A Standard Platform for Testing and Comparison of MDAO Architectures

Justin Gray,* Kenneth T. Moore,† Tristan A. Hearn,‡ Bret A. Naylor§
NASA Glenn Research Center, Cleveland, OH

This manuscript is submitted for consideration for the Special Section on MDO.

The Multidisciplinary Design Analysis and Optimization (MDAO) community has developed a multitude of algorithms and techniques, called architectures, for performing optimizations on complex engineering systems which involve coupling between multiple discipline analyses. These architectures seek to efficiently handle optimizations with computationally expensive analyses including multiple disciplines. We propose a new testing procedure that can provide a quantitative and qualitative means of comparison among architectures. The proposed test procedure is implemented within the open source framework, OpenMDAO, and comparative results are presented for five well-known architectures: MDF, IDF, CO, BLISS, and BLISS-2000. We also demonstrate how using open source software development methods can allow the MDAO community to submit new problems and architectures to keep the test suite relevant.

Nomenclature

AAO	All At Once
BLISS	Bi-Level Integrated Systems Synthesis
CO	Collaborative Optimization
COBYLA	Constrained Optimization By Linear Approximation
DOE	Design of Experiments
IDF	Individual Design Feasible
MDA	Multidisciplinary Analysis
MDAO	Multidisciplinary Design Analysis and Optimization
MDF	Multiple Design Feasible
SAND	Simultaneous Analysis and Design
SLSQP	Sequential Least Squares Quadratic Programming
XDSM	Extended Design Structure Matrix

I. Introduction

The ultimate goal of any Multidisciplinary Design Analysis and Optimization (MDAO) architecture is no different than that of any traditional optimization: to find the best solution possible in a given design space, subject to all specified constraints. The true challenge that research into MDAO seeks to address, however, lies in the ever increasing complexity and computational cost of modern engineering design tools. With the advent of Computational Fluid Dynamics (CFD) and Finite Element Analysis (FEA) a single analysis can easily take several hours or more to run. Systems analysis frameworks seek to integrate large numbers of different analysis tools from a wide range of engineering disciplines. Thus, it is increasingly important that any optimization accounts for the problems associated with modern analyses.

*Aerospace Engineer, MDAO Branch, Mail Stop 5-11, AIAA Member
†Senior Systems Engineer, MDAO Branch, Mail Stop 500-105, AIAA Senior Member
‡Aerospace Engineer, MDAO Branch, Mail Stop 5-10, AIAA Member
§Senior Software Engineer, MDAO Branch, Mail Stop 500-105

To complicate things further, in many cases, analyses for complex systems can display such significant interactions between their components that it becomes necessary to model them in a highly coupled manner. These couplings present an even greater challenge, because system compatibility needs to be maintained at the same time as minimization of the objective function . Lastly, there is the problem of how to efficiently handle optimization where the coupled discipline analyses can have dramatically different run times. This is usually the case when trying to do analysis with coupled high-fidelity and low-fidelity tools.

The MDAO community has developed a number of decomposition-based MDAO optimization algorithms, called MDAO architectures to help tackle the issues identified above. A large number of MDAO architectures have been proposed, implemented, and tested on a wide variety of problems. A survey by Martins identified 13 different architectures found in the literature.[?] However, having so many architectures makes it difficult for an engineer to select the appropriate one for an analysis. Traditionally, this selection has been made somewhat informally, with a tendency to rely on architectures that are already familiar to the researcher.[?] However, a number of efforts have provided comparative results among architectures which could be used to make a more informed decision.[?,?,?] Hulme and Bloembaum developed a testing system called CASCADE which was designed to generate lots of optimization test problems, and they used it to compare the performance of the IDF, MDF and AAO architectures.[?] Most recently Martins et al. developed a prototype MDO framework, πMDO, which was designed specifically to enable the rapid application of MDAO architectures to engineering problems.[?] Tedford and Martins used πMDO to implement a number of architectures and compare their performance on a set of test problems.[?] In their comparison made among IDF, MDF, SAND, CO,and CSSO it was determined that IDF and SAND were the best performing architectures, but that this result was specific to the problems they tested. In related work, Marriage and Martins identified a specific problem where CO outperformed MDF[?] due to a specific problem structure where two disciplines were very highly coupled but the rest were not. Considering these two results together it is clear that architecture performance is problem dependent. Also, Tedford and Martins acknowledge that their analysis was performed using only numerical differentiation techniques and that if analytic derivatives were provided for the disciplines, the results could have been different. Many new architectures have been developed and modifications to existing architectures, have been proposed but standardized performance benchmarking on these has not yet been done.

Tedford and Martins' work compared architecture performance across three different test problems. All three of these problems were mathematical implementations. Padula et al. proposed a number of different possible MDAO test problems in the MDAO test suite.[?] This work provided problem statements and some Fortran source files, via a website, for 13 different problems. Some of the problems were not directly implementable without access to complex analytical models and specialized analysis tools. Other problems were posed in such a way as to be solved by a specific MDAO architecture and were not general enough to be used in studies with multiple architectures. Regardless, work on the MDAO Test Suite established the foundation for a publicly available test suite of problems which the community could use.

So from previous research, we conclude that it is necessary to consider a number of test problems which exhibit characteristics similar to those expected for the real analyses in order to characterize the performance of any given architecture. Furthermore, it is possible that different implementations of the same architecture need to be compared as well.[?,?] The work with πMDO established the foundation for a test procedure by demonstrating that architectures could be automatically applied to a given set of problems. Despite these efforts, no common testing platform has emerged, and a standard test procedure has not been adopted to measure the performance of new architectures. Nor has an exhaustive comparison been performed using the whole range of test problems that have been proposed.

Therefore, we see the following necessary challenges which need to be addressed in order to successfully develop an effective testing platform.

1. Selecting a common software framework on top of which the MDAO community could develop and share a testing platform

2. Constructing a large set of MDAO architectures which are implemented as efficiently as possible

3. Compiling a suite of test problems which have properties that simulate a wide range of engineering analyses and problem scales

4. Ensuring that new test problems and architectures, as well as improvments, can be added into the test suite by the community

In this work we propose the use of an open source engineering framework, OpenMDAO, as the shared platform to address all four of these issues. Within OpenMDAO we have implemented some of the most commonly investigated MDAO architectures and collected a set of test problems from the available literature. Using an automated test process every problem was optimized using each one of the architectures, and the results are presented. This collection of architectures and test problems is not intended to compose a complete test suite. Rather, it serves as an initial test suite that proves the viability of OpenMDAO as a testing platform. By leveraging open-source software development methods, we show how OpenMDAO will provide an avenue for community contribution of new test problems and MDAO architectures so that the test suite can evolve to be more relevant and applicable to engineering problems.

II. Common Software Framework

The OpenMDAO development effort was started by the NASA Subsonic Fixed Wing project under the Fundamental Aeronautics Program to ensure that NASA has the necessary software framework to make effective use of MDAO for the examination of unconventional aviation concepts such as the Hybrid Wing Body aircraft.[?,?] It was recognized that an effective MDAO framework needed to have a wide user base and would require continued collaboration from industry and academia to ensure that new MDAO research was constantly being utilized. For this reason OpenMDAO was designed from the beginning to be an open source software development effort.[?]

OpenMDAO is built on top of the Python programming language[?] and is designed to provide full support for the rapid implementation and application of MDAO architectures. OpenMDAO is designed around a black-box approach to component analyses.[?,?] Analyses are broken down into a number of smaller parts, and each one is modeled as a black-box with inputs and outputs. The boxes are then strung together to build the full analysis. OpenMDAO provides four main classes to support this type of functionality: *Component, Driver, Workflow,* and *Assembly.*

Any given engineering problem would be composed of a set of *Component* and *Assembly* instances. To solve a problem a user would arrange a set of *Driver* and *Workflow* instances to implement some type of optimization algorithm or MDAO architecture. A short description of each class is presented below.

A. Fundamental Classes

1. Component Class

A unit of computational work is represented by the *Component* class in the framework. *Component* instances have input and output variables and perform calculations when executed. Figure 1 gives a conceptual view of what a simple Component looks like.

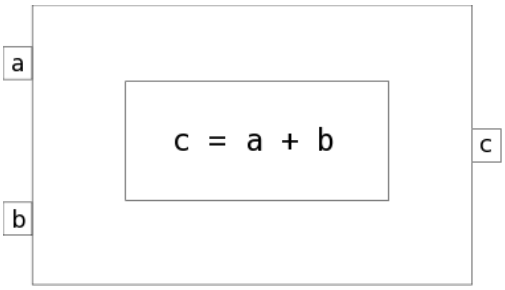

Figure 1: Conceptual view of a simple component. a, b, and c are all variables.

Some components may be written purely in Python code. Native Python components are very easy to construct and provide the ability for users to rapidly implement new analysis tools.

Other components may be comprised of a Python wrapper for a code written in another language, such as Fortran, C, or C++.[?] You can also convert a native Python component to a wrapped component using a small amount of compiled code to handle the most computationally expensive part of calculations to get some performance improvements. In the *OpenMDAO User Guide*, a piston engine model is created as a native Python component. Then that analysis is converted into a compiled executable written in C and

wrapped in Python to make a compiled component. The result of the complied *Component* is a roughly 800% increase in computation speed.[?]

2. Driver Class

Drivers are used to perform any iterative task (e.g., optimization, convergence, or design of experiments). OpenMDAO currently provides a number of drivers in its standard library. It also provides full support for users to provide new drivers to meet their own needs.

To make it simple to exchange one driver for another in an analysis, it was necessary to establish a uniform interface, or application programming interface (API), that all drivers could support. Previous research by Perez et al. focused on developing a single uniform API for all optimizers to utilize.[?] OpenMDAO extends that concept to provide a common API for all drivers.

3. Workflow Class

A *Driver* instance iterates by executing a set of *Component* instances repetitively. Drivers are associated with instances of *Workflow* that specify which components to execute and the order to execute them. Although in many cases a workflow contains just basic components, it can also contain other drivers. This allows nested iterative processes to be created. Nested iterations provides the flexibility needed to build complex optimization processes defined by MDAO architectures. Components are allowed to show up multiple times in a single workflow or in multiple parts of a nested workflow. This can be used, for example, to train a metamodel in one part of a workflow and then optimize it in another.

4. Assembly Class

Instances of the *Assembly* class contain groups of components and drivers that compose an analysis. *Assembly* instances can have their own input and output and, like drivers, can also be included in workflows. This allows assemblies to be included in models with other components. This capability allows nested models to be created by having complex simulations contained within an assembly, e.g. a turbine engine simulation in an Assembly which is used as a component in an aircraft simulation.

