A Standard Platform for Benchmarking 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 that 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.

*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

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 from 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.

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[1] 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 a given design optimization problem. Traditionally, this selection has been made somewhat informally, with a tendency to rely on architectures that are already familiar to the researcher.[2] To address this, a number of efforts have provided comparative results among architectures that could be used to make a more informed decision.[3,4] Hulme and Bloembaum[5] 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. Their research focused on exercising the three architectures on a large number of randomly generated test problems. The results indicated that MDF tended to be the most accurate, especially as the complexity of the system being optimized increases. They note that this accuracy can come at a cost, showing that MDF made as many as 10 times the number of function evaluations of IDF or AAO for the kinds of problems generated by CASCADE. Kodiyalam and Sobieski[6,7,8] laid out a series of requirements for an effective framework that could be used to compare architectures and present some initial comparisons among AAO, IDF, MDF, and BLISS with two engineering test problems. To generate their results, they specify custom problem formulations for each combination of test case and architecture. This makes their work difficult to extend to alternative architectures or test problems. Most recently Martins et al.[9] developed a prototype MDAO framework, pyMDO, that was designed specifically to enable the automatic application of MDAO architectures to engineering problems that address the adaptability challenges from other work. Tedford and Martins[10] used pyMDO to compare architecture performance across 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 they stipulated that this result was specific to the problems they tested and that it would likely change when problems with analytic derivatives were considered. In related work, Marriage and Martins[11] 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. The existing work makes it clear that architecture performance is problem dependent. While many new architectures have been developed and modifications to existing architectures have been proposed, no standardized benchmarking has emerged to demonstrate performance of each one.

Hulme and Bloembaum used randomly generated test problems. Tedford and Martins' work relied heavily on analytic functions. Kodiyalam and Sobieski performed testing with a mix of simplified analytical problems and realistic problems involving actual analysis tools. Padula et al.[12] proposed a number of different possible test problems in the MDO test suite, which included some of the problems previously mentioned. 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 NASA MDO Test Suite established the foundation for a publicly available test suite of problems that the community could use.

So from previous research, we conclude that it is necessary to consider a number of test problems that exhibit characteristics similar to those expected for the real analyses but which do not rely on proprietary external anlysis tools. Furthermore, it is possible that different implementations of the same architecture need to be compared as well.[13,14] The work with CASCADE and pyMDO established the foundation for a test procedure by demonstrating that architectures could be automatically applied to a given set of problems. A number of more recent efforts have attempted to develop MDAO frameworks with useful properties for solving highly integrated engineering problems. Kolonay and Sobolewski[15,16,17] have developed a framework called SORCER that provides Service Oriented Architecture as a platform for distributed, heterogeneous computing. This work provides a foundation for supplying analyses in a distributed manner and suggests a possible implementation for a common integration platform. 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 need to address the following challenges in order to successfully develop an effective testing platform:

1. Select a common software framework as a foundation for the MDAO community to develop a shared testing platform
2. Construct a large set of MDAO architectures that can be automatically applied to problems
3. Compile a suite of test problems with properties that simulate a wide range of engineering analyses and problem scales
4. Ensure that the community can add new test problems and architectures, as well as improvements, to the test suite

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.[18,19] 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.[20]

OpenMDAO is programmed in the Python language[21] 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.[22,23] 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. More complete documentation can be found in the OpenMDAO User Guide.[24]

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.

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++.[25] Users 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

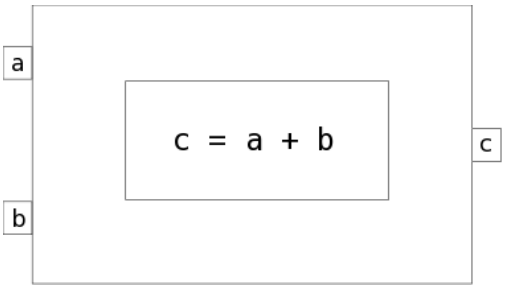

Figure 1: Conceptual view of a simple component; a, b, and c are all variables.

some performance improvements. In the *OpenMDAO User Guide*,[26] 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.[26]

What is refered to here as a "component", is also commonly refered to as a "discipline" when discussion MDAO architectures. For the purposes of this work, the two should be considered interchangable. It is possible for instance, to have two different models (possibly of different fidelities) which both belong to the aerodynamics field. Here, each would be considered a different "component" or "discipline".

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 supply 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.[27] with pyOpt focused on developing a single uniform API for describing MDAO problems to optimizers. OpenMDAO extends that concept beyond optimization to include all types of drivers (e.g., solvers, sensitivity calculators, Design of Experiments). To accommodate the wider range of driver types, the OpenMDAO Driver API is broken down into several sub-APIs:

- HasParameters: accept one or more parameters whose value can be set
- HasObjective: accept one (and only one) objective function
- HasObjectives: accept one or more objective functions
- HasEqConstraints: accept one or more equality constraints
- HasInEqConstraints: accept one or more inequality constraints

- HasStopConditions: accept a test that when evaluates to True, will halt iteration

For example, the DOEdriver from the standard library supports only the HasParameters sub-API. Both BroydenSolver and FixedPointIterator support HasParameters and HasEqConstraints. All optimizers will support HasParameters and either HasObjective or HasObjectives but can then also support one or more of the constraint sub-APIs. Somewhat uniquely, IterateUntil supports only HasStopConditions.

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 provide the flexibility needed to build complex optimization processes defined by MDAO architectures. Components can appear multiple times in a single workflow or in multiple parts of a nested workflow. This process 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 that is used as a component in an aircraft simulation.

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	Vector of constraint functions
N	Number of disciplines
$()_0$	Functions or variables that are shared by more than one discipline (global)
$()_i$,$()_j$	Functions or variables that apply only to discipline i or j (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

B. Problem Formulation

Like traditional optimization problems, MDAO problems can be represented by a fundamental problem formulation that describes the goals of the optimization independent of any details about how the problem will be solved. This fundamental formulation is comprised of a set of six elements:

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

Using more a more formal mathematical syntax from Table 1 the problem formulation is given as:

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

Each coupling variable pair includes both an input to a given component and the output from another component that must be consistent (i.e., $g^c = 0$) at a converged solution. This notation is derived from the notation used by Martins[1] in his survey of architectures but modified slightly to better conform to the conventions used in OpenMDAO.

It could be argued that there is no real distinction between local and global design variables, because one 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, such as the Combustion of the Propane Problem from the MDO Test Suite,[12] 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., vary the state variables to 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. However, we do acknowledge that future studies may require them to be considered more directly.

1. OptProblem Class

OpenMDAO provides a subclass of Assembly, called *OptProblem*, that 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 the optimal solution to be specified. This includes the optimal values for all design variables, coupling variables, and objectives. The inclusion of this information allows accurate benchmarking 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 specific solution formulation for a given fundamental problem formulation. In contrast to the fundamental problem formulation, a solution formulation contains explicit details about how a given problem will be solved, including any new sub-problems representing pieces of the fundamental problem formulation. The solution formulation, in addition to the existing design variables, objectives, and constraints, may also have new variables, objectives, and constraints associated with any new sub-problems. 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 new target variables in the global optimization.

