqueness of that identifier is responsibility of the owners of components and will be used when building large collections of components.
- **version** optionally allows the storing of version information of the evaluator
- **component_type** optionally allows to store the type of a component, for example ExplicitComponent, ImplicitComponent, Group, etc.
- **options** is a dictionary that can be used to store additional information about an element. Note that the content in this should be strings or must be a Python object that is serializable. If it is not a string this could create issues when trying to leverage the information in a non-Python application.

The *EquationInfo* class expands the *EvaluatorInfo* class by adding the *equations* field which allows to store one or multiple strings containing mathematical equations using a Python language syntax.

The last of the currently defined classes describing components in an MDAO assembly is the *GroupInfo* class. As the name suggests instances of this class store information about a group of components. In addition to the fields from the *EvaluatorInfo* class the following fields are defined:

- **component_order** stores a list of the names of components in this group in the order they should be assembled
- **components** stores a dictionary with information about the component in this group. Keys for the dictionary are the names of the component as used within the group, and used in the *component_order* list
- **promotions** stores a dictionary of lists that map the name of an input / output in a component to an input / output of this group. Note that only components inside the group can be referenced in this. The assumption is that only inputs or outputs for components inside a group that are promoted can be used when interacting with the group. That means inputs or outputs of a group that are not explicitly promoted should be considered internal variables to the group. See section vi for an in-debt discussion of this.
- **linkage** contains a map that allows linkage between inputs and outputs between components inside this group. Needs to use *component_name.element_name* format. Note that only components inside the group can be used.

The *JoinedInfo* class is a union of the three classes described in this section that allows referencing components of all classes described in this section abstractly and recursively.

IV. Editing an assembly and independent execution of the assembly

To explain how to use the new assembly information and the utility routines that were developed under this contract several examples were generated.

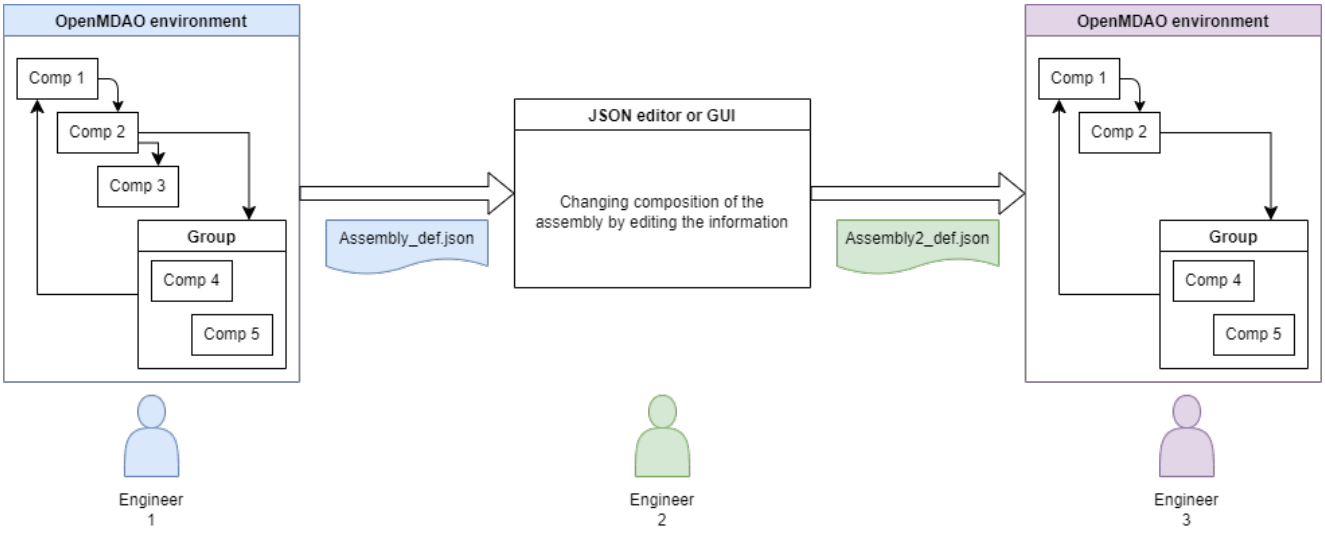

Figure 3 Pictorial description of the process to take an assembly, edit it via the JSON file, and create e new assembly

