pyMDO: An Object-Oriented Framework for Multidisciplinary Design Optimization

JOAQUIM R. R. A. MARTINS, CHRISTOPHER MARRIAGE, and NATHAN TEDFORD
University of Toronto Institute for Aerospace Studies

20

We present pyMDO, an object-oriented framework that facilitates the usage and development of algorithms for multidisciplinary optimization (MDO). The resulting implementation of the MDO methods is efficient and portable. The main advantage of the proposed framework is that it is flexible, with a strong emphasis on object-oriented classes and operator overloading, and it is therefore useful for the rapid development and evaluation of new MDO methods. The top layer interface is programmed in Python and it allows for the layers below the interface to be programmed in C, C++, Fortran, and other languages. We describe an implementation of pyMDO and demonstrate that we can take advantage of object-oriented programming to obtain intuitive, easy-to-read, and easy-to-develop codes that are at the same time efficient. This allows developers to focus on the new algorithms they are developing and testing, rather than on implementation details. Examples demonstrate the user interface and the corresponding results show that the various MDO methods yield the correct solutions.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; G.1.6 [**Numerical Analysis**]: Optimization—*Nonlinear programming, constrained optimization*; G.4 [**Mathematical Software**]: *Algorithm design and analysis*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Multidisciplinary design optimization, object-oriented programming

ACM Reference Format:
Martins, J. R. R. A., Marriage, C., and Tedford, N. 2009. pyMDO: An object-oriented framework for multidisciplinary design optimization. ACM Trans. Math. Softw. 36, 4, Article 20 (August 2009), 25 pages. DOI = 10.1145/1555386.1555389. http://doi.acm.org/10.1145/1555386.1555389.

The authors are very grateful for the support provided by the Canada Research Chairs program and the Natural Sciences and Engineering Research Council.
Authors' addresses: J. R. R. A. Martins, University of Toronto Institute for Aerospace Studies, 4925 Dufferin St., Toronto, ON M3H 5T6, Canada; email: martins@utias.utoronto.ca; C. Marriage and N. Tedford, email: {chris.marriage, nathan.tedford}@utoronto.ca.

1. INTRODUCTION

Multidisciplinary design optimization (MDO) is an area related to the design of engineering systems. MDO problems require execution of several simulations, each representing a specific engineering discipline. For example, in aerospace engineering, the state of an aircraft wing and outputs of interest are found by solving both the aerodynamic and structural state partial differential equations (PDEs). Thus, when the outputs—such as the drag coefficient—are used as objectives or constraints in an optimization problem, they depend on the states of both disciplines, as well as on design variables associated with both the aerodynamic shape and structural sizing.

Various MDO problem solution approaches exist, each requiring different degrees of autonomy for the disciplinary subsystem computations. The approaches range from fully integrated, where a single optimizer controls coupled multidisciplinary simulations, to multilevel distributed methods that optimize disciplinary objectives in their respective domains.

Design variables can be classified as *global* or *local*. Local design variables directly affect the states of only one discipline, while global design variables affect two or more.

For a given discipline i, the state y_i is obtained by the respective simulation. We can write a simulation i as sets of equality constraints,

$$\mathcal{R}_i\left(z, x_i, y_i(z, x, y_j)\right) = 0, \quad j = 1, \ldots, i-1, i+1, \ldots, N, \tag{1}$$

where z are the global design variables, x the local variables, and N the number of disciplines. Note that each simulation is only affected directly by the variables local to its discipline, x_i. In general, the simulation might also depend on the states off all other disciplines, y_j.

A general MDO problem can be stated as

$$\begin{aligned} \text{minimize} \quad & f\left(z, x, y_i(z, x, y_j)\right) \\ \text{with respect to} \quad & z, x \\ \text{subject to} \quad & c\left(z, x, y_i(z, x, y_j)\right) \geq 0, \end{aligned} \tag{2}$$

where f is the objective function, c is the vector of constraints, and y is the vector of states for all disciplines, which are determined by the multidisciplinary simulations (1). The variables in y can also be seen as the *coupling variables*, since each subset y_i can potentially affect all disciplines $j \neq i$.

MDO formulations with single-level optimization are well understood [Cramer et al. 1994]. Various multilevel formulations have been proposed, but their practicality is under investigation, and their theory has not been unified to the same degree.