For some simple MDAO architectures, constructing the solution formulation by building up the appropriate drivers and associated workflows would be fairly straightforward to do manually. 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. In OpenMDAO the solution formulation is implemented automatically by building up a set of drivers and associated workflows by the *Architecture* class. This class configures the specifics of the solution formulation based on an inspection of the fundamental problem formulation from a specific *Optproblem* instance. This design allows for architectures to be implemented automatically for any problem specified as an OptProblem instance. Hence, setting up an automatic architecture for the OpenMDAO test suite also gives users the ability to automatically apply it to any other

real-world problems built as OptProblem instances as well. All of the architectures investigated here are coded as sub-classes of *Architecture* in the architectures section of the OpenMDAO standard library. The formal relationship between Architectures and the rest of the OpenMDAO framework are described in Fig. 2 with UML syntax.

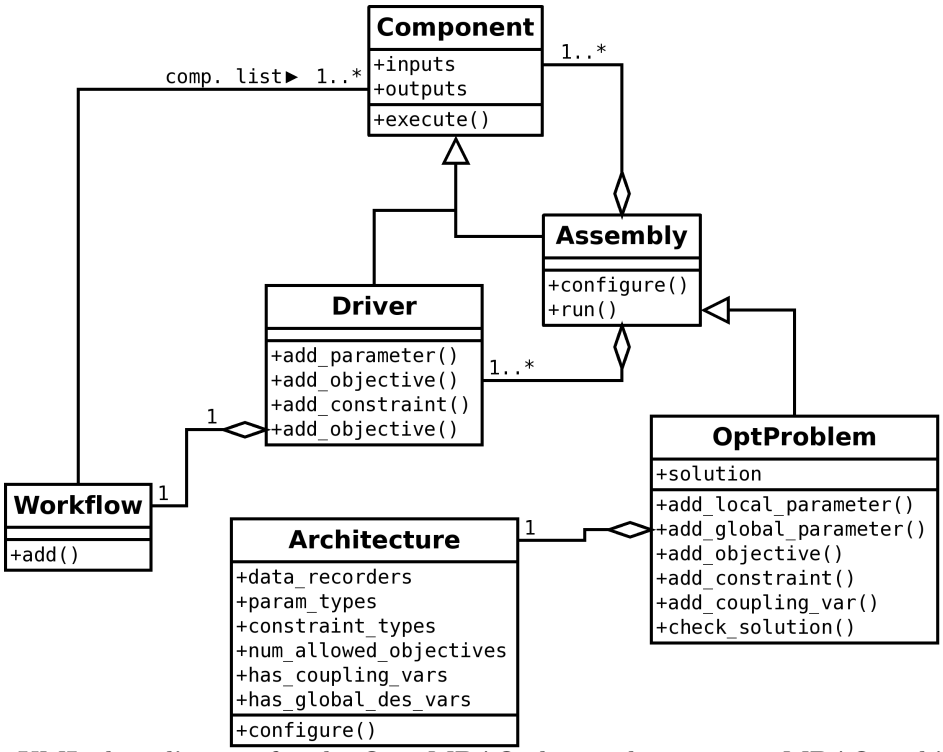

Figure 2: UML class diagram for the OpenMDAO classes that support MDAO architectures.

Fig. 2 shows that an *Architecture* builds a specific arrangement of *Driver, Component, Assembly* and *Workflow* instances to represent the prescribed solution for a given problem. This design has a notable characteristic that after the process is configured, the users are free to make modifications to suit their needs. For instance, users could change the optimizer being used for one of the sub-problems, modify which solver was used in the Multidisciplinary Analysis (MDA), or change to a different type of surrogate modeling tool. 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. It is worth noting that this design, while similar in concept to that of pyMDO, is fundamentally different in implementation. With pyMDO, OptProblem instances are reformulated for a specific framework by a specific Architecture Reformulation service in the framework.[9] This means that making modifications to the solution formulation at runtime is not possible. Instead, you would have to define a new Architecture class, potentially one that inherited from the original one, and make the changes there. Additionally, within OpenMDAO the user could choose not to use any Architecture at all and configure everything manually. The design used in OpenMDAO is significantly more flexible than

the one in pyMDO.

Although this kind of flexibility is valuable for real-world problems and highly useful for developing new architectures, 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.

III. Architectures

The architectures implemented for this work demonstrate the wide range of workflows and processes that can be implemented within OpenMDAO. They serve as templates for future researchers to work from when building new and improved MDAO architectures. They also clearly show how OpenMDAO enables the automatic application of an architecture to a specified problem. 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 all the architectures in this research used the the SLSQP[28] optimizer. Use of a single optimizer allowed for the direct comparison of architecture performance. However, for completeness, some benchmarking was also performed using the COBYLA[29] optimizer, which demonstrates differing performance on the same problem and architecture between optimization algorithms. It should also be noted that in the implementation of each architecture there are a number of settings that can be adjusted to fine-tune its performance. Some of these settings are inherent to the optimizers used in each implementation, such as convergence tolerance. For all cases the convergence tolerance was held fixed at 1×10^{-6}. Other settings, such as the sample size for the regression models in BLISS-2000, are fundamental features of the architecture itself. For each architecture, specific values for these settings were selected and then held fixed. Details for architecture-specific settings are given in the respective sections below.

Below are descriptions of the process and formulation for each of the five 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 proposed by Lambe and Martins[30] 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 the order the blocks get executed. The thick gray 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. A stack of any given block type that has an i in the title (e.g., Analysis i), indicates that n such blocks exist

and may 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) that applies to 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, one follows the path from 1 to 2 to 3, and then returns to step 1 and continues looping until an optimum is reached. The numeric labels in the data-blocks indicate the step during which the data is either input to or output from the block.

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[31] is shown in Fig. 3.

The problem formulation is as follows:

$$\begin{aligned} min \quad & f_0(x, y(x, y^t)) \\ w.r.t. \quad & x \\ s.t. \quad & g_0(x, y(x, y^t)) \geq 0 \\ & g_i(x, y(x, y^t_{j\neq i})) \geq 0 \ for \ i = 1, ..., N \\ & g^c_i(x, y(x, y^t)) = y^t_i - y_i(x, y^t_{j\neq i}) = 0 \ for \ i = 1, ..., N \end{aligned} \tag{2}$$