Figure 3 show the first workflow that was demonstrated in this project. The source code for this example can be found on GitHub at the following three locations:

1. Capture the assembly definition: https://github.com/Boeing/standard-evaluator/blob/main/docs/source/demos/group_creation_NASA.ipynb
2. Example of editing the created JSON file to remove a component: https://github.com/Boeing/standard-evaluator/blob/main/docs/source/demos/group_manipulation.ipynb
3. Loading assembly information from the modified JSON description and creating an OpenMDAO problem from it: https://github.com/Boeing/standard-evaluator/blob/main/docs/source/demos/group_reading.ipynb

In this scenario we assume that there are three different engineers involved. The first is the developer of the large, overarching MDAO model. They build a large assembly representing many different aspects and combining many different analysis capabilities into an overall model. Often this model will have many components and a large run time for even a single execution, much less running a full optimization. Sometimes a few components in the assembly dominate execution time. In many cases the assembly is organized as groups within groups.

With the new capability developed in this contract the first engineer can extract the assembly information from the instantiated OpenMDAO problem and save the information in a JSON file. One benefit of this is that the JSON file is a text file and can be managed by a version control system.

The second step of the demonstration assumes that a decision was made to do a study that only involves outputs from a subset of the analyses that are in the full assembly. The second engineer is asked to create a new description of the specific subset of the assembly that only contains the components needed for this new study. The engineer loads the full assembly into an editor and removes all the components and the references to the inputs and outputs of those components and stores the new description in another JSON file. This new JSON file can again be managed by a version control system.

In the current demonstration the editing of the assembly is done in another Python script / Jupyter Notebook, but it could be done for example in a JAVA based Graphical User Interface. The Python code used for this demonstration is very specific to the example, but a more general Python script could be developed to automatically remove specific components and all their references.

In the final step of the example a third engineer is executing the actual study and instantiates a new OpenMDAO problem using the new JSON description and the utilities in the standard-evaluator library.

The library also provides functionality to extract the state of an OpenMDAO problem, that is, the values of all inputs and outputs, and stores it in the HDF5 binary format. The decision to use HDF5 was made since in some MDAO models the state can be very large, especially if grids are used as ways of connecting one component to another. State is saved and loaded from an HDF5 file (binary). Additional functionality to set the state is also provided in the library.

V. Assembling an MDAO problem from a catalog of components

The functionality shown in the prior example is a pre-cursor to the ability of creating the MDAO assembly directly from a catalog of components as shown in Figure 4. JSON files containing the interface information for each of the components in the catalog can be generated, and optionally version controlled. A system to assemble them into an MDA could be developed, either by building utilities that use an input file, or by building a Graphical User Interface (GUI).

This has not been developed as part of the contract but is an opportunity for follow-on work.