A number of articles discussed the relative performance of various MDO methods [Padula et al. 1996; Sobieszczanski-Sobieski and Haftka 1997; Alexandrov and Kodiyalam 1998; Kodiyalam 1998; Kodiyalam and Yuan 2000; Perez et al. 2004; Brown and Olds 2006; Tedford and Martins 2006a]. They considered a small number of problems—ranging from one to 10—because

statistical comparisons were outside the scope of those efforts. However, most methods must ultimately be compared on statistical grounds, and in that case a framework for a unified implementation of MDO methods becomes extremely useful. Such a comparison must necessarily include a set of problems that is large enough for the results to be statistically significant. Even if a suite with a large number of MDO problems existed, a very large effort would be required to implement all known MDO methods for every problem, since current implementation approaches are formulation specific and problem specific, and do not emphasize reusability. Furthermore, researchers who wish to test a new MDO algorithm must develop their code from scratch, including the code of one or more existing algorithms for benchmarking.

There has been, however, some progress in toward reusability. Alexandrov and Lewis [2004a; 2004b] developed a standard abstract syntax for MDO problems and methods that enables the reusability of basic components. The development of pyMDO was partially motivated by these seminal articles.

The value of using object-oriented programming for numerical optimization has already been recognized, especially in cases where multiple algorithms are required in the same software framework. DAKOTA, for example, is a software toolkit developed at Sandia National Laboratories that provides various optimization algorithms, parameter estimation, uncertainty quantification with sampling, reliability, sensitivity analysis, design of experiments, and parameter study capabilities [Eldred et al. 1996; 2006]. Another open-source project worth mentioning is OPT++, developed by Meza et al. [2007]. This is an object-oriented toolkit for nonlinear optimization that makes use of inheritance to provide a common interface to users, while making it easy for developers to add new algorithms.

The goal of this research is to develop a framework for MDO algorithms that facilitates the following:

—*Use.* Ease of use is achieved by defining a common, algorithm-independent interface for describing MDO problems. For general users, the aim is to provide an interface with which the MDO problem is easily stated. pyMDO is then responsible for implementing the various MDO methods automatically.

—*Development.* This is accomplished by using object-oriented programming concepts such as inheritance and operator overloading, which greatly facilitate reusability. These features make it easy for developers to implement new MDO methods and fine tune existing ones.

—*Benchmarking.* By making it easy to describe problems in an algorithm-independent manner, developers can benchmark their own algorithm against existing MDO algorithms more promptly. Furthermore, the creation and maintenance of a benchmarking suite will require much less effort.

This article is laid out as follows. The next section outlines MDO methods implemented in the framework. Section 3 describes the software design of pyMDO and Section 4 details the implementation. Examples of usage and numerical results are presented in Section 5. Conclusions and future work are presented in the final section.

2. MDO METHODS

In this section, we present a brief overview of the MDO methods that are implemented in pyMDO. More details on these methods can be found in the literature [Cramer et al. 1994; Kroo 1997; Tedford and Martins 2006b].

2.1 Multidisciplinary Design Feasible

The traditional approach to MDO has been to solve the set of disciplinary equations to convergence at each step of the optimization. Cramer et al. [1994] named this approach *multidisciplinary feasible* (MDF). This approach has been used in engineering since the inception of MDO [Haftka and Gürdal 1993; Sobieszczanski-Sobieski and Haftka 1997; Kroo 1997]. It consists in solving the original problem (2), that is,

$$\begin{aligned} \text{minimize} \quad & f\left(z, x, y(x, z)\right) \\ \text{with respect to} \quad & z, x \\ \text{subject to} \quad & c\left(z, x, y(x, z)\right) \geq 0. \end{aligned} \tag{3}$$

For each iteration, the multidisciplinary state is found by solving the coupled system of N simulations,

$$\mathcal{R}_i\left(z, x_i, y_i(z, x, y_j)\right) = 0\,, \tag{4}$$

where $i = 1, \ldots, N$ corresponds to each and every discipline. For a given discipline, the subscript j represents all disciplines except the local one, that is, $j = 1, \ldots, i-1, i+1, \ldots, N$. The state of a simulation i affects another simulation j through the coupling variables. In general, these variables are functions of the states. Thus, for the multidisciplinary simulation to converge, it is not sufficient that the state within each simulation converge: the coupling variables must converge as well. The coupled simulation is typically solved by a block-iterative procedure and is considered to be converged when the coupling variables generated by each discipline analysis remain constant (within a specified tolerance) over successive iterations. This means that feasibility with respect to the simulation constraints (4) is enforced at each optimization iteration.