B. Problem Formulation

Like traditional optimization problems, MDAO problems can be represented by a fundamental problem formulation which describes the goals of the optimization. This fundamental formulation is comprised of a set of six things:

1. Local design variables
2. Global design variables
3. Objective(s)
4. Constraints
5. Coupling variable pairs
6. Analysis components

More formally this formulation can be represented as:

$$\begin{aligned} min \quad & f(x, y(x)) \\ w.r.t. \quad & x \\ s.t. \quad & g(x, y(x)) \geq 0 \\ & y_i(x) = y_j(x) \; for \; i \neq \; j \end{aligned} \tag{1}$$

Each coupling variable pair includes both an input to a given component and the output from another component that must be consistent (i.e. $y_i(x) = y_j(x)$) at a converged solution.

It could be argued that there is no real distinction between local and global design variables, because you can infer that a global exists by the occurrence of the same variable name in multiple disciplines. In theory this works, but in practice different analysis tools will give different variable names to the same physical quantity. In the interest of solving this practical issue OpenMDAO requires explicit definition of local design variables and global design variables

Some of the problem formulations in the literature also include the specification of disciplinary state variables and residual functions of those state variables. In these cases, disciplinary codes are relying on external solvers or optimizers to bring them into a self-consistent state (i.e. drive their residuals to zero). Only the All At Once (AAO) and the Simultaneous Analysis and Design (SAND) architectures directly deal with state variables and residuals. In all other cases, an additional solver needs to be added to drive the disciplines to consistency. Since neither AAO or SAND is considered for this work, for the sake of simplicity we leave state variables and residuals out of the fundamental formulation. We do acknowledge that future studies may require them though.

1. OptProblem Class

OpenMDAO provides a subclass of *Assembly*, called *OptProblem*, which supports the explicit definition of the above six elements of a problem formulation. An instance of *OptProblem* includes all the necessary components, or analysis codes, to perform an optimization, as well as a complete definition of the problem formulation.

In addition to the information regarding the problem formulation, *OptProblem* instances also allow for the specification of the optimal solution. This includes the optimal values for all design variables, coupling variables, and objectives. The inclusion of this information allows for a direct scoring of the performance of any MDAO architecture relative to the optimal solution. All the test problems implemented for this work were written as subclasses of *OptProblem* in the optproblems section of the OpenMDAO standard library.

2. Architecture Class

An MDAO architecture derives a new problem formulation from the fundamental one. This new problem formulation is composed of one or more optimization problems which when solved will yield an answer that also minimizes the fundamental problem formulation. The new formulation, in addition to the existing design variables, objectives, and constraints, may also have new variables, objectives, and constraints as well. For example, the top level optimization in CO adds a compatibility constraint to minimize the difference between the design variables in the sub-optimizations and the targets in the global optimization The goal of this refactoring is to provide a more effective (e.g. fewer function calls, greater stability, lower objective, etc.) optimization.

In OpenMDAO the reformulation is implemented by building up a set of drivers and associated workflows. For some simple MDAO architectures, this would be fairly straightforward to do. However, in many situations, the initial setup can be difficult because the reformulated problem looks vastly different from the original and can have many different steps. To simplify the process, OpenMDAO provides the *Architecture* class which defines an automatic process that builds up the new problem formulation based on an inspection of the problem formulation from a specific *Optproblem* instance. All of the architectures investigated here are coded as sub-classes of *Architecture* in the architectures section of the OpenMDAO standard library.

An *Architecture* builds up a new problem formulation by instantiating new drivers and then organizing them with workflows to build an algorithmic representation of the architecture's mathematical formulation. In other words, an architecture instance just arranges a specific set of *Driver, Component, Assembly* and *Workflow* instances to represent the decomposed problem. After the process is configured, the users are free to make modifications to suit their needs. For instance, users could change the optimizer from a Sequential Quadratic Programming algorithm to Conjugate Gradient algorithm. In some sense it is correct to think about a specific architecture implementation as defining a recommended or initial configuration. If the specifics of a problem demand a deviation then the change can be made easily after initial configuration.

Although this kind of flexibility is valuable for real-world problems, from the point of view of a test suite it is not a realistic approach. For this work, no deviations to the default configuration of any architecture were made before testing an architecture on any given test problem.

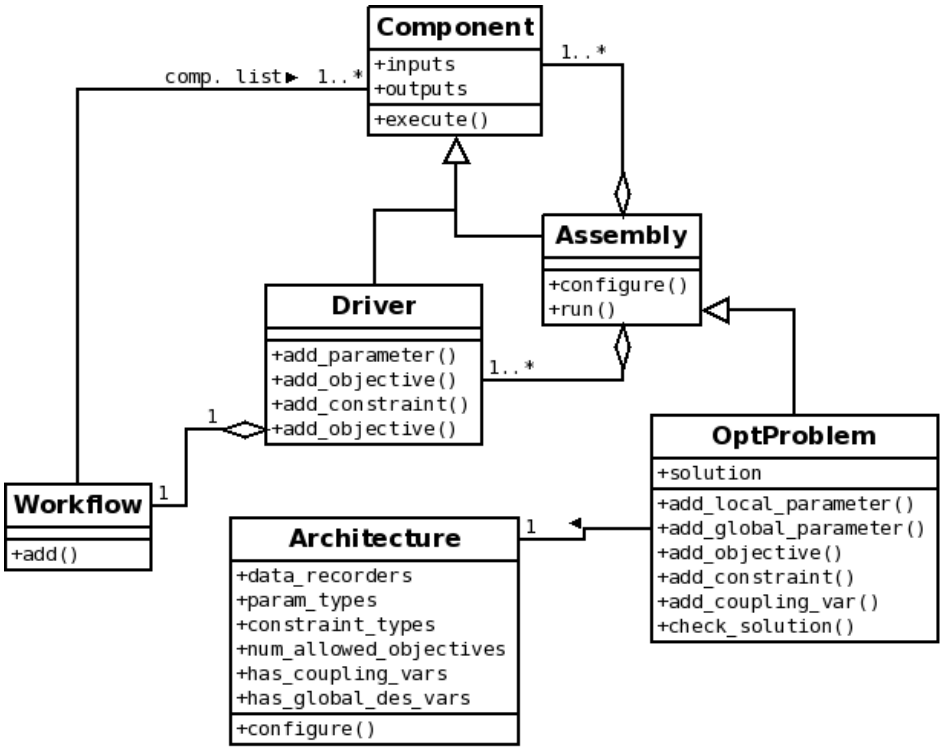

Figure 2: UML class diagram for the OpenMDAO classes that support MDAO architectures.

III. Architectures

The architecture implementations used here are not presented as definitive implementations of each architecture. While they do follow the formulations spelled out by their creators, their primary purpose is to demonstrate a wide range of MDAO architecture implementations in OpenMDAO. They serve as templates for future researchers to work from when building new and improved MDAO architectures. When taken as a set these architectures demonstrate the flexibility of OpenMDAO and provide examples of the fundamental elements necessary to create any other MDAO architecture. There are examples of nested workflows, training of metamodels with a DOE, optimization of metamodels, and convergence loops.

By default of all the architectures in this research used the the SLSQP[?] optimizer. Usage of a single optimizer allowed for the direct comparison of architecture performance. However, for completeness, some data was also collected using the COBYLA[?] optimizer. It should also be noted that in the implementation of each architecture there are a number of parameters that can be adjusted to fine tune its performance. Some of these parameters are inherent to the optimizers used in each implementation. Other parameters, such as the ones in BLISS and BLISS-2000, are fundamental features of the architecture itself. For each architecture, specific values for these parameters were selected and then held fixed.

Below are descriptions of the process and formulation for each of the 5 architectures implemented. The notation used to describe the formulations is described in Table 1. The workflows for each architecture are defined using the Extended Design Structure Matrix (XDSM) notation defined by Lambe and Martins.[?] for all the test problems. XDSM diagrams describe both data flow and process flow, so they provide a complete description of the algorithm. The thin-black lines in the diagram describe process flow, indicating what order the blocks get executed in. The thick-grey lines describe the movement of data, with vertical lines indicating inputs to a given block and horizontal lines indicating outputs. All of the parallelogram blocks are data-blocks, representing variables. All other blocks represent components or drivers in the analysis. When any given block is shown stacked up , and has an i in the title (e.g. Analysis i), that indicates that n such blocks could exist and could be run in parallel if desired. Each step in the process is given a numeric label (the first step in the process is always 0), which applies to both process both process flow and data flow. For process flow, the labels are used to indicate loops (e.g. solver loops, optimizations). For example in Fig. 3 the optimization loop is given the label "$0, 3 \rightarrow 1$". This indicates that starting at 0, you follow

the path through from 1 to 2 to 3 and then step 3 loops back through step 1 until an optimum is reached. The numeric labels in the data-blocks indicate during which step the data is either input to or output from the block.

Table 1: Notation for the description of MDAO problem formulations.

Symbol	Definition
x	Vector of design variables
y	Vector of coupling variable responses (outputs from a discipline analysis)
y^t	Vector of coupling variable targets (inputs to a discipline analysis)
x^t	Design variable target values, created as extra inputs for certain architectures
f	Objective function
g	Constraint function
N	Number of disciplines
$()_0$	Functions or variables that are shared by more than one discipline (global)
$()_i$	Functions or variables that apply only to discipline i (local)
$()^*$	Functions or variables at their optimal value
$\tilde{()}$	Approximation of a given function or vector of functions
$()^c$	Functions or variables related to coupling variables

A. Individual Design Feasible (IDF)

This is one of the simplest architectures. It uses a single optimizer to drive the whole process. To ensure that the coupled system is consistent, IDF adds one equality constraint per set of coupling variables in the original formulation. The XDSM for IDF is shown in Fig. 3.

The problem formulation is as follows:

$$\begin{aligned} min \quad & f_0(x, y(x)) \\ w.r.t. \quad & x \\ s.t. \quad & g_0(x, y(x)) \geq 0 \\ & g_i(x, y(x)) \geq 0 \ for \ i...N \\ & g_i^c(x, y(x)) = y_i^t - y_i(x) = 0 \ for \ i...N \end{aligned} \tag{2}$$

B. MultiDisciplinary Feasible (MDF)

Like IDF, MDF does not modify the problem formulation at all. The primary distinction is in the handling of the coupling variables. System coupling is handled by a solver which drives the system to compatibility for every iteration of the optimizer. In MDF the optimizer never sees anything except a compatible system. The XDSM formulation for MDF is shown in Fig. 4.