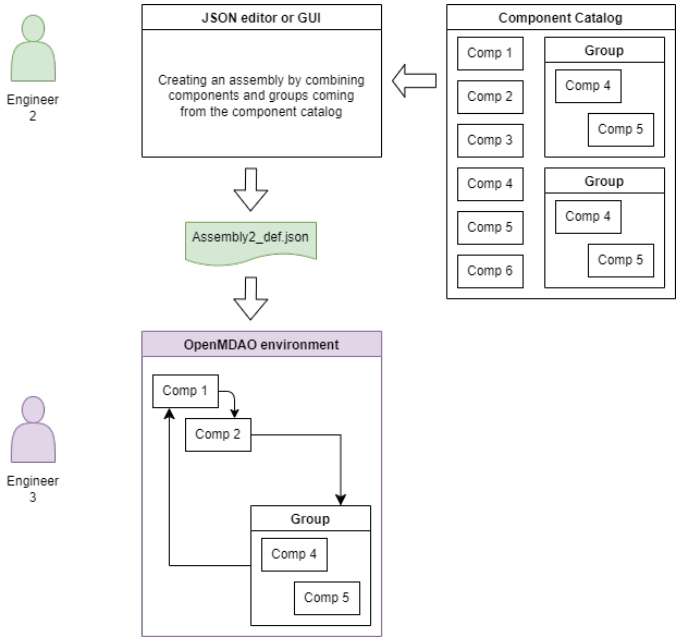

Figure 4 Creating an MDAO Assembly from a catalog

VI. Explicit definition of inputs and outputs of a group

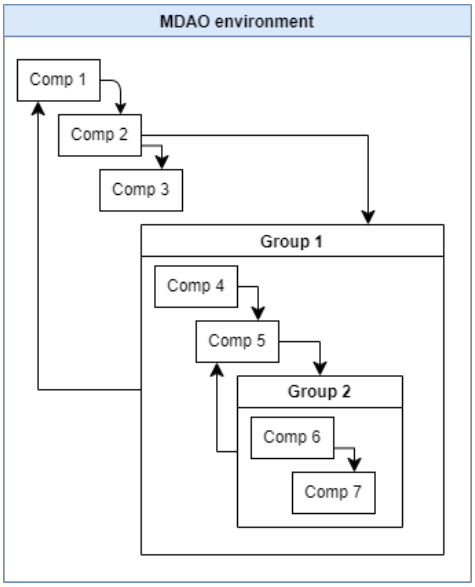

Figure 5 Example of an MDA with nested groups

Figure 5 shows an example of a commonly seen situation when creating a Multidisciplinary Analysis (MDA) assembly. An inner group (Group 2) is nested inside an outer group (Group 1) which itself is part of the MDA.

In many MDAO frameworks, including OpenMDAO, all inputs and outputs of all components are visible to the full system. For example, component *Comp 2* in the assembly can access any input or output of components *Comp 6* and *Comp 7* even if they are not promoted as inputs or outputs of *Group 2*.

The philosophy the standard-evaluator supports is a more restrictive view that each component in an MDA, including groups, define an explicit definition of inputs and outputs, and only those can be used by other components. This follows the concept of encapsulation from object-oriented programming where an object has external and internal variables, and a user of the object can only access the external variables.

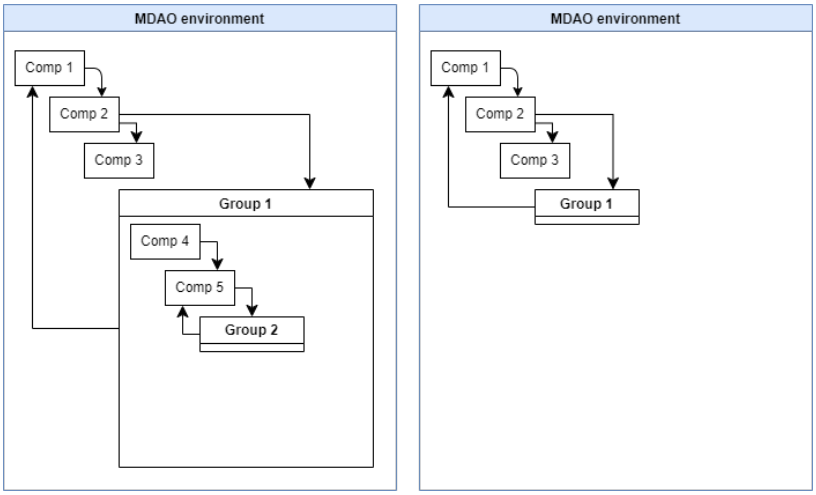

Figure 6 Folding of the nested group in an MDA

Figure 6 shows that with the assembly definition *Comp 5* only sees the defined external interface of *Group 2* and cannot see any of the internal variables. Similarly, *Comp 2* and *Comp 1* only see and can access the external interface of *Group 1* and doesn't even know that there is an internal *Group 2* inside *Group 1*.