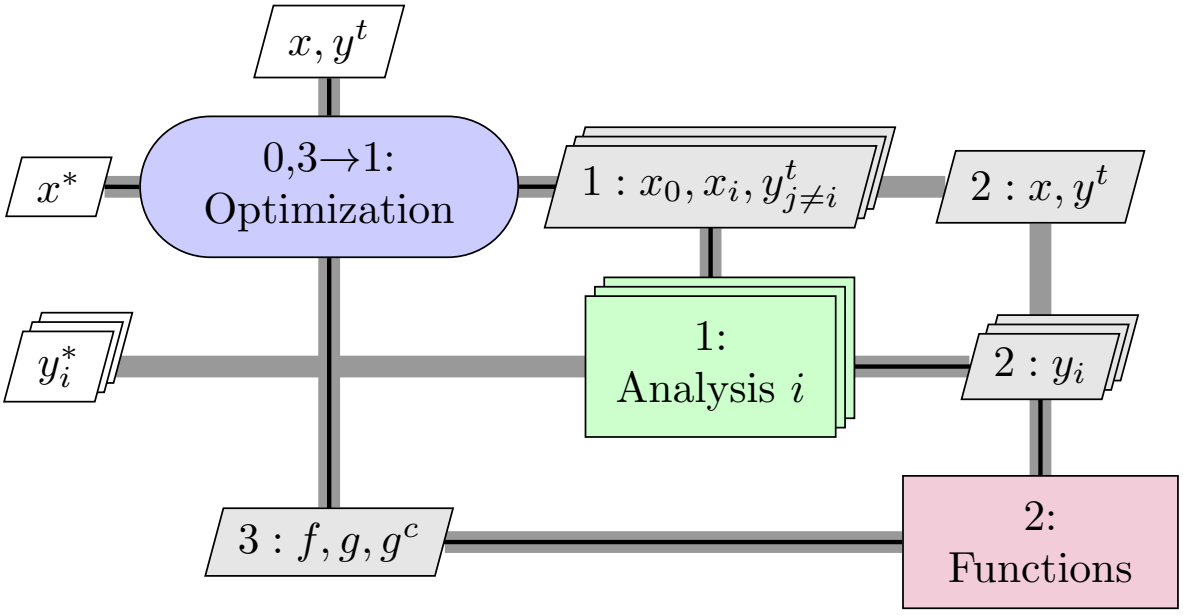

Figure 3: XDSM diagram for IDF adapted from Martins MDO Architecture Survey.[1]

B. MultiDisciplinary Feasible (MDF)

Like IDF, MDF[31,32] 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 that finds values of the coupling varialbes that satisfy the coupling constraints for every iteration of the optimizer. The result is that in MDF the optimizer is only allowed to vary the design variables and never sees anything except a converged system. The XDSM formulation for MDF is shown in Fig. 4.

In Fig. 4, the loop from 3 → 2 represents the MultiDisciplinary Analysis (MDA). In this implementation convergence is accomplished using the *BroydenSolver* driver from 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.

$$\begin{aligned} min \quad & f_0(x, y(x, y)) \\ w.r.t. \quad & x \\ s.t. \quad & g_0(x, y(x, y)) \geq 0 \\ & g_i(x, y(x, y_{j \neq i})) \geq 0 \ for \ i = 1, ..., N \end{aligned} \tag{3}$$

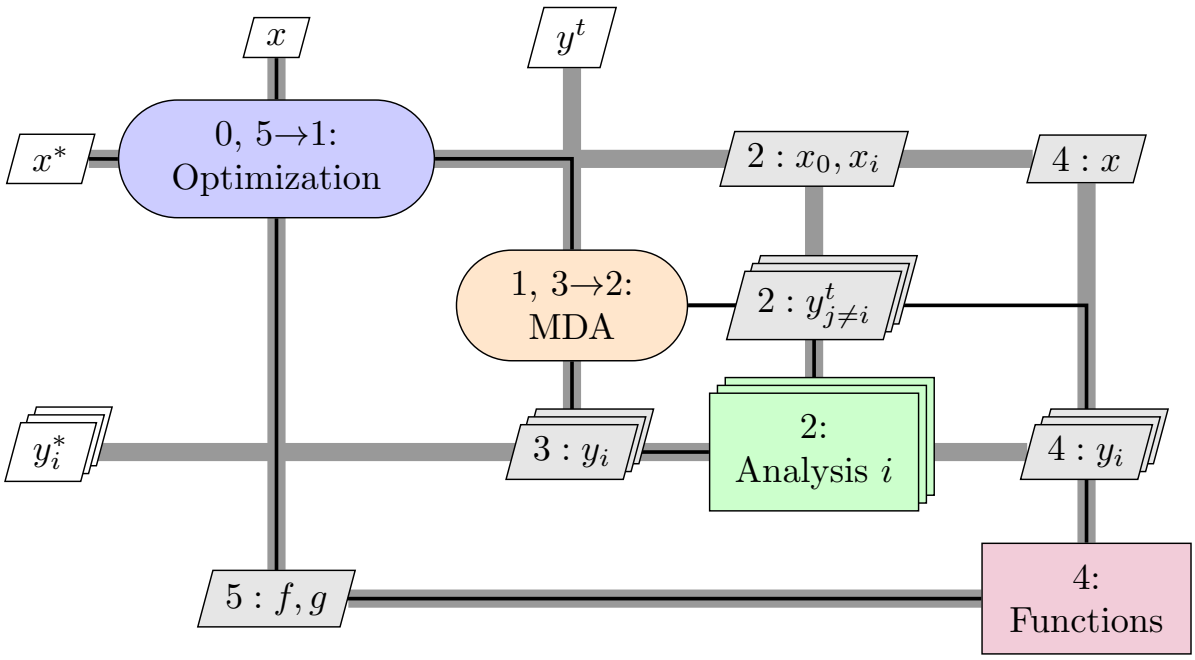

Figure 4: XDSM diagram for MDF adapted from Martins MDO Architecture Survey.[1]

C. Collaborative Optimization (CO)

CO[33] 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 that 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 $1.3 \rightarrow 1.1$ denotes 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:

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

For the local optimizations the problem formulation is as follows:

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

D. Bi-Level Integrated Systems Synthesis (BLISS)

BLISS[34] 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 unique to that specific discipline.

The general process is to generate a linearized approximation of the system, optimize with respect to that approximation, and then generate a new linearized approximation at the optimum point. Target variables