In Fig. 4, the loop $3 \rightarrow 1$ represents the MultiDisciplinary Analysis (MDA). In this implementation convergence is accomplished using the *BroydenSolver* in the OpenMDAO standard library. However, that driver could be replaced with either *FixedPointIterator* from the standard library or any other solver a user wished to specify at run time.

$$\begin{aligned} min \quad & f_0(x, y(x)) \\ w.r.t. \quad & x \\ s.t. \quad & g_0(x, y(x)) \geq 0 \\ & g_i(x, y(x)) \geq 0 \ for \ i...N \end{aligned} \tag{3}$$

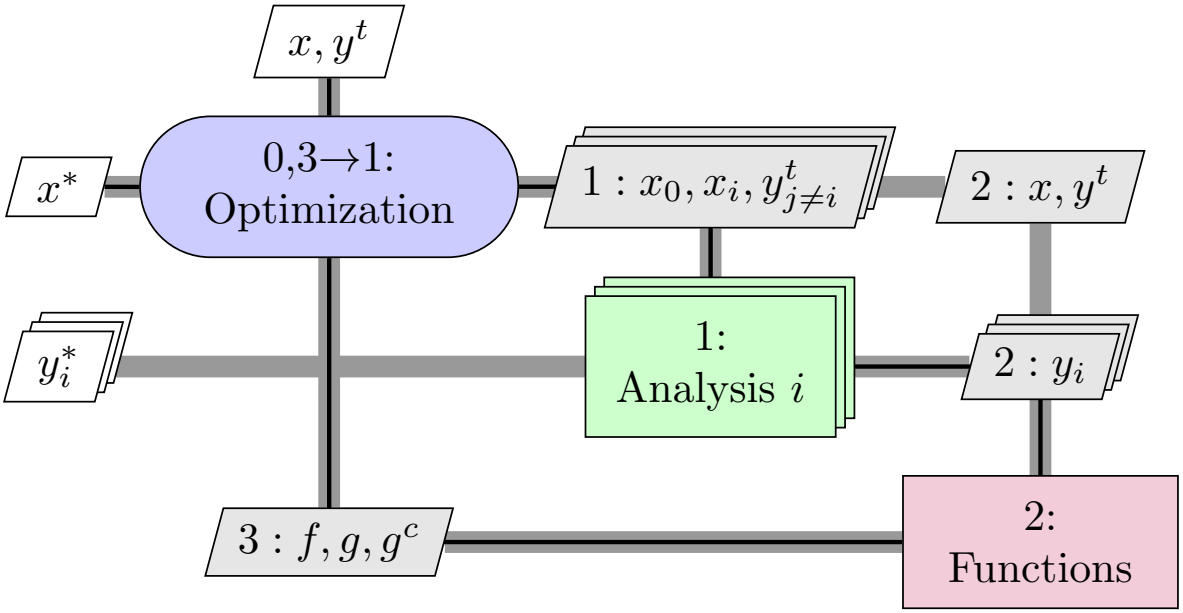

Figure 3: XDSM diagram for IDF

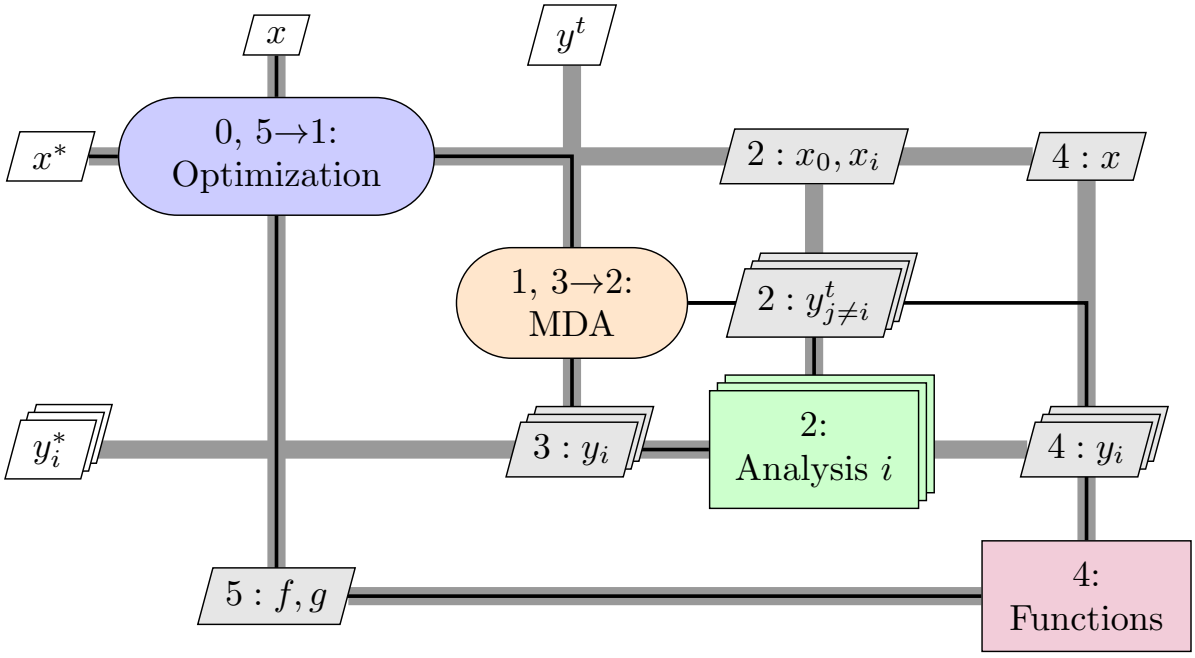

Figure 4: XDSM diagram for MDF

C. Collaborative Optimization (CO)

CO decomposes a problem into separate global and local optimizations. One optimizer is used at the global level, and then one additional optimizer is employed for each individual discipline. Since each discipline is optimized independently, a number of new target variables must be introduced for each coupling variable at the global level. Likewise a new constraint must be introduced for each discipline that drives the residual between the target variable and the real variable to zero. In addition, target variables are created for any local variables which show up explicitly in the objective function, and their residuals are included in the respective discipline's residual constraint. The XDSM for CO is shown in Fig. 5.

In Fig. 5 the loop from $1.3 \rightarrow 1.1$ demonstrates a basic nested optimization loop. This type of workflow is trivially generated by adding the sub-optimizers to the workflow of the global optimizer in OpenMDAO.

For the global optimization the problem formulation is as follows:

$$\begin{aligned} min \quad & f_0(x_0, x^t, y^t) \\ w.r.t. \quad & x_0, x^t, y^t \\ s.t. \quad & g_0(x_0, x^t, y^t) \geq 0 \\ & g_i^* = ||x_{0i}^t - x_0||_2^2 + ||x_i^t - x_i||_2^2 + \\ & ||y_i^t - y_i(x_{0i}, x_i, y_{j\neq i}^t)||_2^2 = 0 \ for \ i...N \end{aligned} \tag{4}$$

For the local optimizations the problem formulation is as follows:

$$\begin{aligned} min \quad & g_i \\ w.r.t. \quad & x_{0i}^t, x_i \\ s.t. \quad & g_i(x_{0i}^t, x_i, y_i(x_{0i}^t, x_i, y_{j\neq i}^t)) \geq 0 \end{aligned} \tag{5}$$

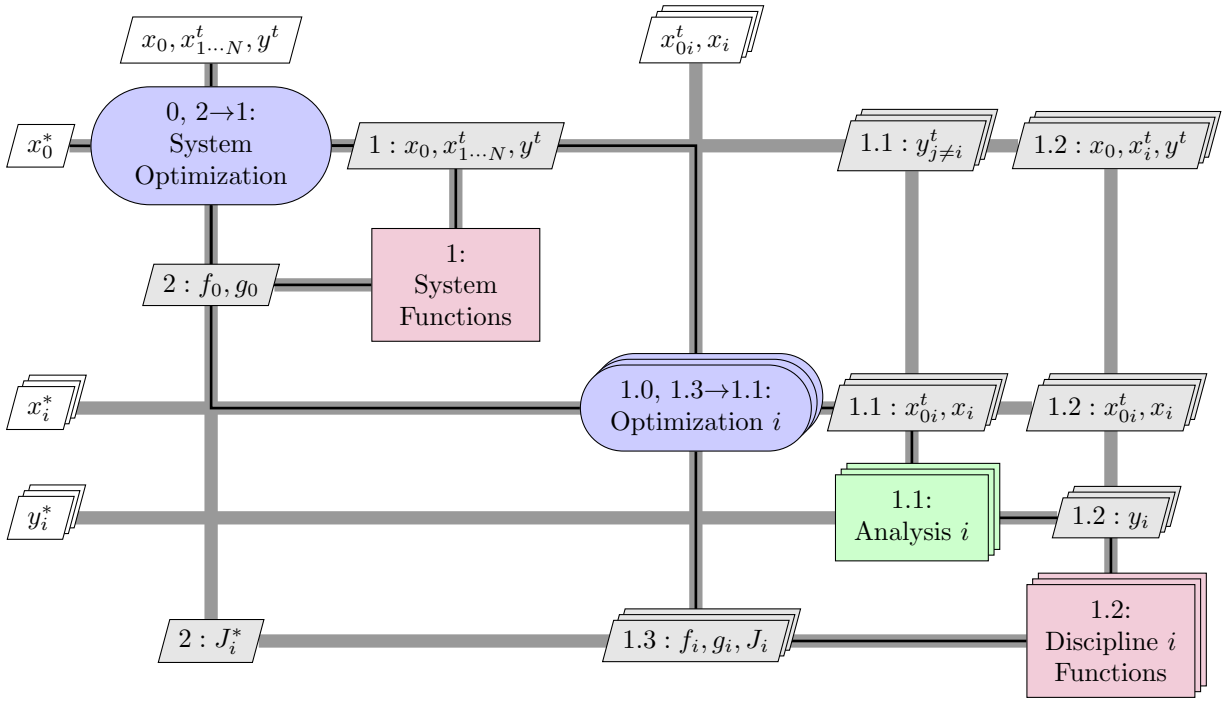

Figure 5: XDSM diagram for CO

D. Bi-Level Integrated Systems Synthesis (BLISS)

BLISS operates on a series of linear approximations of the actual objective function and constraints. To get those approximations the architecture can use either a numerical finite difference engine or analytic

derivatives. For the system level problem, sensitivities are only taken with respect to global variables. Likewise, for the discipline level problem sensitivities are only taken with respect to local design variables for that specific discipline.

The general process is to generate a linearized approximation of the system, optimize on that and then generate a new linearized approximation at the optimum point. Target variables are created for all the design variables, and a fixed-point iteration is used to converge the targets with the design variables. So the defining feature is that no optimization ever occurs directly on the actual discipline analyses, only on the approximate models. BLISS uses move limits to constrain the distance the optimization can move during one iteration. To enforce system compatibility, BLISS uses a solver to perform an MDA for each major iteration. Hence the coupling variables do not show up in the problem formulation. The XDSM for BLISS is shown in Fig. 6.