There are many benefits for ensuring explicit definition of interfaces to a group.

First, it enables the owner of the group to change things internal to the group if the interface to the group does not change without impacting any user of the group. If the owner or developer of the group decides that all inputs and outputs should be accessible to a user of the group, they can always expose all of them via the interface definition of the group. In OpenMDAO this is easily done by promoting all using the "*" shortcut.

Second, if a group is being replaced by for example a surrogate model the surrogate model needs to know all inputs and outputs that are being consumed to allow this to happen. As we show in section VII, often the inputs and outputs of components inside a group that are internal can be large and not modeling them with a surrogate model can significantly reduce the complexity of the surrogate model.

A third reason is that while many times MDA are executed within a single computing environment there is an increased desire to be able to execute MDA where some of the components are executed remotely. When trying to execute a group remotely data needs to be sent to and from the remote computer, so an explicit external interface is required since only that data is being send and retrieved to and from the remote environment.

If the remote execution of a group or component is done across company boundaries this is even more important. Assuming an MDA is assembled by company A and they want to add a group that is owned and managed by company B. Company B wants to control what information is being accessible by company A, so an explicit interface that does not expose any of the interna of the group is critical.

Finally, if an MDA is used in conjunction with the ideas of a digital thread having the explicit definition of internal and external inputs and outputs is also very important. This allows tracing of relationships between components and being able to decide which components in an assembly need to be rerun when an input changes. While MDAO frameworks like OpenMDAO build graph networks that allow that tracing, especially when analytical derivatives are provided, it is simplified by defining explicit interfaces distinguishing between external and internal elements.

VII. Replacing a group with a new component

A key deliverable of this work is enabling the ability to replace a component in an OpenMDAO assembly with another component, which is what is shown in the demonstration in https://github.com/Boeing/standard-evaluator/blob/main/docs/source/demos/replace_group.ipynb.

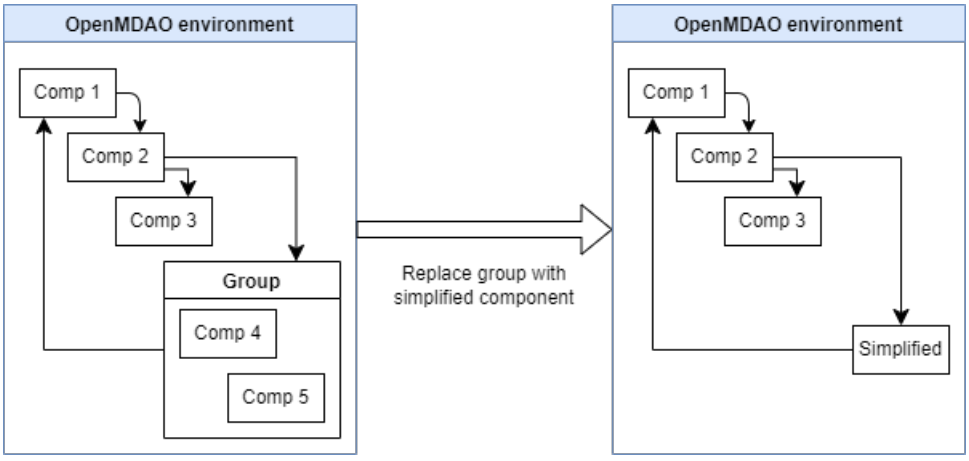

Figure 7 Replacing a group with a simplified component using the standard-evaluator library

Figure 7 shows the idea in graphical form. The group will be replaced with a simplified component. In the specific example this simplified component performs the same calculations as the group, although it collapses some calculations. This functionality will often be used to replace a computational expensive group with a significantly cheaper surrogate model. The work done under this contract did not include using a surrogate model.