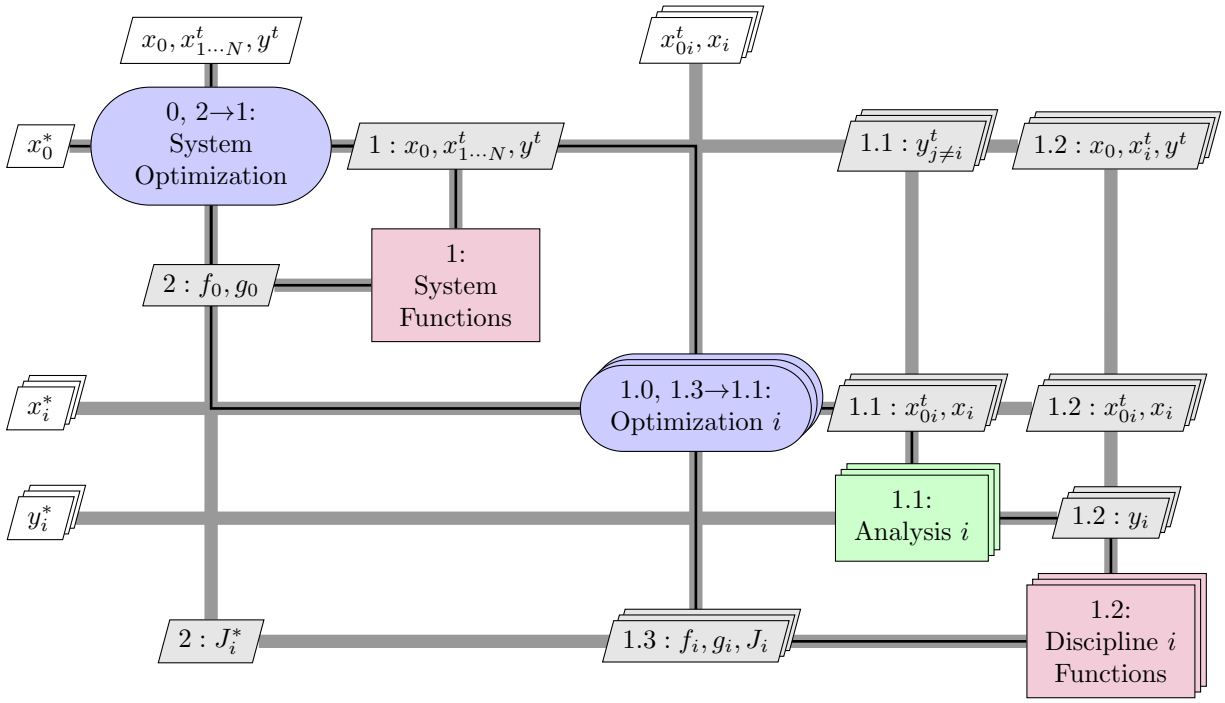

Figure 5: XDSM diagram for CO adapted from Martins MDO Architecture Survey.[1]

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, $\Delta x_{i\ limit}$, to constrain the distance the optimization can move during one iteration. For this work, the move limit was set to 5% of the initial value for any given design variable. 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 major iteration for BLISS is controlled by a Fixed Point Iteration that checks to see if the design variables have converged to within a specified tolerance between the current and previous iterations. For this work the lowest value of tolerance that could be used without always hitting the maximum iteration limit was .005. Since BLISS operates on a linear approximation for the actual system, it can be difficult to get this architecture to converge very tightly. This tolerance should not be confused with the specific tolerance on the optimizers used in the sub-problems of BLISS, which remained set at 1×10^{-6} so as to be consistent with the other cases investigated. 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 that 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 to automatically calculate the derivatives of all objectives and constraints with respect to all parameters. If any of the components provides analytic derivatives, those are used automatically instead of running finite difference calculations.

Moore[35] demonstrated how the *SensitivityDriver* could be configured to work with different derivative calculation algorithms: the finite difference method, forward-mode algorithmic differentiation via the chain rule, and matrix solution of the coupled analytic equations. The default method within OpenMDAO is a modified finite difference algorithm where any analytic derivatives present in a component are used to speed up the calculation. The user can reconfigure the *SensitivityDriver* to use one of the other two algorithms or one they wrote themselves, if desired. Moore's work also showed how the built-in capability of OpenMDAO to handle analytic derivatives allows the framework to automatically solve for the coupled system derivatives as part of the MDA. This automatic process in the framework helps to dramatically reduce the initial setup time for architectures that can benefit from the use of coupled derivatives. When applied to the BLISS architecture without any additional work on the part of the user, the result is that when analytic derivatives are available for the components, the cost of calculating the coupled derivatives for the MDA are reduced dramatically.

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[36] is 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. For every one iteration of the system-level problem, a subproblem is also optimized for each discipline with respect to its local variables and constraints. The XDSM for BLISS-2000 is shown in Fig. 7. Note from the XDSM that the major iteration is controlled by a convergence check that operates similar to the one used in BLISS. It checks to see how much the design variables are changing and stops when they differ by less than a given tolerance between two iterations. Again, similar to BLISS, since BLISS-2000 operates on approximations for the true system, it can be difficult to reach tight convergence levels for the design variables. Hence, just like for BLISS, the convergence tolerance of the major iteration loop is set to .005, though the sub-problem optimizations still converge to the tighter 1×10^{-6}.

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 is governed by Eqn. 8, with n being the number of design variables, in a neighborhood around the current point.[37] Equation 8 is only valid for 2^{nd} order polynomial regression models. For other implementations of BLISS, with different types of regression, a different method must be used to select appropriate sample sizes for each discipline.

$$number\ of\ points = \frac{n^2 + 3n + 2}{2} \tag{8}$$

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:

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

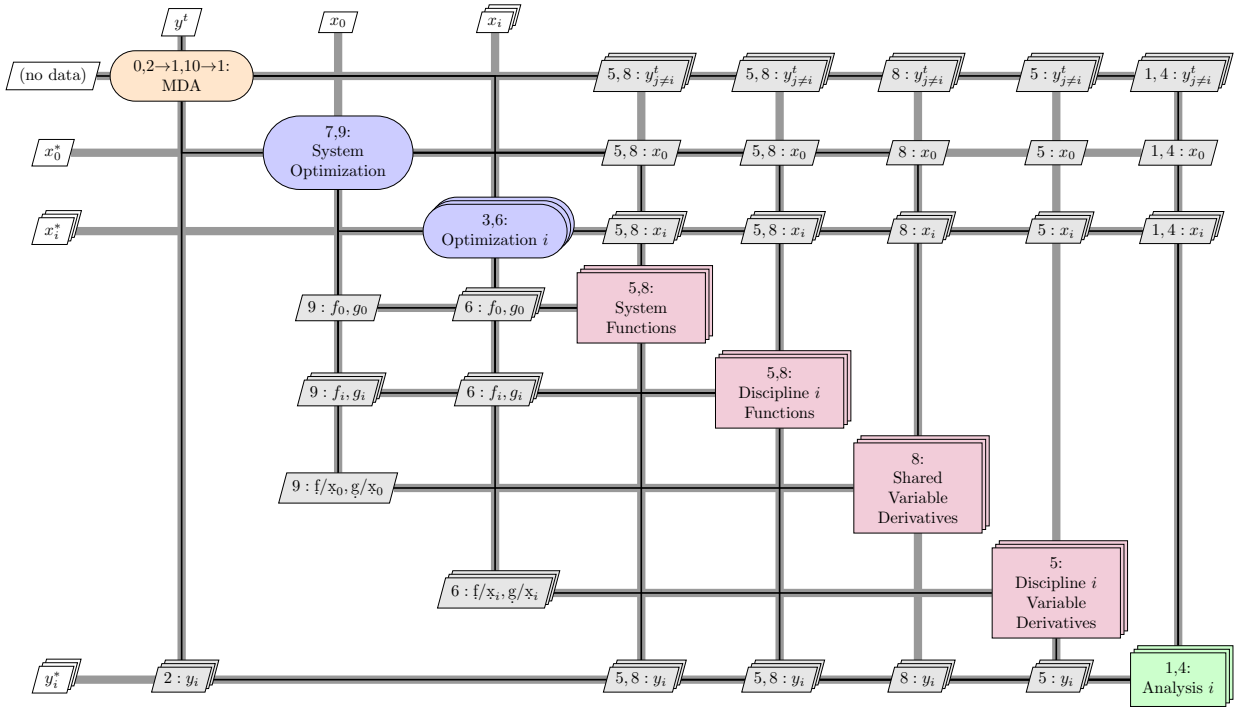

Figure 6: XDSM diagram for BLISS adapted from Martins[1] MDO Architecture Survey.