The MDF method is practical for engineers because complex single- and multidisciplinary simulations have been used well before numerical optimization was applied to engineering problems.

2.2 Individual Discipline Feasible

The individual discipline feasible (IDF) method [Cramer et al. 1994; Kroo 1997] removes the need for the iterative procedure that ensures multidisciplinary feasibility in the traditional approach by removing direct communication between the disciplinary simulations. Each discipline is solved in isolation—possibly in parallel—and the optimizer is made responsible for the convergence

of the multidisciplinary state. The optimization problem can be stated as follows:

$$\begin{aligned} \text{minimize} \quad & f\left(z, x, y^t\right) \\ \text{with respect to} \quad & z, x, y^t \\ \text{subject to} \quad & c\left(z, x, y\left(x, y^t, z\right)\right) \geq 0 \\ & y_i^t - y_i\left(x, y_j^t, z\right) = 0, \end{aligned} \tag{5}$$

where y^t represents the targe value for the coupling variables chosen by the optimizer. The states of each discipline, y_i are determined by each simulation, that is, they are such that

$$\mathcal{R}_i\left(z, x_i, y_i(z, x, y_j^t)\right) = 0, \tag{6}$$

where the state of the other disciplines, y_j^t for $j \neq i$ is determined by the optimizer.

Using IDF, there is no need for an iterative procedure to converge the multidisciplinary simulation at each optimization iteration. Consequently, each simulation can be run in parallel. This approach has also been referred to as *optimizer-based decomposition* [Kroo 1997] .

2.3 Simultaneous Analysis and Design

Simultaneous analysis and design (SAND) goes beyond IDF and decouples the multidisciplinary problem further by treating the governing equations for each simulation (1) as equality constraints in the optimization problem. The SAND formulation can be written as a single optimization problem:

$$\begin{aligned} \text{minimize} \quad & f\left(z, x, y\left(z, x, y\right)\right) \\ \text{with respect to} \quad & z, x, y \\ \text{subject to} \quad & c\left(z, x, y\left(z, x, y\right)\right) \geq 0 \\ & \mathcal{R}_i\left(z, x_i, y_i\left(z, x, y_j\right)\right) = 0, \quad i = 1, \ldots, N, \end{aligned} \tag{7}$$

where $\mathcal{R}_i$ represents the residuals of the governing equations for each simulation.

This method is usually impractical for MDO involving large simulations, such as PDEs in a three-dimensional domains, since specialized algorithms outperform optimization algorithms when solving these large systems of equations. To use the SAND approach, it is required that the simulation be in residual form. Another disadvantage of this method is that if the optimization process stops before convergence, there is no guarantee that the design is physically feasible.

2.4 Collaborative Optimization

Collaborative optimization (CO) [Schmit and Ramanathan 1978; Thareja and Haftka 1986; Braun and Kroo 1997; Sobieski and Kroo 2000] is a bilevel MDO approach designed to provide discipline autonomy while maintaining interdisciplinary compatibility. The optimization problem is decomposed into a

number of independent optimization subproblems, each corresponding to one discipline. Each disciplinary optimization is given control over its (local) design variables and is responsible for satisfying its (local) constraints.

The CO system-level problem is given by

$$\begin{aligned} \text{minimize} \quad & f\left(z^t, x_{\text{obj}}, y^t\right) \\ \text{with respect to} \quad & z^t, x_{\text{obj}}, y^t \\ \text{subject to} \quad & c_{\text{global}}(z^t, y^t) \\ & J_i^* = 0, \qquad i = 1, \ldots, N, \end{aligned} \tag{8}$$

where each J_i^* is a measure of interdisciplinary compatibility that is the solution of the subproblem corresponding to discipline i and N is the number of disciplines. The vector x_{obj}^t represents local design variables that directly affect the objective function, and c_{global} are the global constraints, that is, constraints that depend directly on global variables.