Below is the source code of the *Aero* class used in this example. The group contains two components, a *geometry* and a *cfd* component. The geometry component has three floating-point inputs (*q, z,* and *l*), and two outputs that are one or two-dimensional arrays (*grid* and *bla*). *Grid* is an input to the *cfd* component and builds the connection between the two components. *Cfd* has two more floating point inputs (*r* and *k*) and a single floating-point output (*drag*). This represents a very common workflow where a parameterized geometry component creates a grid for a computational fluid dynamics (CFD) code which uses some additional parameters to perform calculations and then extract a single or a small number of characteristics or measures of the design. Note that it would be possible to split this step up and create a CFD component and a post-processing step.

To tell the standard-evaluator that the *grid* and *bla* outputs or inputs are internal to the group we use the 'internal' tag, which allows the library to distinguish these elements from the external variables for the group, which are *q, z, l, r, k,* and *drag*.

```
class Aero(om.Group):
    def setup(self):
        self.add_subsystem('geometry', om.ExecComp([f"grid = (q + z
*l)*ones({grid_size})", f"bla = outer(ones({grid_size}), ones({grid_size}))"],
                    grid={'tags': 'internal', 'shape': (grid_size)},
                    bla ={'tags': 'internal', 'shape': (grid_size, grid_size)}
                    ),
                    promotes_inputs=['q', 'z', 'l'])
        self.add_subsystem('cfd', om.ExecComp(f"drag = (inner(grid,
ones({grid_size})) + r * k)/1000",
                    grid={'tags': 'internal', 'shape_by_conn': True},
                    drag={'units': 'N'}),
                    promotes_inputs=['r', 'k'],
                    promotes_outputs=['drag'])
        self.connect('geometry.grid', 'cfd.grid')
```

Below we show the source code of the simplified component that we use to replace the *aero* group. Note that we call the subsystem which is the simplified component aero so that the it has the same name as the group in the full assembly.

```
class Surrogate(om.Group):
    def setup(self):
        self.add_subsystem('aero',
                om.ExecComp("drag = ((q + z *l)*20 + r * k)/1000",
                drag={'units': 'N'}),
                promotes_inputs=['q', 'z', 'l', 'r', 'k'],
                promotes_outputs=['drag'])

prob_replace = om.Problem(model=Surrogate())
```