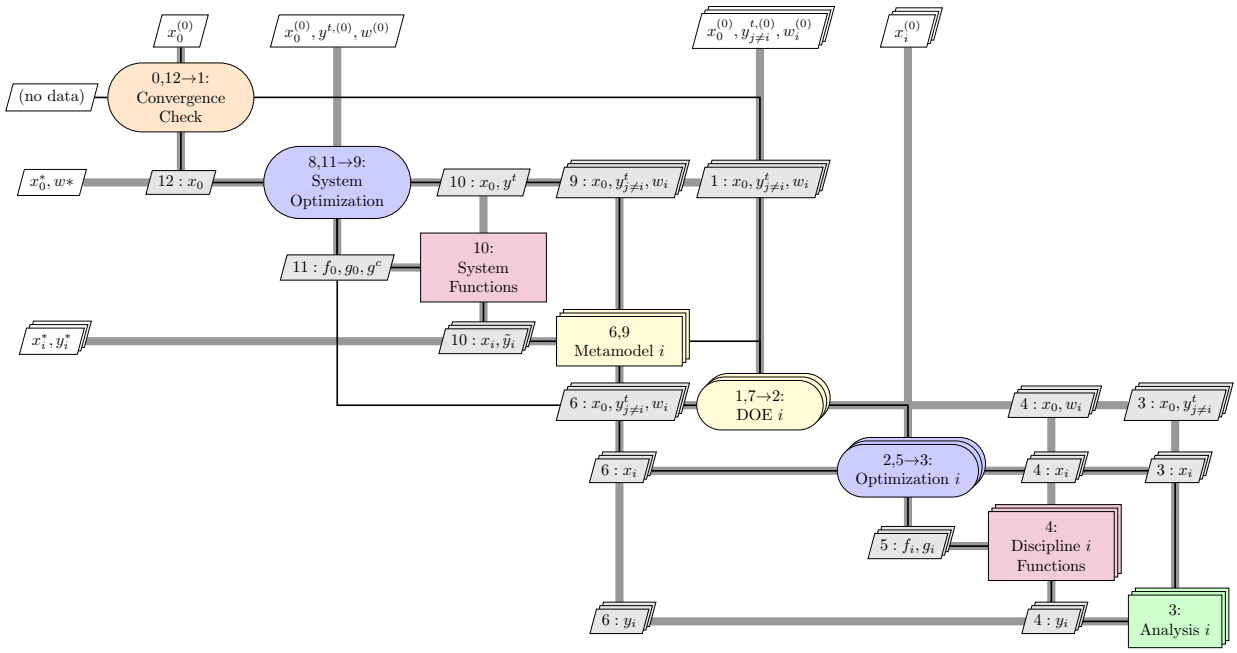

Figure 7: XDSM diagram for BLISS-2000 adapted from Martins[1] MDO Architecture Survey.

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. Secondly, the use of the same implementations of the test problems across different architectures gives a basis for consistent comparison between results, removing a potential source of discrepancy between benchmark results. At present, with only two test problems, the time advantage may be small; however, as the test problem suite grows in size and complexity, the benefit grows as well.

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 serve as templates that could be extended to construct more complex and realistic problems in OpenMDAO. 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.[38] It has since become a commonly used test problem for MDAO architectures.[39,40] The problem is fairly small, having two disciplines and a limited

number of design variables, but it does provide local and global design variables and is sensitive to the selection of initial conditions. 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.[40] It provides the ability to specify the number of local design variables, global design variables, disciplines, and the degree of coupling between them to be 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 variables. 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 the optimal objective value that the architecture 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 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 exact 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 that 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 specifying the exact solution. 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(), that will compare the current state of any instance against its 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 within 1.2% of the exact optimum. MDF was the most accurate and BLISS-2000 the least 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.

For the Sellar Problem with analytic derivatives, the accuracy of IDF and BLISS-2000 were improved to converge within .05%. MDF, CO, and BLISS saw only marginal improvements in their accuracy, since they had already converged to within .05% for the finite difference version of the Sellar Problem. The data in Table 5 shows the results of this test.

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
	Δz_1^*	Δz_2^*	Δx_1^*	Δy_1^*	Δy_2^*	Δ Objective
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
	Δz_1^*	Δz_2^*	Δx_1^*	Δy_1^*	Δy_2^*	Δ Objective
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

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 with respect to the known solution for all three disciplines. Each discipline has three global, three local, and three output variables, but each group takes the same value at the optimal solution. There were variations in the values that the optimizer found for each architecture, but the maximum error indicates how much the calculated optimum deviated from the exact value. For the Scalable Problem, not all of the architectures were able to find an optimal solution. In particular, BLISS could not converge to the proper solution and completes with an objective value three times larger than the initial condition. CO had trouble balancing the constraints and the objective function and could only get to within 300% of the known minimum objective value, though it did reduce its value relative to the initial condition. Once again MDF provided the most accurate convergence, getting the correct answer within .001.

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. It is important, when considering function counts, to consider the overall tolerance that each architecture is working toward. In this case, IDF, MDF, and CO are all governed by the convergence criterion of the global optimization step. A convergence tolerance

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
	Δz^*	Δx^*	Δy^*	Δ Objective
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

of 1×10^{-6} was used for all optimization considered here. However, BLISS and BLISS-2000 are governed by a top level fixed point iteration which checks for convergence on the value of the design variables. Since these architectures rely on approximations of the overall system, they have difficulty converging to very tight tolerance and required a relaxed convergence of .005. Hence, when considering function evaluation counts, it is important to keep in mind that the tighter tolerance of MDF, IDF, and CO.

The data from the optimizations on the Sellar Problem, listed in Table 7, indicates that IDF uses the fewest function calls, followed by MDF. BLISS-2000 uses more function evaluations for Discipline 1 than MDF, but fewer 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 twenty 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*	818	108

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 8. 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

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*	602 (269)	125 (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*	2349	2357	2295

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 50% 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 in particular benefits from the availability of analytic derivatives thanks to OpenMDAO's built-in capability to automatically implement the coupled derivatives for the MDA. 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, which for Sellar is only the case for Discipline 1. 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 also highlights how each architecture sees a different degree of benefit from the presence of analytic derivatives.

The data for the Scalable Problem is listed in Table 9. 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 four 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.

Overall, it is clear from all three test cases that even with their tighter convergence tolerance, IDF and MDF provide the smallest total function evaluation counts. The looser tolerance for BLISS and BLISS-2000

counts even more heavily against them, since they exhibit higher function counts already.