Steps 5 and 8 from Fig. 6 indicate the explicit calculation of discipline and system level sensitivities. These sensitivities are then used to construct a linearized model which is optimized. To calculate these sensitivities we used an instance of *SensitivityDriver*, from the driver section of the OpenMDAO standard library. This driver is configured with the same parameters, objectives, and constraints as its associated optimizer/solver in workflow. *SensitivityDriver* makes use of OpenMDAO's built in derivative calculation capabilities in order to automatically calculate the derivatives of all objectives and constraints with respect to all parameters. If any of the components provide analytic derivatives, those are used automatically instead of running finite difference calculations.

The problem formulation for the system level is as follows:

$$\begin{aligned} min \quad & (f_0^*)_0 + \left(\frac{df_0^*}{dx_0}\right)(x_0^t - x_0) \\ w.r.t. \quad & x_0^t \\ s.t. \quad & (g_0^*)_0 + \left(\frac{dg_0^*}{dx_0}\right)(x_0^t - x_0) \geq 0 \\ & (g_i^*)_0 + \left(\frac{dg_i^*}{dx_0}\right)(x_0^t - x_0) \geq 0 \ for \ i...N \\ & |x_0^t - x_0| \leq \Delta x_{0limit} \end{aligned} \tag{6}$$

The discipline level problem formulation is as follows:

$$\begin{aligned} min \quad & (f_0)_0 + \left(\frac{df_0}{dx_i}\right)(x_i^t - x_i) \\ w.r.t. \quad & x_i^t \\ s.t. \quad & (g_0)_0 + \left(\frac{dg_0}{dx_i}\right)(x_i^t - x_i) \geq 0 \\ & (g_i)_0 + \left(\frac{dg_i}{dx_i}\right)(x_i^t - x_i) \geq 0 \ for \ i...N \\ & |x_i^t - x_i| \leq \Delta x_{ilimit} \end{aligned} \tag{7}$$

E. Bi-Level Integrated Systems Synthesis 2000 (BLISS-2000)

BLISS-2000 is a a reformulation of the original BLISS algorithm developed to eliminate the need for calculating sensitivities on the MDA.[?] BLISS-2000 does not perform an MDA at all. It uses an IDF-like formulation to drive the system level problem, which is run on quadratic response surface approximations of the system. The disciplines are each then optimized directly with respect to their local variables and constraints. The XDSM for BLISS-2000 is shown in Fig. 7.

As seen in Fig. 7, in step 5 of the process, a metamodel for each of the i disciplines must be created based on training data. This data is collected by executing a LatinHypercube Design of Experiments (DOE), where the number of points executed are governed by Eqn. 8 with n being the number of design variables, in a neighborhood around the current point.[?]

$$\frac{n^2 + 3n + 2}{2} \tag{8}$$

To accomplish this, a specific sub-class of *DOEdriver* was added to the standard library called *NeighborhoodDOEdriver*. During every major iteration a DOE must be re-run around the current global design point for each discipline and then a new metamodel must be trained from that data.

For the global optimization the problem formulation is as follows:

$$\begin{aligned} min \quad & f_0(x, \tilde{y}(x, y^t)) \\ w.r.t. \quad & x_0, y^t \\ s.t. \quad & c_0(x, \tilde{y}(x, y^t)) \geq 0 \\ & g_i = y_i^t - \tilde{y}_i(x_{0i}, x_i, y_{j\neq i}^t) = 0 \ for \ i...N \end{aligned} \tag{9}$$

For each discipline the problem formulation is as follows:

$$\begin{aligned} min \quad & \sum y_i \\ w.r.t. \quad & x_i \\ s.t. \quad & c_i(x_0, x_i, y(x_0, x_i, y_{j\neq i})) \geq 0 \ for \ i...N \end{aligned} \tag{10}$$

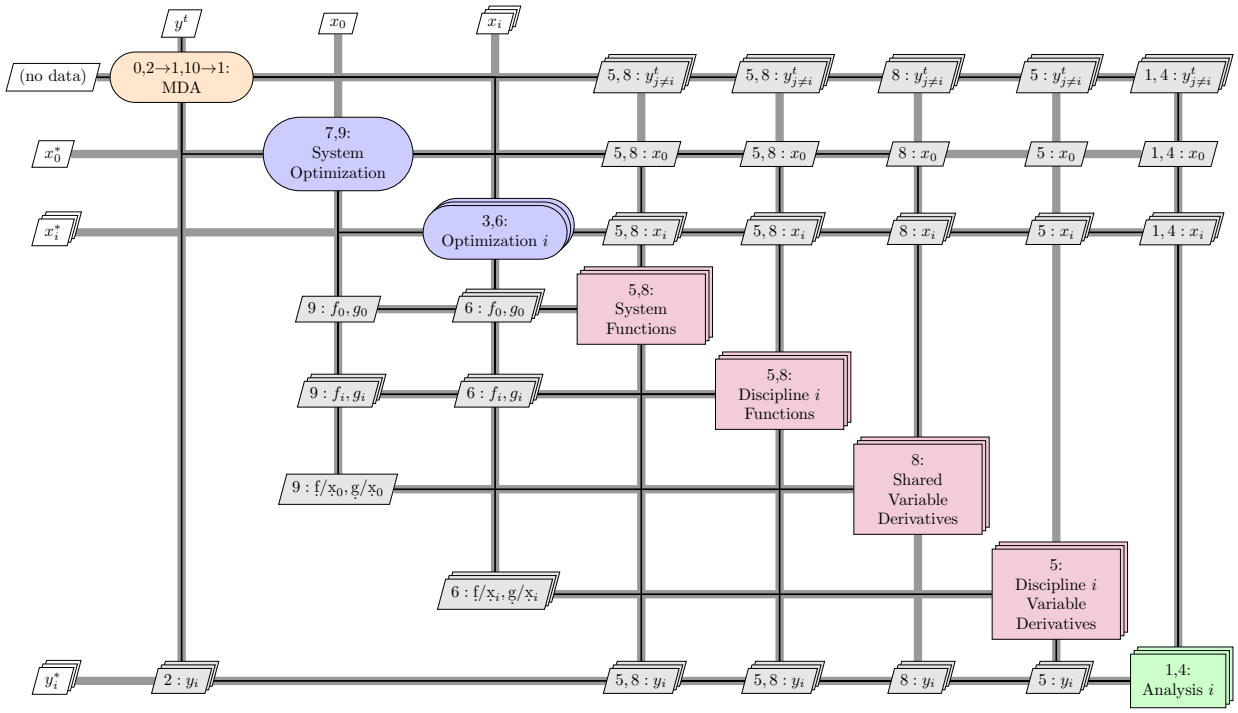

Figure 6: XDSM diagram for BLISS

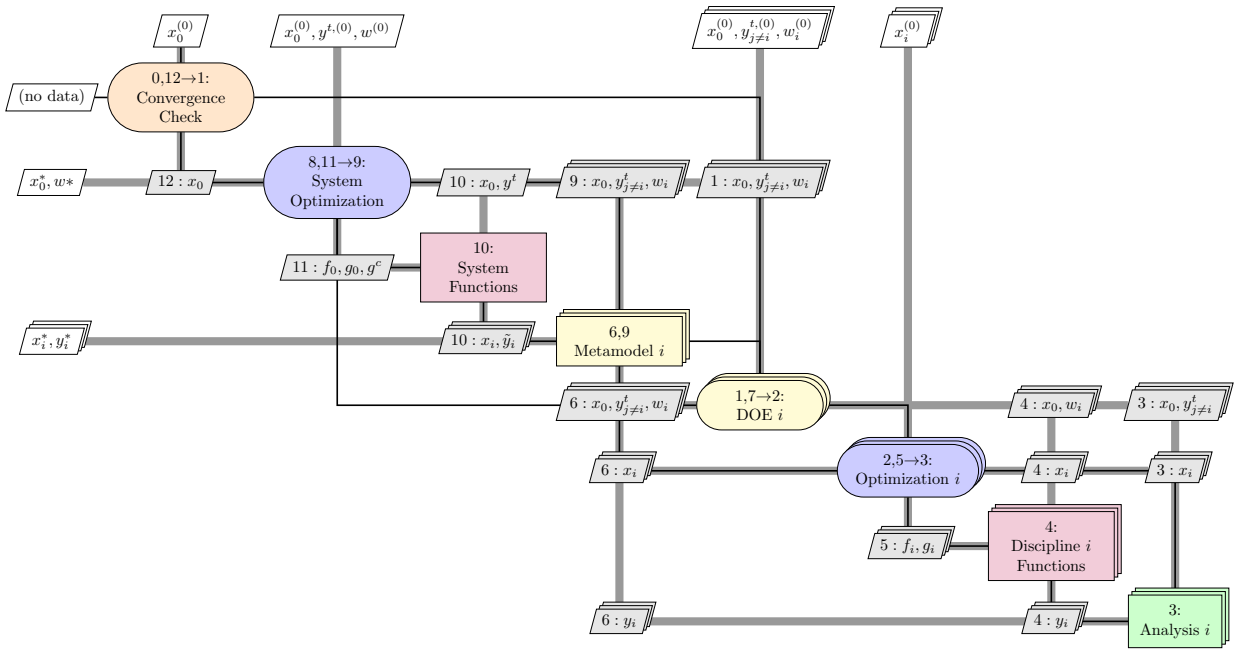

Figure 7: XDSM diagram for BLISS-2000

IV. Test Problems

Using a common set of test problem implementations – not just a common set of test problem formulations – provides two key advantages for researchers. Firstly, it dramatically reduces the time necessary to start testing new algorithms since no implementation of the test problems is necessary. With only two test problems, the benefit is somewhat small; however, as the test problem suite grows in size and complexity the time savings grow with it. Secondly, using the same implementations for the test problems gives a basis for true apples-to-apples comparison between results. There can be no question of the results on grounds of differences or problems in the implementations.

This section presents the fundamental problem formulation for the two test problems used for this work. These problems do not comprise a comprehensive problem set. But they demonstrate the way in which more complex and realistic problems could be implemented in OpenMDAO with complete problem formulations and initial conditions being specified. Complete implementation details are not fully described here, but have been included in the OpenMDAO version 0.2.5 (or later) software distribution. The problems are included in the optproblems module in the standard library. They are available by downloading the framework.