Each discipline subproblem i can be stated as

$$\begin{aligned} \text{minimize} \quad & J_i = \|z_i - z_i^t\|^2 + \|y_i - y_i^t\|^2 \\ \text{with respect to} \quad & z_i, x_i \\ \text{subject to} \quad & c_i\left(x_i, z_i, y_i\left(x_i, y_j^t, z_i\right)\right) \geq 0. \end{aligned} \tag{9}$$

For each discipline, a simulation

$$\mathcal{R}_i\left(z, x_i, y_i(z, x, y_j^t)\right) = 0 \tag{10}$$

enforces the governing equations and determines the local state variables, y_i.

The advantage of CO is that each discipline has a certain autonomy, where the burden of solving for the sets of local variables and satisfying local constraints is distributed. Each discipline can be solved in parallel, which is advantageous from the computational point of view. Also, the disciplinary autonomy in the CO formulation mimics the structure and processes used in industry, and therefore this formulation is likely to be easier to implement in the industrial setting.

In spite of these benefits, CO is not without its drawbacks. As the number of coupling variables increase, the dimensionality of the system level problem increases, as does the number of variables involved with the calculation of the system level compatibility constraints. Therefore, CO tends to be most effective on problems for which the number of local variables and states is much larger than the number of coupling variables. Furthermore, because CO is a bilevel programming formulation, it suffers from the difficulties inherent in nonlinear bilevel programming [Alexandrov and Lewis 2002; DeMiguel and Murray 2006].

2.5 Concurrent Subspace Optimization

Concurrent subspace optimization (CSSO) [Sobieszczanski-Sobieski 1988; Wujek et al. 1997] is another bilevel MDO method that uses approximations to model the mutual effect of coupling variables.

The CSSO system-level optimization problem can be stated as

$$\begin{aligned} \text{minimize} \quad & f\left(z, x, \tilde{y}(z, x)\right) \\ \text{with respect to} \quad & z, x \\ \text{subject to} \quad & c\left(z, x, \tilde{y}(z, x)\right) \geq 0, \end{aligned} \tag{11}$$

where $\tilde{y}$ represents a surrogate model or response surface of the coupling variables.

Within each discipline i, the following subspace optimization problem is solved:

$$\begin{aligned} \text{minimize} \quad & f\left(z_i, x_i, y_i\left(z_i, x_i, \tilde{y}_j\right), \tilde{y}_j\right) \\ \text{with respect to} \quad & z_i, x_i \\ \text{subject to} \quad & c\left(z_i, x_i, y_i\left(z_i, x_i, \tilde{y}_j\right), \tilde{y}_j\right) \geq 0, \end{aligned} \tag{12}$$

where z_i and x_i represent the global and local variables that directly affect discipline i. The complements of these respective sets of variables do not affect discipline i in a direct manner and are kept constant in this subspace optimization. The complement of y_i is approximated by $\tilde{y}_j$, which is the vector of nonlocal coupling variables given by the surrogate model.

3. SOFTWARE DESIGN

We now explain the choice of programming language and outline our software design principles, keeping in mind that our objective is to design a framework that is flexible and efficient, while being straightforward to use.

3.1 Language Selection

We considered a number of programming languages and selected Python [Beazley 2006], which has a large following in the scientific computing community [Langtangen 2004].

Python is an interpreted language that can be run in interactive mode, making it easy to learn and debug. The downside of being an interpreted language is that it is much slower than a compiled one. However, because Python excels at interfacing with other languages, the most numerically intensive computations are usually implemented in C/C++ or Fortran. Python is then used as the high-level language connecting the various numerical simulations. Wrapping C code with Python is straightforward since Python was designed to interface directly with C. It is more involved to wrap C++, but tools such as SWIG automate this process [Blezek 1998]. For Fortran code, we use `f2py`, a tool that automatically creates Python modules that can access all of the functionality of the Fortran code [Peterson et al. 2001]. This wrapping procedure has been tested with various combinations of compilers and platforms [Alonso et al. 2004].

Python is very convenient for scripting but is also a full-fledged programming language that supports object-oriented programming. These features

are particularly important for our purposes. A variety of basic data types are available in Python: numbers (integers, floating point, complex, and unlimited-length long integers), strings (both ASCII and Unicode), and container objects (lists and dictionaries). Lists are similar to arrays, but more flexible since they do not have a fixed size and can be nested. A dictionary is a list whose elements are associated to a keyword, rather than indices.