C. Convergence Characteristics

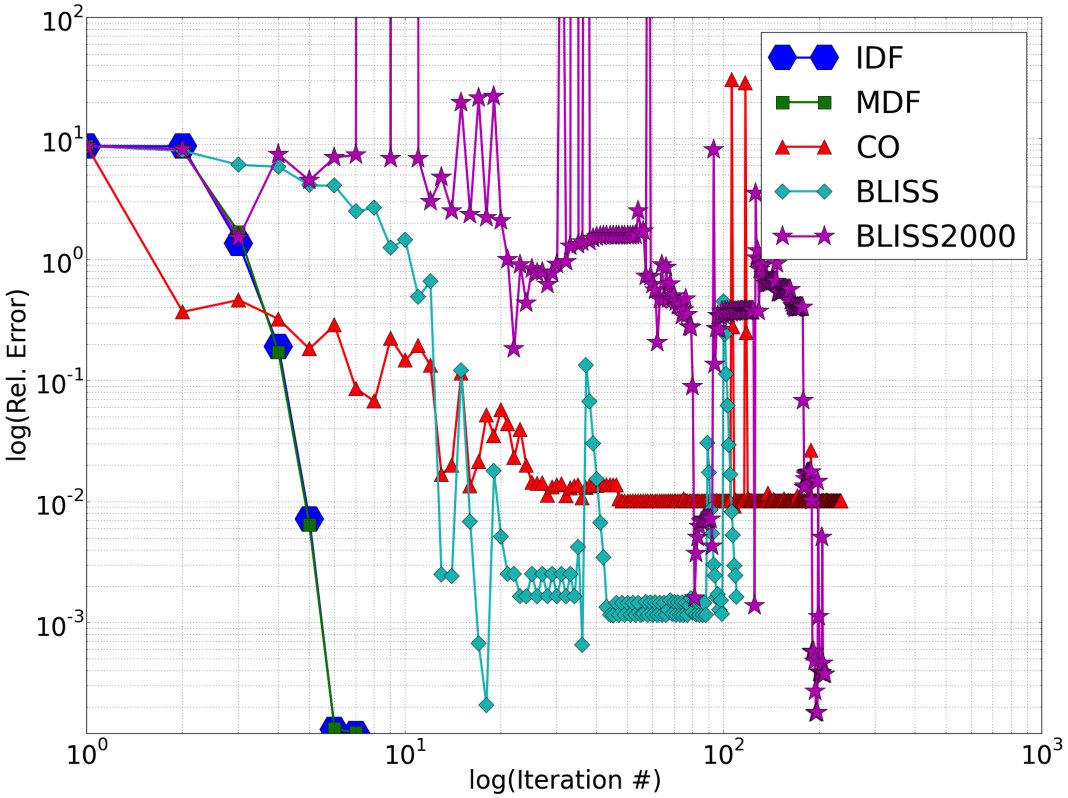

Figure 8: Relative error vs iteration number for all architectures run on the Sellar Problem.

The convergence behavior for all five architectures was tracked for the two test problems. The data presented is relative error vs the global optimizer iteration count (the number of times the global optimizer was run).

The data for the Sellar Problem, in Fig. 8, indicates a clear convergence trend for the MDF and IDF architectures. The slow convergence issues that Braun et al.[33] identified for CO are clearly visible in the data. BLISS demonstrates 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 Fig. 9. 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 lose 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.

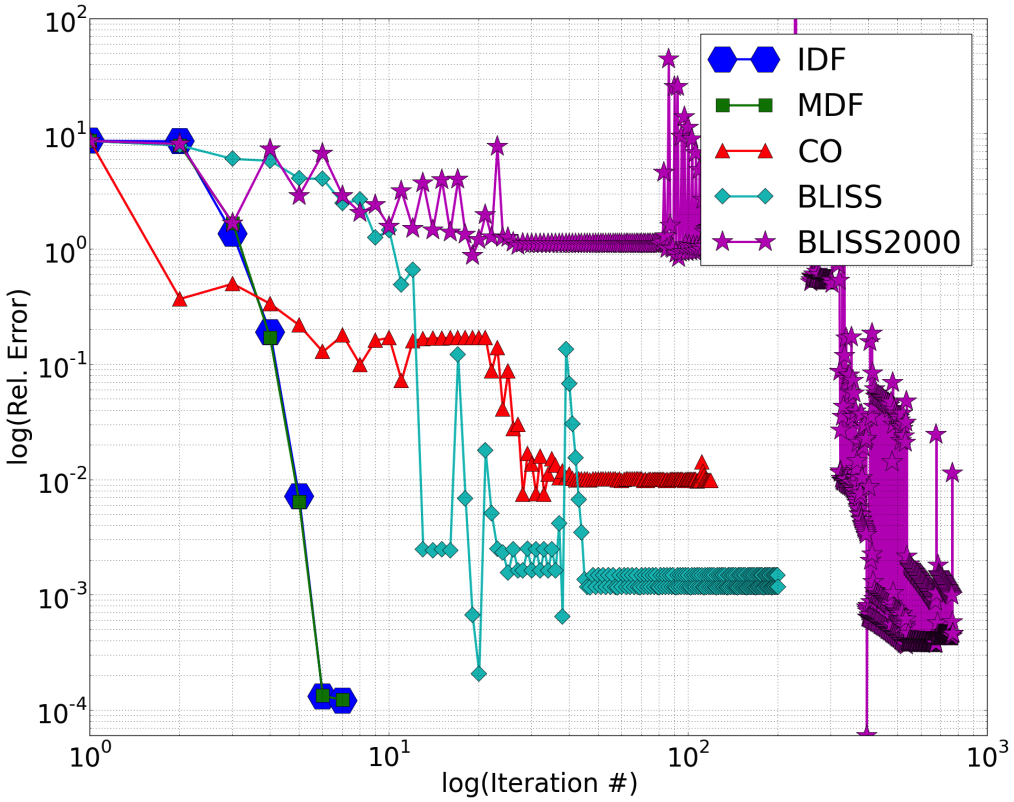

Figure 9: Relative error vs iteration number for all architectures run on the Sellar Problem with analytic derivatives.