A. Sellar Problem

This algebraic problem was introduced by Sellar et al. in 1996.[?] It has since become a commonly used test problem for MDAO architectures.[?,?] The problem is fairly small, having two disciplines and a limited number of design variables, but it does provide some behaviors that mimic larger more realistic problems. Hence it is an ideal test case on which to perform benchmarks. At the very least, the simplicity of Sellar Problem allows for it to be solved by any effective architecture. The problem formulation is given in Eq. 11 and parameter values for the initial and known optimal solution are given in Table 2.

$$
\begin{aligned}
min \quad & x_1^2 + z_2 + y_1 + e^{-y_2} \\
w.r.t. \quad & z_1, z_2, x_1 \\
s.t. \quad & 1 - \frac{y_1}{3.16} \leq 0 \\
& \frac{y_2}{24} - 1 \leq 0 \\
& -10 \leq z_1 \leq 10 \\
& 0 \leq z_2 \leq 10 \\
& 0 \leq x_1 \leq 10
\end{aligned}
\tag{11}
$$

Table 2: Initial conditions and known optimal solution for the Sellar Problem

	Initial	Optimal
z_1	5.000	1.978
z_2	2.000	0.000
x_1	1.000	0.000
y_1	0.000	3.160
y_2	0.000	3.756
objective	31.001	3.183

In addition to the standard version for the Sellar Problem a second version that includes the use of analytic derivatives was also implemented. The problem formulation remains exactly the same between the two versions. Derivatives for both y_1 and y_2 as a function of all inputs were specified, and the framework automatically makes use of them. This allows for performance comparisons between two identical problems, where one uses finite differences and the other uses analytic derivatives.

B. Scalable Problem

This problem was introduced by Martins et al. in 2002.[?] It provides the ability to change the number of local design variables, global design variables, disciplines, and the degree of coupling between them to any size the user desires. The Scalable Problem is designed to allow investigation of how architectures scale to larger-sized problems without adding too much computational burden. The problem has a quadratic objective function and linear dependence between the disciplines. The general problem formulation is given in Eq. (12). Equation (13) gives the governing equation for each discipline, where C_z, C_{x_i}, and C_{y_j} are all matrices of positive coefficients.

$$
\begin{aligned}
min \quad & z^T z + \sum_i^N y_i^T y_i \\
w.r.t. \quad & z, x \\
s.t. \quad & 1 - \frac{y_i}{C_i} \leq 0, i = 1, ..., N \\
& -10 \leq z \leq 10 \\
& -10 \leq x \leq 10
\end{aligned}
\tag{12}
$$

$$
y_i(z, x_i, y_j) = -\frac{1}{C_{y_i}} \left(C_z z + C_{x_i} x_i - C_{y_j} y_j \right) \tag{13}
$$

For this work, the problem was configured with three disciplines, each having three local design variables, three global design variables, and three output state variable. Each discipline was coupled to the two other disciplines. C_z and C_{x_i}, were each defined as 3×3 matrices of ones. C_{y_j} was defined as a 3×3 Identity

matrix. The outputs of each discipline, were scaled with appropriate values of C_{y_i} so that at the optimum solution their values would all be 1. This creates a more complex problem than Sellar with a difficult coupling challenge. We named this form of the Scalable Problem the Unit Scalable Problem. The initial state and known optimal state for the Unit Scalable Problem are given in Table 3.

Table 3: Initial conditions and known optimal solution for the Scalable Problem

	Initial	Optimal
z	-1.0	0.000
x	-1.0	-0.666
y	0.000	1.000
objective	40.000	3.000

V. Test Results

There are a number of different ways to measure the effectiveness of a given MDAO architecture. The most obvious way is simply to compare optimal objective value found to the known optimum objective. Assuming that one architecture finds a lower objective than the other, is it then considered more effective? The reality of the situation is not that simple. Other relevant pieces of information need to be considered. For instance, how many function evaluations did each architecture make for the disciplines? Did the lower objective function value come at a higher computational cost? Along a similar line of reasoning, it might be important to consider the ability of an architecture to take advantage of parallelization. Different potential users will want to analyze performance from a number of different metrics, all of which should be calculated automatically when running the MDAO test suite. To allow for this OpenMDAO has a section of the test suite where new metrics can be added. As with architectures and test problems, community contribution can help provide the most complete set of test metrics and ensure that new metrics are added to keep the measurement set relevant. For this work we considered three different performance metrics:

1. Proximity to known optimal solution
2. Total function evaluations for each discipline
3. Convergence characteristics

A. Proximity to Known Optimal Solution

All optimization test problems in the OpenMDAO framework are required to include a complete specification of the known optimal solution. To enforce this requirement, OpenMDAO includes a specific test in its unit test suite which checks the specified solution point against validity. If there is no solution, or if the specified solution does not match the calculated one, then the test fails. So test problems cannot be added to a distribution of OpenMDAO without a proper solution specified. The given solution data must include the expected values for all of the design variables, all of the coupling variables, and the objectives.

The *OptProblem* class provides a utility method, check_solution(), which will compare the current state of any instance against it's specified condition. For each combination of architecture and test problem in the test suite, this method is run and the results reported to the user. The results are reported in terms of an absolute difference from the specified solution. So if some variable, x, has a solution of 1.0 and the current solution is at 1.5, then the error reported would be 0.5.

The data in Table 4 shows that all of the architectures solved the Sellar Problem to essentially the exact solution. MDF was the most accurate, but the deviations are not significant for the other architectures. This performance substantiates the assertion above that the Sellar Problem can be considered a baseline test problem that must be solvable by any reasonable architecture.

The performance of the architectures did not change significantly when run on the Sellar Problem with analytic derivatives. The data in Table 5 shows the results of this test. CO got more accurate with the use of analytic derivatives, but otherwise the data was effectively the same.

The results from the optimizations on the Scalable Problem are shown in Table 6. Note that the values reported for the z, x, and y are the maximum error from the known solution from all three disciplines. Each

Table 4: Absolute difference from optmium solution for all architectures solving the Sellar Problem.

	z_1	z_2	x_1	y_1	y_2	Objective
Optimum	1.978	0.000	0.000	3.160	3.756	3.183
IDF	0.006	0.00	0.000	0.039	-0.085	0.039
MDF	0.000	0.000	0.000	-0.001	0.000	0.000
CO	-0.001	0.000	0.000	-0.070	-0.025	0.003
BLISS	0.002	0.000	0.000	0.006	0.003	0.006
BLISS-2000	0.003	0.013	0.000	0.066	-0.072	0.050

Table 5: Absolute difference from optimum solution for all architectures solving the Sellar Problem with analytic derivatives.

	z_1	z_2	x_1	y_1	y_2	Objective
Optimum	1.978	0.000	0.000	3.160	3.756	3.183
IDF	0.000	0.000	0.000	0.001	0.001	0.000
MDF	0.000	0.000	0.000	0.000	0.001	0.000
CO	0.000	0.000	0.011	-0.030	-0.054	0.001
BLISS	0.001	0.000	0.002	0.000	0.000	0.000
BLISS-2000	-0.002	0.000	0.007	0.010	-0.001	0.001

discpline has three gloabal, three local, and three output variables but each group takes the same value at the optimal solution. Variations in the values each architectures found did occur , but the maximum error indicates how the maximum deviation from the optimum for each architecture. For the Scalable Problem, not all of the architectures were able to satisfactorily solve the problem. In particular, BLISS could not converge on the proper solution and was ineffective. CO also had significant trouble converging to a consistent answer. Once again MDF provided the deepest convergence and got closest to the true optimum.

Table 6: Maximum distance from the optimum solution for all architectures solving the Scalable Problem.

	z	x	y	Objective
Optimum	0.000	-0.333	1.000	3.000
IDF	0.000	0.000	-0.001	0.000
MDF	0.000	0.000	0.000	0.000
CO	1.721	-3.635	-0.052	5.547
BLISS	10.820	-2.833	-1.830	127.958
BLISS-2000	-0.021	0.030	0.001	0.030

B. Total Function Evaluations

When OpenMDAO executes, it tracks the number of times that all Component instances are executed. This tally includes any executions made while performing finite difference calculations. This data is presented for all optimizations at the end of the execution of the test procedure.

The data from the optimizations on the Sellar Problem, listed in Table **??**, indicates that IDF uses the fewest function calls, followed by MDF. BLISS-2000 uses more function evaluations for Discipline 1 than MDF, but less for Discipline 2. For the Sellar Problem, Discipline 2 does not have any local design variables so in this case, BLISS-2000 does not need to make any response surface approximations of this discipline and the system optimizer calls the analysis code directly. The function counts for BLISS-2000 are averages of 5 test runs, as indicated by the "*". This was necessary since the response surface approximations are trained by a randomly generated Latin Hypercube DOE, which is a stochastic process. Hence, the function

counts vary slightly from test to test. For the Sellar Problem, CO and BLISS are by far the most expensive and represent an order of magnitude higher cost than the other architectures.

Table 7: Function evaluation counts for all architectures solving the Sellar Problem.

	Discipline 1	Discipline 2
IDF	60	54
MDF	222	216
CO	5647	8252
BLISS	3344	3130
BLISS-2000*	1007	141

Table 8: Function evaluation counts for all architectures solving the Sellar Problem with analytic derivatives. Number of derivative evaluations are in parenthesis.

	Discipline 1	Discipline 2
IDF	6 (6)	6 (6)
MDF	42 (6)	42 (6)
CO	191 (673)	661 (707)
BLISS	1017 (543)	946 (543)
BLISS-2000*	600 (269)	132 (0)

Table 9: Function evaluation counts for all architectures solving the Scalable Problem.

	Discipline 1	Discipline 2	Discipline 3
IDF	80	88	88
MDF	546	541	545
CO	52900	51999	52747
BLISS	7881	8665	6502
BLISS-2000*	2059	2029	1991

For the Sellar Problem with analytic derivatives, the results are somewhat different. In addition to the number of function evaluations the number of derivative evaluations are also presented in parenthesis, in Table **??**. For some analyses, the cost of the derivative evaluation may be negligible and not worth considering for the overall computational cost of an optimization. However in some of the modern adjoint methods for computing analytic derivatives with CFD, the calculation of the adjoint can be as expensive as a function evaluation. In those cases, the total cost would be better represented by the sum of the function evaluations and the derivative evaluations. So the data for both function evaluations and derivative calculations are presented separately.

The data shows that IDF and MDF are still the least expensive options. Both architectures used approximately half as many function calls when analytic derivatives were provided. CO displayed an order of magnitude reduction in the number of function calls, while BLISS benefited by a factor of about three. BLISS-2000 only showed a decrease in the function calls for Discipline 1. Analytic derivatives only assist in the creation of the response surface equations, when there are local design variables that need to be optimized for each DOE case. The end result for the Sellar Problem is that when derivatives are present, BLISS and CO are significantly less expensive and are more competitive with BLISS-2000.

The data for the Scalable Problem is listed in Table **??**. This data also shows that IDF is the least computationally expensive, followed by MDF. However, for this problem BLISS-2000 is in third place, and similar to the data from the Sellar Problem uses about 4 times as many function evaluations. CO and BLISS are both significantly more expensive than any of the other three architectures, but given their accuracy their cost is not really relevant.