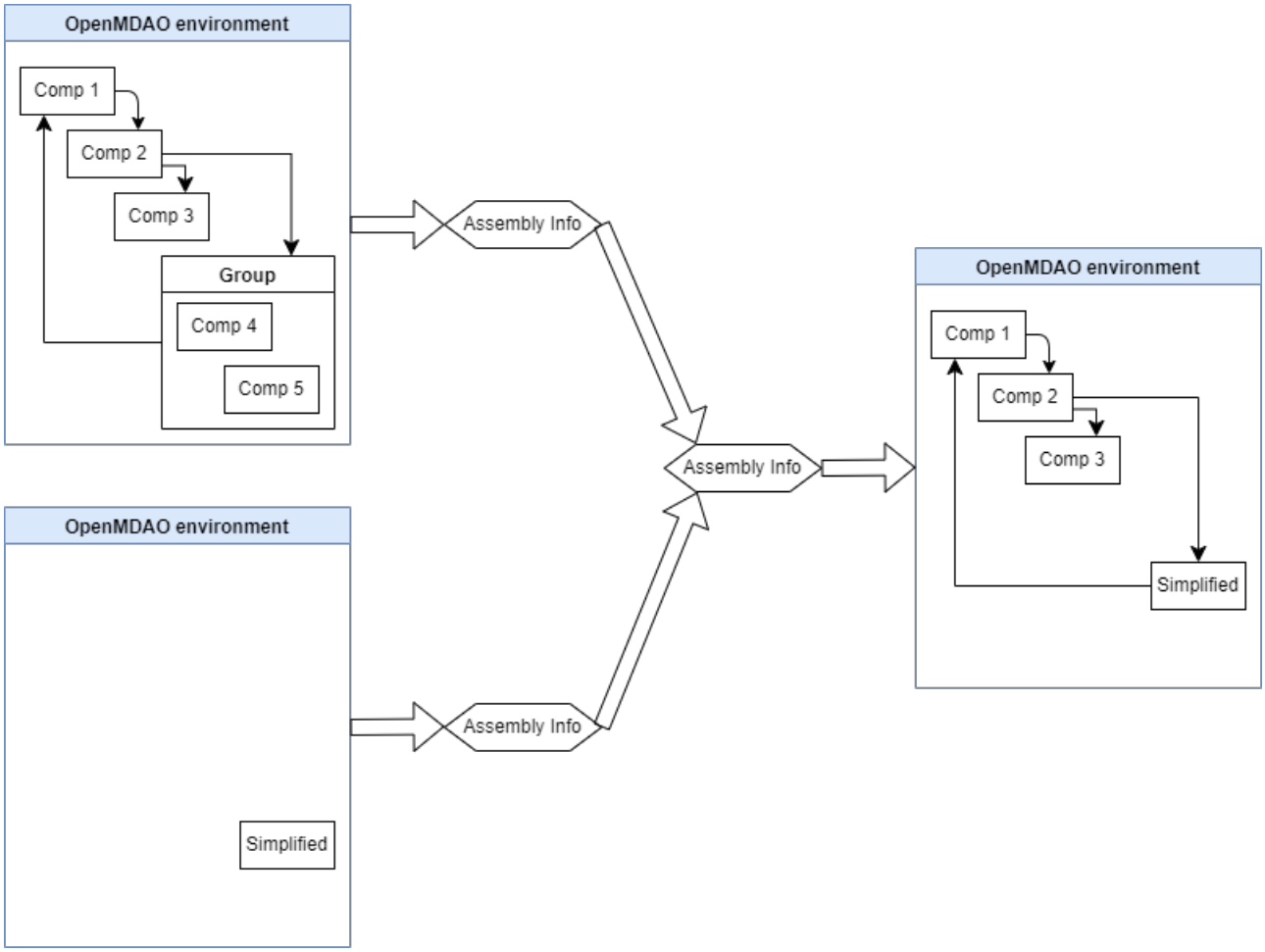

Figure 8 Details of how the standard-evaluator enables replacing a group with a simplified component

Figure 8 shows the steps that are required to replace the *aero* group with the simplified model. First, both an instance of the original assembly as well as an instance of an OpenMDAO problem that only contains the simplified component are created. The assembly information for both is extracted using the *get_interface* utility from the standard-evaluator library, and then we create a new assembly description where we replace the group with the information for the simplified component. This is done in a single line of Python code:

```
info.components['aero'] = surrogate_info
```

where *info* contains the description about the full assembly, and *surrogate_info* is the information about the simplified component.

The final step is to instantiate the new OpenMDAO problem using the edited assembly information to be able to use the assembly with the simplified model in place of the *aero* group. The state, that is values of all inputs and outputs that are in both problems, can be transferred from the full model to the simplified model using utility routines from the library.