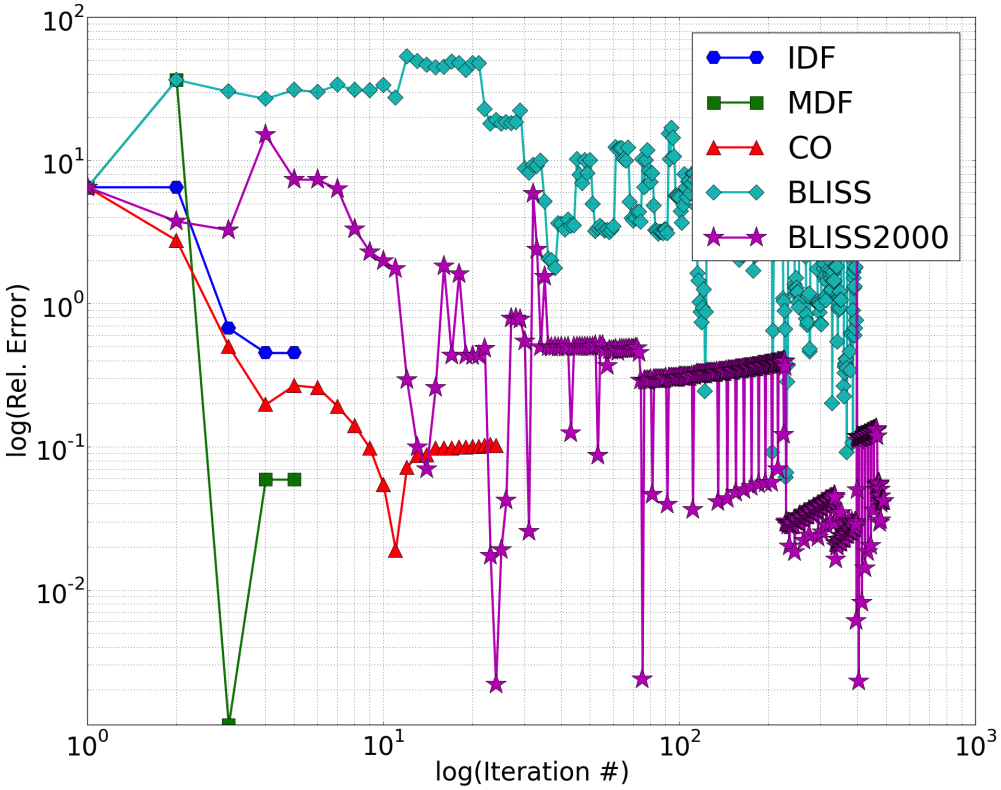

Figure 10: Relative error vs iteration # for all architectures run on the Scalable Problem.

In comparison to the convergence trends from the Sellar Problem, the data for the Scalable Problem in Fig. 10 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. BLISS-2000 shows slow convergence for both Sellar and the Scalable problems, though the stepped behavior is specific to the scalable problem. This behavior is not surprising, as Martins[1] classifies BLISS-2000 along with CO as a Multi-level distributed IDF scheme, so it is reasonable that they would display similar convergence challenges. The erratic convergence for BLISS in the data gives some insight into why it cannot solve the scalable problem. 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 it is 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 is 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 and re-ran the architectures against the Sellar Problem. The COBYLA optimizer is a derivative free optimization algorithm that may be useful for problems where analytic derivatives are not available and finite differencing is not feasible. The results in Table 10 show that with COBYLA the architectures converged to within a maximum error of .2% on the objective value for Sellar Problem. That represents a small improvement in accuracy over the similar results with SLSQP.

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
	Δz_1^*	Δz_2^*	Δx_1^*	Δy_1^*	Δy_2^*	Δ Objective
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 shows that that the number of function evaluations, and hence, the computational cost of

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*	951	123

the optimizations was reduced for IDF, MDF, and BLISS when using COBYLA on the Sellar Problem, as compared with SLSQP with finite differencing. CO and BLISS-2000 showed a negligible change in computational cost. While the number of function evaluations did drop for some architectures, the reduction was not nearly as dramatic as was seen with SLSQP on the Sellar Problem with analytic derivatives. Regardless, if derivatives are not available, COBYLA could be a more efficient choice than SLSQP.

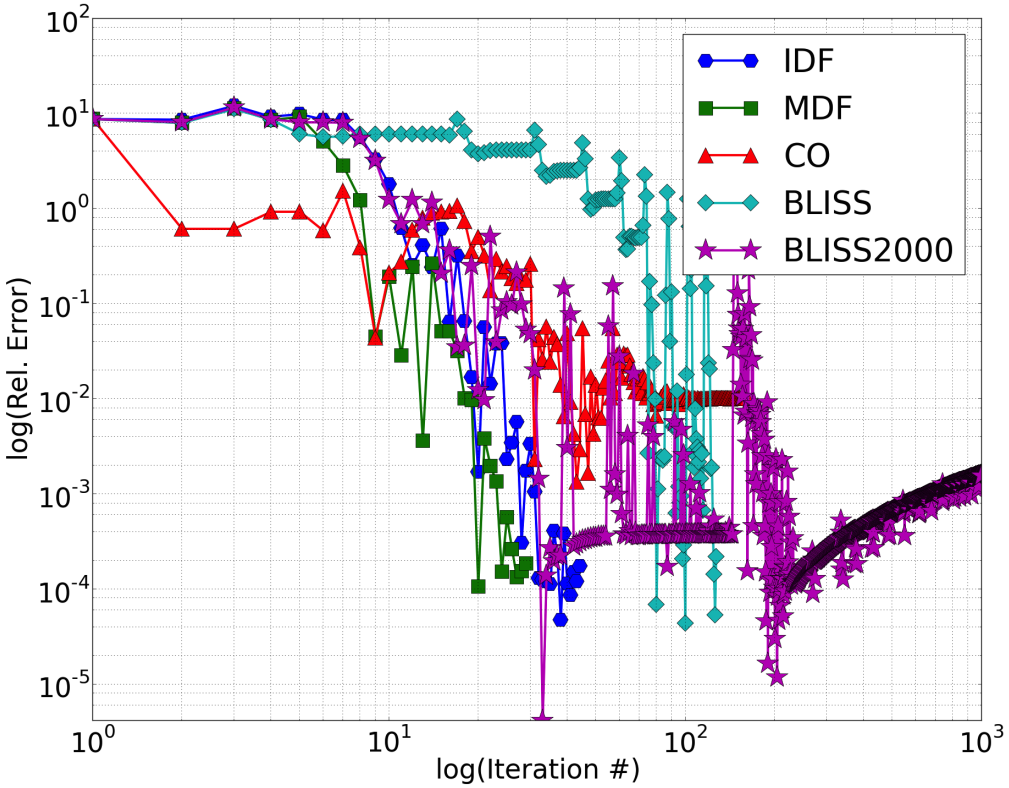

Figure 11: Relative error vs iteration # for all architectures run on the Sellar Problem with the COBYLA optimizer.

In Fig. 11 the convergence data is shown for the COBYLA tests. Most of the architectures display a more noisy convergence path compared to the same problem solved with SLSQP and finite differencing. The one exception is BLISS-2000, which actually displays a smoother convergence when using COBYLA. The convergence behavior does not detract from the fact that the overall function counts were lower with

COBYLA than with SLSQP and finite differencing for IDF, MDF, or CO. This indicates that for problems where analytic derivatives are not available COBYLA may be a better choice. However, comparing the COBYLA results with SLQSP results with analytic derivatives, it is clear that if available, an optimizer that can take advantage of analytic derivatives would be preferable.

VI. Community Contribution

There are multiple other existing MDAO architectures that have not yet been implemented in OpenMDAO, such as Analytical Target Cascading,[41,42] Enhanced Collaborative Optimization,[39] or Asymmetric SubOptimization.[43] There are also additional test problems that need to be implemented to provide a stronger basis for comparing architecture performance. For instance, this study did not include any test problems that 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 A. This code shows that adding new problems to the framework is straightforward.

OpenMDAO is developed and distributed under the Apache V2.0 open source license,[44] 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.[45] 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,[46] which makes it easy for the community to submit code contributions.