C. Convergence Characteristics

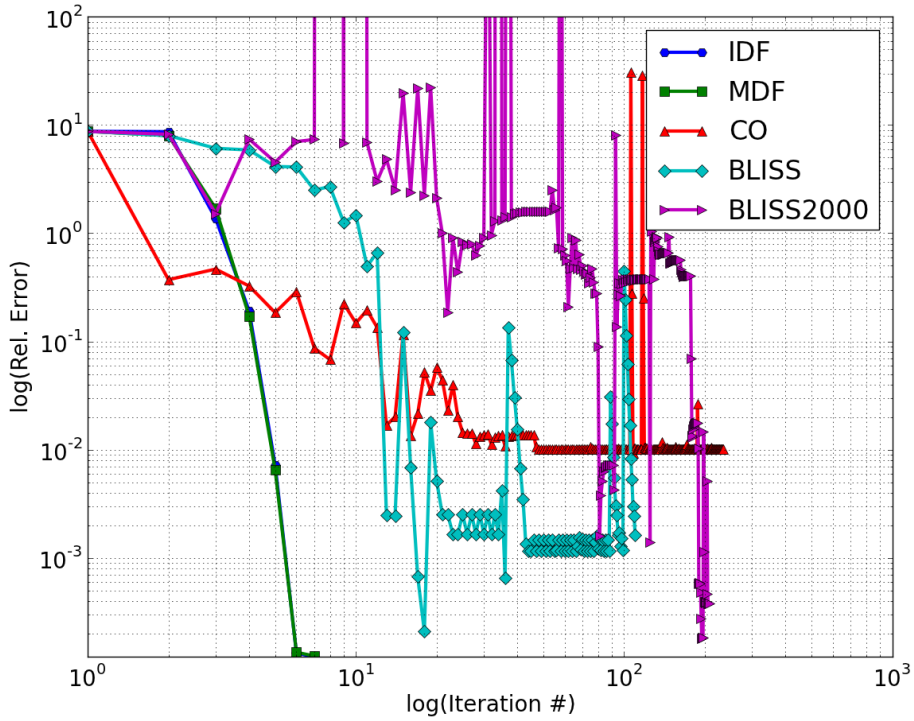

Figure 8: Relative error vs iteration # for all architectures run on the Sellar Problem

The convergence behavior for all five architectures was tracked for the two test problems. The data presented is relative error vs the iteration for the global optimization step of each architecture. The data for the Sellar Problem, in Fig. **??**, indicates a clear convergence trend for the MDF and IDF architectures. The slow convergence issues that Braun et al. identified for CO[?] are clearly visible in the data. BLISS demonstrate a stepped convergence behavior, where improvements to the objective are made in bursts followed by periods of stagnation. BLISS-2000 has a difficult time converging and shows a lot of scatter in the data.

The convergence behavior for the Sellar problem with and without analytic derivatives is considerably different. The trends with analytic derivatives are shown in in Fig. **??**. Both MDF and IDF, with analytic derivatives, show a clear convergence trend with each iteration showing improvement. BLISS-2000 shows a similar convergence trend at first but the algorithm appears to loose sensitivity. CO shows a more well behaved convergence behavior as well. The stepped behavior for BLISS also changes, so that convergence improves initially and the algorithm stagnates at the end.

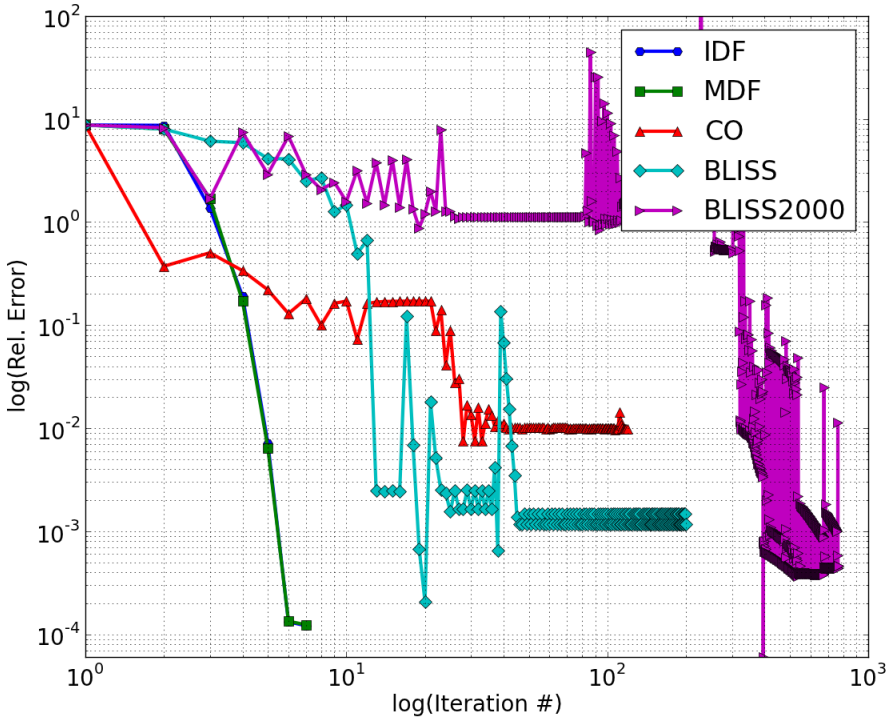

Figure 9: Relative error vs iteration # for all architectures run on the Sellar Problem with analytic derivatives.

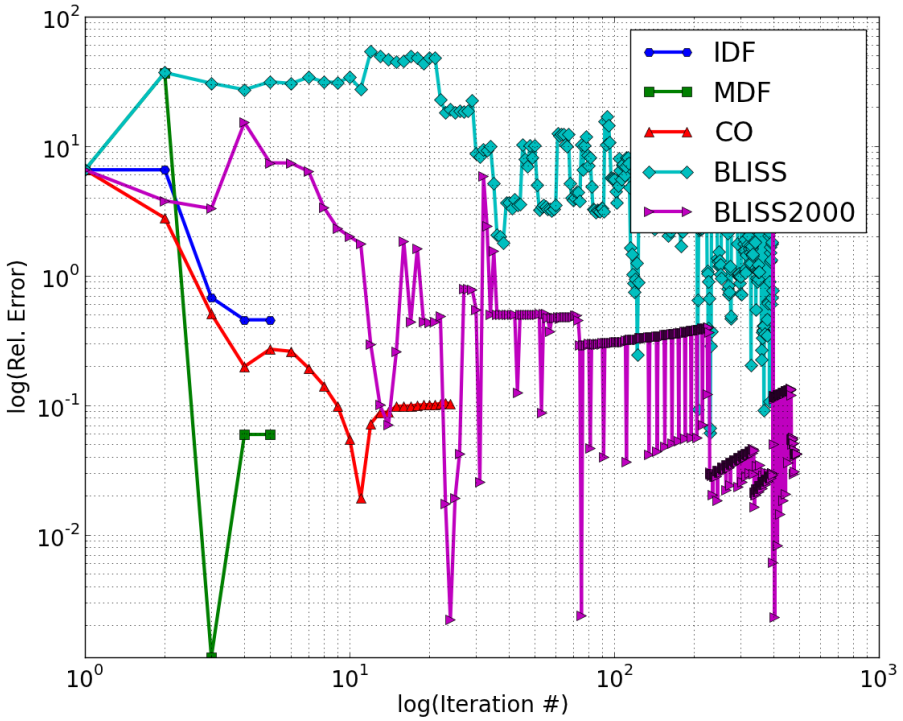

Figure 10: Relative error vs iteration # for all architectures run on the Scalable Problem.

In comparison to the convergence trends from the Sellar Problem, the data for the Scalable Problem in Fig. **??** looks very similar for IDF, MDF, and CO. However BLISS-2000 converges with a clear stepped trend and BLISS fails to converge to an acceptable solution. The erratic convergence for BLISS in the data gives some insight into why. The algorithm appears to lose all sensitivity and starts searching wide sections of the design space near the true optimum. One possible cause of this is a very flat gradient for the Scalable Problem near the optimum. Since BLISS uses linear approximations for all models, a flat gradient would allow large movements. BLISS employs move limits, where design variables are constrained to a neighborhood around their current value, for any one iteration. Our implementation uses fixed move limits, but its possible that adding some kind of decay into the move limits would help convergence.

D. Optimizer Choice

The "no free lunch" theorem proves that no one optimizer can be the most effective for all problems. By extension then, it's impossible to select a single optimizer to use for any given architecture and expect the best performance for all problems. By making use of the flexibility of OpenMDAO architectures we reconfigured the architectures to use the COBYLA optimizer instead of SLSQP, but made no other changes to them. The COBYLA optimizer is a derivative free optimization algorithm which may be useful for problems where analytic derivatives are not available and finite differencing is not feasible. The results in Table **??** show that the COBYLA converged well for the Sellar Problem and accuracy was not significantly affected.

Table 10: Absolute difference from optimum solution for all architectures solving the Sellar Problem using the COBYLA optimizer.

	z_1	z_2	x_1	y_1	y_2	Objective
Optimum	1.978	0.000	0.000	3.160	3.756	3.183
IDF	0.000	0.000	0.000	0.001	0.001	0.000
MDF	0.000	0.000	0.000	0.000	0.000	0.000
CO	0.003	0.000	0.000	-0.026	0.050	-0.001
BLISS	0.000	0.000	0.000	0.001	0.000	0.001
BLISS-2000	0.000	0.000	0.007	0.010	-0.020	0.010

Table 11: Function evaluation counts for all architectures solving the Sellar Problem with the COBYLA optimizer.

	Discipline 1	Discipline 2
IDF	43	42
MDF	179	179
CO	8313	5250
BLISS	2062	1843
BLISS-2000	502	77

Table **??** shows that that the number of function evaluations and hence the computational cost of the optimizations was reduced for IDF, MDF, and BLISS when using COBYLA compared to the Sellar Problem with finite differencing. CO and BLISS-2000 showed a negligible change in computational cost.

In Fig. **??** the convergence data is shown for the COBYLA tests. Most of the architectures display a more noisy convergence path, with the exception of BLISS-2000 which gets less noisy with COBYLA. Despite the less smooth path COBYLA may be a better choice for problems where analytic derivatives are not available if you're using IDF, MDF, or CO.