8. Independent Optimization Definition

Another aspect of design space exploration and MDAO that is being addressed by the standard-evaluator library is a way to define an optimization problem independent of a specific framework. As with other parts of the library this was done by defining a Pydantic class and associated JSON schema. The *OptProblem* class has the following fields:

- **name** stores the name of the optimization problem. This field is used when serializing and printing information. Defaults to 'opt_problem'. Note that when running multiple different optimization problems having the ability to name each is beneficial.
- **variables** are a list of *Variables* instances that are the design or input variables for this optimization problem. At least one variable must be defined. Note that a user can set the upper and lower bounds of a variable to the same value and thereby fixing this variable.
- **responses** are a list of *Variables* instances that are the responses or output variables for this optimization problem. At least one response needs to be defined. All objectives and constraint used in the optimization problem must be defined as a variable or response.
- **objectives** contain a list of strings with the names of the objective(s) for the optimization problem. Must be either variables or - responses defined in the problem.
- **constraints** contain a list of strings with the names of the constraints of the optimization problem. Must be responses defined in the problem. To define bounds on variables, use the variable bounds. The upper and lower bounds of a constraint is defined when defining the associated response.
- **description** optionally allows a textual description of the optimization problem. This enables a more expressive explanation of the optimization problem. To define mathematical symbols, use markdown syntax.
- **cite** optionally allows the listing of relevant citations that should be referenced when publishing work that uses this optimization problem.
- **options** is a dictionary that can be used to store additional information about an optimization problem. Note that the content in this should be strings or must be a Python object that is serializable. If it is not a string this could create issues when trying to leverage the information in a non-Python application. Options to drive the behavior of an optimizer that is being used to solve this problem could be stored here, but of course that becomes something that is not optimizer independent.

Note that when defining objectives or constraints that involve array variables the user can define a specific element of the variable to be an objective or constraint via normal Python array syntax, that is if the variable `reach` is a two-dimensional array and the user wants to use the third element of the second row as the objective they would use `reach[1,2]` as the objective.

VIII. Defining optimization studies in a textual format

One of the things that the independent optimization problem definition developed in this contract enables is a workflow where multiple optimization problems are solved for a given assembly of simulations. That is, instead of solving a single MDAO problem the engineer is running a study to understand how the solution of the optimization problem changes when for example bounds are changed. Or in many industrial uses of MDAO the engineer is asked to create a series of solutions or even a Pareto front of a multiple objective optimization problem to show to decision makers. Often there are considerations that are not easily expressed in a simulation or mathematical formula that decision makers want to consider, and presenting solution from multiple optimization problems provides the data the decision makers need.

In such a situation it is also beneficial to version control the optimization problems as well as the definition of the assembly used.

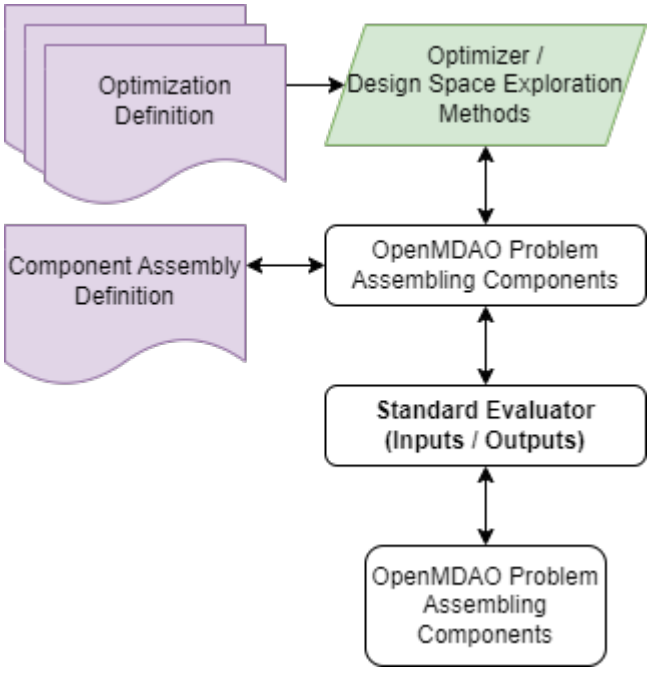

Figure 9 Exploring the design space via file-driven MDAO

Figure 9 shows this idea, where multiple optimization definitions for a specific assembly definition are created, and can then be executed in the chosen framework. In the example in https://github.com/Boeing/standard-evaluator/blob/main/docs/source/demos/design_space_exploration.ipynb we of course use OpenMDAO as the MDAO framework, and show how to create, store, manipulate, and load a series of optimization problems.