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

This work has demonstrated that the OpenMDAO framework has the necessary features to address all four requirements laid out for a successful MDAO benchmarking tool. Because it is distributed as an open source tool, it can easily serve as a common platform for further research. 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 on 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 automatic application of these architectures to any given set of test problems gives OpenMDAO the ability to support a large expansion of the test problem suite without requiring a re-implementation of the architectures. Similarly, additional architectures can be added without re-implementing all of the test problems.

The results of the test-suite runs 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 that 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.

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. The comparison between the SLSQP and COBYLA results on the Sellar Problem also indicate that optimizer choice can have an impact on architecture performance. Over all, these results re-enforce that the "No Free

Lunch" theorem holds even more strongly for MDAO architectures than it does for single optimizers. There are far more factors, including parallel potential, problem characteristics, and details of the architecture implementation, that will have an impact on which architecture is most appropriate for a given problem.

VIII. Moving Forward

Since only two test problems were considered in this work, the resulting benchmarks cannot be considered definitive. A larger set of test problems in OpenMDAO is needed to better indicate to potential users which architectures would be effective for their needs. In addition, since these results make it clear that architecture performance is impacted by many factors, potentially different implementations of each architecture should be considered as part of the test suite.

Developing a broad test suite is a challenge that can only be conquered by cooperative efforts of the greater MDAO community. The collection of problems that comprise the test suite is something that needs to be constantly updated and expanded to include challenging problems from a wide range of technical fields. Additionally, as new architectures are developed, running them on the test set would provide the consistent performance comparisons against established architectures that are needed to give a solid basis assessment.

Traditionally, two main factors have hampered efforts to build and maintain a comprehensive test suite: the upfront cost of implementing any given MDAO architecture for a large number of problems and the time cost of implementing the problems themselves. OpenMDAO provides a solution to both problems through the ability to apply architectures automatically across the test suite and an open-source contribution process. OpenMDAO can be delivered with the full set of test problems already implemented, which frees a researcher to focus on proper implementation of their new architecture. Then researchers can contribute their new architectures back to the OpenMDAO code base, which greatly reduces other researchers' upfront implementation cost.

Moving forward, community contributions are needed to continually expand and improve the MDAO test suite. GitHub and other community coding tools like it are a necessary addition to the toolbox of MDAO research so they can efficiently share each other's implementations, similar to how we leverage publications to share our theories and results. Using the OpenMDAO code-hosting site on GitHub, researchers could develop and distribute new problems to serve both as benchmark cases for existing architectures and also as challenges for new MDAO architectures and methods. These problems would then be immediately available to the entire community of users for their own work.

Beyond just sharing code, establishing a consistent test procedure and providing an open-source implementation for it 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. However, by adopting the practice of shared code development via a common research platform like OpenMDAO and making use of open-source development tools, researchers can dramatically accelerate the pace of and broaden the potential user base for their work.

References

[1]Martins, J. R. R. A. and Lambe, A. B., "Multidisciplinary Design Optimization: Survey of Architectures," *AIAA Journal*, 2012, pp. 1–46.

[2]Alexandrov, N. and Kodiyalam, S., "Initial results of an MDO method evaluation study," *Proceedings, Seventh AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, No. 98-4884, Sept. 2-4 1998.

[3]Alexandrov, N. and Lewis, R., "Comparative properties of collaborative optimization and other approaches to MDO," *8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 1999.

[4]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.

[5]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.

[6]Kodiyalam, S., "Evaluation of methods for multidisciplinary design optimization (MDO), Phase I," Tech. rep., NASA/CR-1998-208716, Hampton, Virginia, September 1998.

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

[8]Kodiyalam, S. and Sobieszczanski-Sobieski, J., "Multidisciplinary design optimization-some formal methods, framework requirements, and application to vehicle design," *International Journal of Vehicle Design*, Vol. 25, No. 1, 2001, pp. 3–22.

[9]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.

[10]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.

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

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

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

[14]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.

[15]Sobolewski, M., "Object-Oriented Service Clouds for Transdisciplinary Computing," *Cloud Computing and Services Science*, 2012, pp. 3–31.

[16]Sobolewski, M., Kolonay, R., Czejdo, B., Ferragut, E., Goodall, J., Laska, J., Burrell, A., Papantoni-Kazakos, P., Semushin, I., Haider, W., et al., "Unified Mogramming with Var-Oriented Modeling and Exertion-Oriented Programming Languages," *Int'l J. of Communications, Network and System Sciences*, Vol. 5, No. 29, 2012, pp. 579–592.

[17]Kolonay, R. and Sobolewski, M., "Service ORiented Computing EnviRonment (SORCER) for large scale, distributed, dynamic fidelity aeroelastic analysis and optimization," *2011 International Forum on Aeroelasticity and Structural Dynamics*, 2011.

[18]Felder, J., Kim, H., and Brown, G., "Turboelectric Distributed Propulsion Engine Cycle Analysis for Hybrid-Wing-Body Aircraft," *47th AIAA Aerospace Sciences Meeting*, American Institute of Aeronautics and Astronautics, Jan 2009.

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

[20]Moore, K., Naylor, B., and Gray, J., "The development of an open source framework for multidisciplinary analysis and optimization," *Proceedings of the 12th AIAA/ISSMO multidisciplinary analysis and optimization conference, Victoria, BC, Canada, AIAA paper*, Vol. 6069, Victoria, British Columbia, 2008, pp. 1–13.

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

[22]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.

[23]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.

[24]Gray, J. S., "Setting up a Basic Model With Paraboloid," OpenMDAO User Guide, V0.4.0, 2012, http://openmdao.org/releases/0.4.0/docs/basics/assembly.html.

[25]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.

[26]Moore, K. T., "Wrapping an External Module Using F2PY," OpenMDAO User Guide, V0.4.0, 2012, http://openmdao.org/releases/0.4.0/plugin-guide/extension_plugin.html.

[27]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.

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

[29]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.

[30]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.

[31]Cramer, E., Dennis, J. J., Frank, P., Lewis, R., and Shubin, G., "Problem Formulation for Multidisciplinary Optimization," *SIAM Journal on Optimization*, Vol. 4, No. 4, 1994, pp. 754–776.

[32]Alexandrov, N. M. and Lewis, R. M., "Analytical and Computational Aspects of Collaborative Optimization for Multidisciplinary Design," *AIAA Journal*, Vol. 40, No. 2, Feb. 2002, pp. 301–309.

[33]Braun, R. D., Gage, P. J., Kroo, I. M., and Sobiesiki, I., "Implementation and Performance Issues in Collaborative Optimization," Sixth AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, 1996.

[34]Sobieszczanski-Sobieski, J., Agte, J., Sandusky, R., and Center, L. R., "Bi-level integrated system synthesis (BLISS)," Citeseer, 1998, pp. 1543–1557.

[35]Moore, K. T., ""Calculation of Sensitivity Derivatives in an MDAO Framework"," *14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Indianapolis, Indiana, September 2012.

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

[37]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.

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

[39]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*, Victoria, British Columbia, 2008.

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

[41]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.

[42]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.

[43]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.

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

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

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

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
```