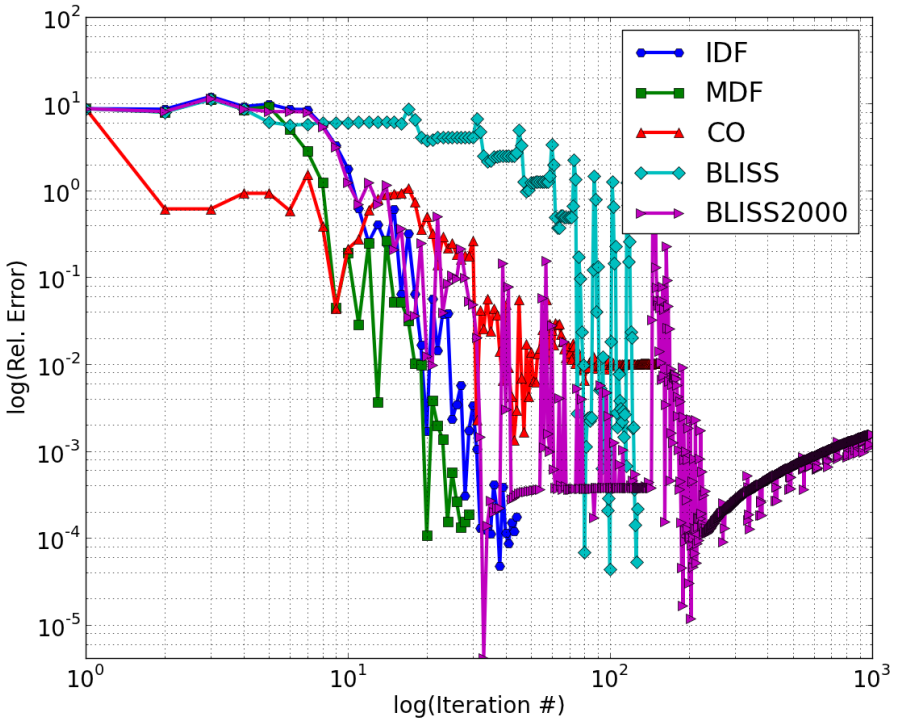

Figure 11: Relative error vs iteration # for all architectures run on the Sellar Problem with the COBYLA optimizer. Problem

VI. Community Contribution

There are multiple other existing MDAO architectures which have not yet been implemented in OpenMDAO, such as Analytical Target Cascading,[?,?] Enhanced Collaborative Optimization,[?] or Asymmetric SubOptimization.[?] There are also additional test problems which need to be implemented to provide a stronger basis for comparison of architecture performance. For instance, this study did not include any test problems which simulated the presence of noise in results or displayed problems with analysis failures. Both kinds of problems have the potential to strongly impact the performance of MDAO architectures. The code from the OpenMDAO code base that defines the Sellar Problem is listed in Appendix **??**. This code shows that addition of new problems to the framework is strait forward and does not require an excessive knowledge of the OpenMDAO framework.

OpenMDAO is developed and distributed under the Apache V2.0 open source license,[?] so it is possible for the MDAO community to contribute additional architectures and test problems as needed. OpenMDAO hosts its source code on the GitHub social coding website, which combines social networking with a public source code repository to encourage community participation on software development.[?] The OpenMDAO code repository can be found at http://github.com/OpenMDAO. Interested parties are able to visit the source code repository and browse the code right from the website. They can also fork, or copy, the repository to their own workspace to make changes and add additional architectures or test problems. GitHub provides all the necessary tools to merge, or re-combine, the code from each person's fork back into the main repository through a web-based interface,[?] which makes it easy for the community to submit code contributions.

GitHub also provides an automated means for tracking contributions to the codebase, so that appropriate credit can be given for contributions from researchers. Figure **??** shows a sample network graph from the OpenMDAO repository on GitHub. The graph gives an indication of who has code out there that has not been incorporated into the main development branch.[?] The network graph, is interactive so that clicking on a node will take you to the particular version of the code being worked on by a specific researcher at any given time. So it becomes simple to work not only off the main development version of OpenMDAO, but

also from any researchers personal fork and keep track of where the collective code base is headed.

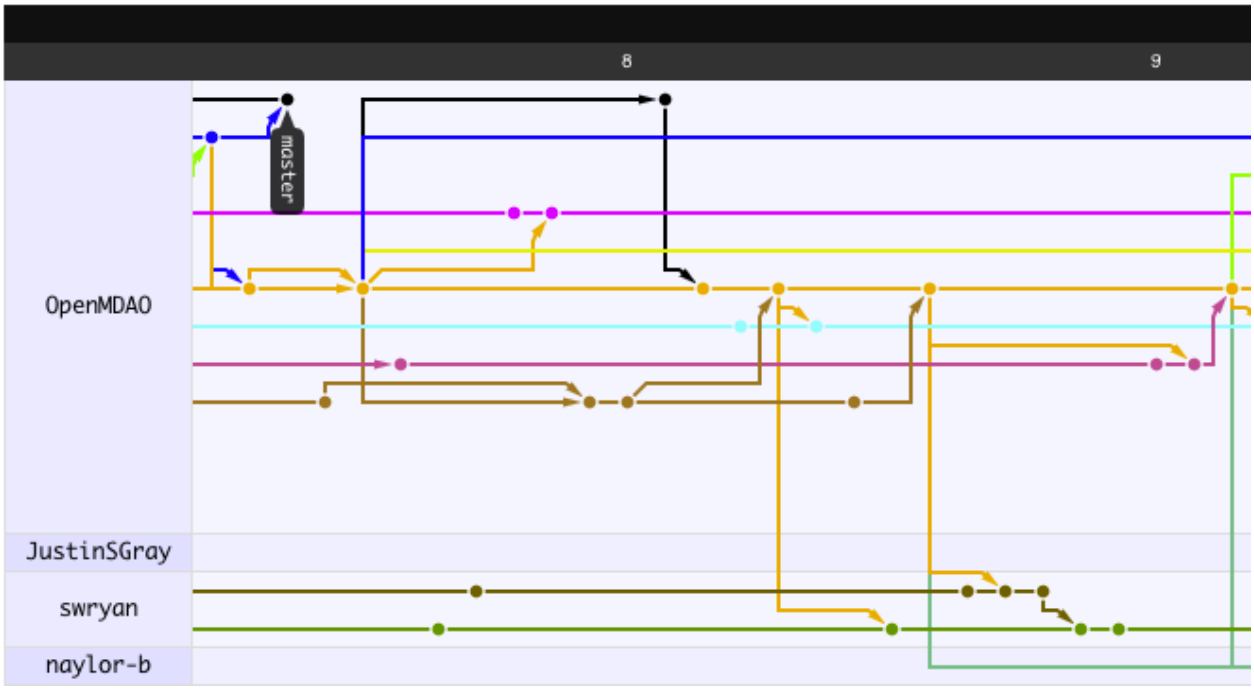

Figure 12: Sample commit network from the OpenMDAO repository on GitHub. Problem

From the perspective of a researcher working on a new test problem or new architecture, a large test suite built by community contributions would provide a solid benchmark to test against. The easiest way to test against the existing test suite would be to add the new material into a fork of the OpenMDAO software. Then it would be simple to contribute the new material to the community by issuing a pull request to have the fork integrated back into main code base. By using open source software development processes OpenMDAO provides the means for establishing this test suite and also ensures its continued relevance through community contributions of new test problems and MDAO architectures.

VII. Conclusions

Initially we laid out four required elements for establishing a successful MDAO testing framework:

1. Selecting a common software framework on top of which the MDAO community could develop and share a testing platform
2. Constructing a large set of MDAO architectures which are implemented as efficiently as possible
3. Compiling a suite of test problems which have properties that simulate a wide range of engineering analyses and problem scales
4. Ensuring that new test problems and architectures, as well as improvments, can be added into the test suite by the community

This work has demonstrated that the OpenMDAO framework has the necessary features to address all four of the above needs. We have implemented five different MDAO architectures: IDF, MDF, CO, BLISS, and BLISS-2000. These implementations prove that there is sufficient flexibility to handle a very wide range of architecture designs, including nested optimizations and the algorithmic creation of metamodels as part of a workflow. New components (*SensitivityDriver* and *NeighboorhoodDOEdriver*) were added to the standard library to support this effort, and those components are now available to developers who wish to use them for new architecture concepts. Additionally, we demonstrated that OpenMDAO can easily support the use of multiple optimization algorithms for any given architecture. This feature is important so that architecture implementations can be usable for real problems outside the test suite.

Building off this foundation, a testing procedure was defined and built into the OpenMDAO framework to automatically run all of the architectures for all of the test problems. The results of each optimization were compared using a number of performance metrics. The data indicates that IDF and MDF are the most effective of the tested architectures and by far the least computationally expensive. BLISS-2000 is also fairly inexpensive, when analytic derivatives are not available.

This work used the number of function evaluations and derivative evaluations as an indication of computational cost for each architecture. This assumes a completely serial execution of all discipline analyses. CO, BLISS, and BLISS-2000 all offer varying degrees of potential parallelization which could be taken advantage of given enough computational resources. Quantifying the effect of parallelism on the overall algorithms cost is not trivial and is highly dependent on implementation details of a given architecture. So even two implementations of MDF, using different kinds of solver configurations, might have different parallelization potentials. Regardless, given that BLISS-2000 was relatively close to MDF in terms of function calls without analytic derivatives, further investigation into the effective cost of BLISS-2000 when considering its parallel potential is warranted.

Since only two test problems were considered in this work, the resulting benchmarks can not be considered definitive. A larger set of test problems in OpenMDAO would better indicate to potential users which architectures would be effective for their needs. The data collected for the Sellar Problem with and without analytic derivatives demonstrates that the availability of analytic derivatives can dramatically alter the performance potential of different architectures.

Using the established testing procedure with a larger test suite would provide a clear and consistent basis for comparison among architectures. As new architectures are developed, running them on the test set would provide apples-to-apples performance comparisons against established architectures.

Community contributions are needed to expand the test suite beyond the five architectures and two test problems already implemented. Using the OpenMDAO code-hosting site on GitHub, researchers could make development forks of the code and merge those forks back to the project to grow the set of architectures and test problems. GitHub provides a number of tools to facilitate community collaboration on the OpenMDAO codebase.

Establishing a consistent test procedure is necessary for the continued progress of the MDAO field. Without such a procedure, no obvious measurement exists to judge the effectiveness of new MDAO architectures and potential users will continue to have a difficult time determining which architecture is appropriate to solve their problem.

References

[1]Martins, J. R. R. A. and Lambe, A. B., "Multidisciplinary Design Optimization: Survey of Architectures," 2011, pp. 1–46.

[2]Alexandrov, N., "Initial results of an MDO method evaluation study," *AIAA Paper*, 1998, pp. 1–13.

[3]Alexandrov, N. M. and Lewis, R. M., "Comparative Properties of Collaborative Optimization and Other Approaches to MDO," *Proceedings of the First ASMO UK ISSMO Conference on Engineering Design Optimization*, , No. 99, 1999.