9. Availability of the implementation

The standard-evaluator library that was developed is easy to install via pip as it is deployed to PyPi: https://pypi.org/project/standard-evaluator/.

Source code of the library is available and maintained at https://github.com/Boeing/standard-evaluator/tree/main and documentation is available at https://boeing.github.io/standard-evaluator/.

The library is released under the Apache License, Version 2.0.

10. Suggestions for OpenMDAO

During the development of the library the authors gained deeper understanding of the OpenMDAO library, and have the following suggestions for the OpenMDAO team:

- Add methods to define and access programmatically defined options to OpenMDAO library, see section 6.
- Add ability to define inputs and outputs as internal to a group (make the 'internal' tag a suggested tag for OpenMDAO users?) and create routines that check that no internal elements are accessed within an OpenMDAO problem
- Add ability to detect that inputs and outputs of a group are internal based on the current OpenMDAO problem and suggesting to the developer of the group to mark them as internal.
- Add ability to report to a user which component in an assembly are required to calculate specific outputs of an OpenMDAO problem? What components are impacted by changes to an input to the OpenMDAO problem? Those utilities will assist users with pruning components from an assembly that are not required for a specific study.

11. Next steps and future vision

The standard-evaluator library developed under this contract enables several new capabilities.

It proposes a solution to allow the programmatical definition of options, and it is hoped that this can be directly incorporated into OpenMDAO.

The capability to take any OpenMDAO assembly and generate a description of its state in an OpenMDAO independent format, and recreate it, enables many new use cases, several of which have been demonstrated in the examples provided via GitHub. Replacing a component or group with another component is expected to one of the core use cases of the standard-evaluator library.

Releasing the library as open-source and making the source code available on GitHub is key to help adoption of this functionality and ensure continued development.

Additional work that needs to be done is to add the ability to handle assemblies with unique features that are not yet supported by the standard-evaluator. Examples are how to deal with custom groups, how to deal with options that are complex and not serializable, and mapping of elements of vector variables or responses.

Integrating these capabilities with the Aircraft Data Hierarchy developed under CLIN001 and the integration with the MBSE models that was developed in CLIN003 are additional work that should be done.

A larger effort would be required to create a GUI to edit an assembly, and to generate new assemblies from a catalog of components. This would open the potential user base of OpenMDAO to engineers that are not versed in programming.

Building converter that for example take an MDA developed in Ansys ModelCenter and convert it into the assembly definition and then from there into an OpenMDAO assembly and back would allow engineers to leverage the best of multiple frameworks, reduce time when transitioning, and avoid vendor lock-in.

One of the key extensions to the Assembly definition would be to define a class describing the information about how to use an external executable as a component.

Coordination with the OpenMDAO team has started to see what functionality of the standard-evaluator can be directly integrated into OpenMDAO.

12. References

A. Page Risueño, J. B. (2020). MDAx: Agile Generation of Collaborative MDAO Workflows for Complex Systems. *AIAA Aviation Forum.* Virtual Event: AIAA. doi:10.2514/6.2020-3133

Carl Recine, J. K. (2024). *NASA's Aviary Takes Flight: A Public Software for Aircraft Design.* Mountain View, CA: NASA. Retrieved from https://ntrs.nasa.gov/citations/20240009217

Imco van Gent, G. L. (2018). CMDOWS: a proposed new standard to store and exchange MDO systems. *CEAS Aeronautical Journal, 9*, 607-627. Retrieved from https://link.springer.com/article/10.1007/s13272-018-0307-2#Sec5

J. S. Gray, J. T. (2019, April). OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization, 59*(4), 1075-1104. Retrieved from https://link.springer.com/article/10.1007/s00158-019-02211-z