[4]Kodiyalam, S. and Yuan, C., "Evaluation of Methods for Multidisciplinary Design Optimization (MDO), Part II," Tech. Rep. November, NASA/CR-2000-210313, Hampton, Virginia, Nov. 2000.

[5]Marler, R. and Arora, J., "Survey of multi-objective optimization methods for engineering," *Structural and Multidisciplinary Optimization*, Vol. 26, No. 6, April 2004, pp. 369–395.

[6]Hulme, K. F. and Bloembaum, C. L., "A MULTIDISCIPLINARY DESIGN TEST SIMULATOR," *Sixth AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Bellevue, WA, 1996, pp. 438–447.

[7]Martins, J., Marriage, C., and Tedford, N., "pyMDO: an object-oriented framework for multidisciplinary design optimization," *ACM Transactions on Mathematical Software (TOMS)*, Vol. 36, No. 4, 2009, pp. 1–25.

[8]Tedford, N. P. and Martins, J. R. R. a., "Benchmarking multidisciplinary design optimization algorithms," *Optimization and Engineering*, Vol. 11, No. 1, March 2009, pp. 159–183.

[9]Marriage, C. J. and Martins, J. R. R. A., "Reconfigurable Semi-Analytic Sensitivity Methods and MDO Architectures within the πMDO Framework," *12th AIAAISSMO Multidisciplinary Analysis and Optimization Conference*, No. September, 2008.

[10]Padula, S. L. S., Alexandrov, N., and Green, L. L., "MDO Test Suite at NASA Langley Research Center," *Proceedings of the 6th AIAANASAISSMO Symposium on Multidisciplinary Analysis and Optimization*, No. 96, NASA Langley Research Center, Bellevue, WA, 1996, pp. 1–13.

[11]Perez, R. E., Liu, H. H. T., and Behdinan, K., "Evaluation of Multidisciplinary Optimization Approaches for Aircraft Conceptual Design," *Proceedings of the 10th AIAAISSMO Muldisiciplinary Analysis and Optimization Conference*, No. September, 2004, pp. 1–11.

[12]Brown, N. F. and Olds, J. R., "Evaluation of Multidisciplinary Optimization Techniques Applied to a Reusable Launch Vehicle," *Journal of Spacecraft and Rockets*, Vol. 43, No. 6, 2005, pp. 1289–1300.

[13]Felder, J. L., Kim, H. D., and Brown, G. V., "Turboelectric Distributed Propulsion Engine Cycle Analysis for Hybrid-Wing-Body Aircraft," *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition AIAA Paper 20091132*, , No. January, 2009, pp. 1–25.

[14]Kim, H., Brown, G., and Felder, J., "Distributed turboelectric propulsion for hybrid wing body aircraft," *9th International Powered Lift Conference, London, United Kingdom*, 2008.

[15]Moore, K., Naylor, B., and Gray, J., "The development of an open source framework for multidisciplinary analysis and optimization," *12th AIAA/ISSMO multidisciplinary analysis and optimization conference*, AIAA, Victoria, Canada, Aug. 2008.

[16]Langtangen, H. P., "Python Scripting for Computational Science," *New York*, Vol. 3, 2008, pp. 727.

[17]Parsons, D., Rashid, A., and Speck, A., "A framework for object oriented frameworks design," *IEEE Society*, Technology of Object-Oriented Lanauges and System, 29th Int'l Conference and Exhibition, Nancy, France, 1999, pp. 141–151.

[18]Parnas, D. L., "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, Vol. 15, No. 12, 1972, pp. 1053–1058.

[19]Peterson, P., Martins, J. R. R. A., and Alonso, J. J., "Fortran to Python interface generator with an application to aerospace engineering," *Proceedings of the 9th International Python Conference*, 2001.

[20]Moore, K. T., "Wrapping an External Module Using F2PY OpenMDAO Documentation, V 0.1.7," 2011.

[21]Perez, R. E., Jansen, P. W., and Martins, J. R. R. A., "pyOpt : A Python-Based Object-Oriented Framework for Nonlinear Constrained Optimization," *Optimization and Engineering*, Vol. V, 1993, pp. 1–22.

[22]Kraft, "A software package for sequential quadratic programming." Tech. rep., DLR German Aerospace Center Institute for Flight Mechanics, Koln, Germany, 1998.

[23]Powell, M., "A direct search optimization method that models the objective and constraint functions by linear interpolation," *Advances in Optimization and Numerical Analysis*, edited by Dordrecht, Kluwer Academic, 1994, pp. 51–67.

[24]Lambe, A. B. and Martins, J. R. R. A., "Extensions to the Design Structure Matrix for the Description of Multidisciplinary Design, Analysis, and Optimization Processes," *Structural and Multidisciplinary Optimization*, 2012.

[25]Sobieski, J., Altus, T. D., Phillips, M., and Sandusky, R., "Bilevel Integrated System Synthesis for Concurrent and Distributed Processing," *AIAA Journal*, Vol. 41, No. 10, 2003, pp. 1996–2003.

[26]Altus, T. D., "A Response Surface Methodology for Bi-Level Integrated System Synthesis (BLISS)," Tech. Rep. May, NASA Langley Research Center, Hampton Virginua, May 2002.

[27]Sellar, R., Batill, S., and Renaud, J., "Response surface based, concurrent subspace optimization for multidisciplinary system design," *34th AIAA Aerospace Sciences Meeting and Exhibit*, AIAA, Citeseer, Reno, NV, Jan. 1996.

[28]Roth, B. and Kroo, I., "Enhanced collaborative optimization: application to an analytic test problem and aircraft design," *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, AIAA, Victoria, British Columbia, 2008.

[29]Martins, J. R. R. A. and Marige, C., "An Object-Oriented Framework for Multidisciplinary Design Optimization," *3rd AIAA Multidisciplinary Design Optimization Specialist Conference*, AIAA, Waikiki, Hawaii, 2007.

[30]Braun, R. D., Gage, P. J., Kroo, I. M., and Sobiesiki, I., "Implementation and Performance Issues in Collaborative Optimization," 1996.

[31]Allison, J., Kokkolaras, M., Zawislak, M., and Papalambros, P., "On the use of analytical target cascading and collaborative optimization for complex system design," *Proceedings of the 6th world congress on structural and multidisciplinary optimization, Rio de Janeiro, Brazil*, Citeseer, 2005, pp. 1–10.

[32]Michalek, J. J. and Papalambros, P. Y., "An Efficient Weighting Update Method to Achieve Acceptable Consistency Deviation in Analytical Target Cascading," *Journal of Mechanical Design*, Vol. 127, No. 2, 2005, pp. 206.

[33]Chittick, I. R. and Martins, J. R. R. A., "An Asymmetric Suboptimization Approach to Aerostructural Optimization," *Optimization and Engineering*, Vol. 10, No. 1, 2009, pp. 133–152.

[34]"Apache License, Version 2.0," *The Apache Software Foundation http://www.apache.org/licenses/LICENSE-2.0.html*.

[35]Storey, M.-A. and Treude, C., "The Impact of Social Media on Software Engineering Practices and Tools," *Research Studies*, 2010, pp. 359–363.

[36]Eaves, D., "How GitHub Saved OpenSource," *eaves.ca http://eaves.ca/2011/06/14/how-github-saved-opensource/*.

[37]Preston-Werner, T., "Say Hello to the Network Graph Visualizer," *GitHub Blog*, April 2008, pp. https://github.com/blog/39–say–hello–to–the–networ.

A. Sellar Problem Definition

Below is the code from the OpenMDAO code base that defines the Sellar Test Problem

```
from openmdao.main.api import Component, ComponentWithDerivatives
from openmdao.main.problem_formulation import OptProblem
from openmdao.lib.datatypes.api import Float

class Discipline1(Component):
    """Component containing Discipline 1"""

    z1 = Float(0.0, iotype='in', desc='Global Design Variable')
    z2 = Float(0.0, iotype='in', desc='Global Design Variable')
    x1 = Float(0.0, iotype='in', desc='Local Design Variable')
    y2 = Float(0.0, iotype='in', desc='Disciplinary Coupling')

    y1 = Float(iotype='out', desc='Output of this Discipline')


    def execute(self):
        """Evaluates the equation
        y1 = z1**2 + z2 + x1 - 0.2*y2"""

        z1 = self.z1
        z2 = self.z2
        x1 = self.x1
        y2 = self.y2

        self.y1 = z1**2 + z2 + x1 - 0.2*y2


class Discipline2(Component):
    """Component containing Discipline 2"""

    z1 = Float(0.0, iotype='in', desc='Global Design Variable')
    z2 = Float(0.0, iotype='in', desc='Global Design Variable')
    y1 = Float(0.0, iotype='in', desc='Disciplinary Coupling')

    y2 = Float(iotype='out', desc='Output of this Discipline')


    def execute(self):
        """Evaluates the equation
        y2 = y1**(.5) + z1 + z2"""

        z1 = self.z1
        z2 = self.z2

        #abs deals with some convergence issues if solver tries negative values
        y1 = abs(self.y1)

        self.y2 = y1**(.5) + z1 + z2


#Note: Inherits from OptProblem
class SellarProblem(OptProblem):
    """ Sellar test problem definition."""

    def __init__(self):
        """ Creates a new Assembly with this problem
```

```
Optimal Design at (1.9776, 0, 0)

Optimal Objective = 3.18339"""

super(SellarProblem, self).__init__()

#add the discipline components to the assembly
self.add('dis1', Discipline1())
self.add('dis2', Discipline2())

#START OF MDAO Problem Definition
#Global Des Vars
self.add_parameter(("dis1.z1","dis2.z1"),name="z1",low=-10,high=10,start=5.0)
self.add_parameter(("dis1.z2","dis2.z2"),name="z2",low=0,high=10,start=2.0)

#Local Des Vars
self.add_parameter("dis1.x1",low=0,high=10,start=1.0)

#Coupling Vars
self.add_coupling_var(("dis2.y1","dis1.y1"),name="y1",start=0.0)
self.add_coupling_var(("dis1.y2","dis2.y2"),name="y2",start=0.0)

#Objectives and Constraints
self.add_objective('(dis1.x1)**2 + dis1.z2 + dis1.y1 + math.exp(-dis2.y2)',name="obj1")
self.add_constraint('3.16 < dis1.y1')
self.add_constraint('dis2.y2 < 24.0')

#solution to the opt problem
self.solution = {
    "z1":1.9776,
    "z2":0.0,
    "dis1.x1":0.0,
    "y1":3.16,
    "y2": 3.756,
    'obj1':3.1834
}

#END OF MDAO Problem Definition
```