The language also supports user-defined raising and catching of exceptions, resulting in cleaner error handling. It also does automatic garbage collection that frees the programmer from the burden of memory management.

Python runs on many different computers and operating systems, and provides numerous standard libraries that support many common programming tasks such as connecting to web servers, regular expressions, and file handling.

In addition to the useful features we just described, a number of available Python tools are specially useful in our work, in particular, the scientific computing module, `NumPy`, and the parallel computing module, `pyMPI`.

3.2 Design Principles

The design and implementation of pyMDO are based on the following principles:

—*Clarity.* Implementations of MDO methods should resemble the respective mathematical formulations. This is in contrast to lower-level compiled languages, which often require complicated function calls or subroutines with a long list of parameters. When developing this framework, we strove to provide an intuitive user interface. Our intention is that it should be possible for this framework to be usable by someone who has only a basic knowledge of Python and optimization. All classes and methods automatically use a set of default parameters that can be tuned by overwriting or modifying the default values.

—*Flexibility.* This is essential for creating a truly useful framework that can be used for any MDO problem. The ability of Python to interface with any code greatly enhances flexibility. The end result is a framework that can solve a wide range of problems: from small examples coded in Python to multiple large-scale parallel simulations coded in compiled languages. Another feature of pyMDO that adds to its flexibility is that the numerical optimization modules are interchangeable: one could even use different optimization packages for the various subproblems of a hierarchical MDO method.

—*Extensibility.* MDO methods are far from being completely understood for all classes of problems. It is important for the framework to be easily extended so as to develop and validate new methods. The use of inheritance and operator overloading eases this process immensely.

—*Portability.* The framework should be easily ported across computer platforms. Since Python is available for all platforms, this is not an issue for pyMDO and its portability is only limited by the user's simulation code.

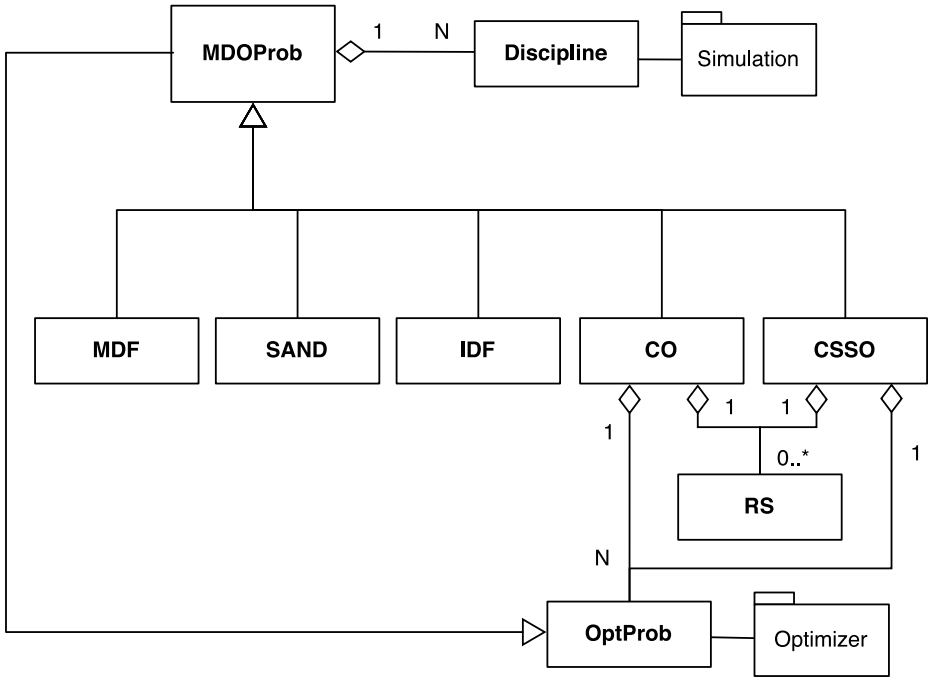

Fig. 1. UML class diagram for pyMDO.

4. IMPLEMENTATION

The classes in pyMDO are based on the mathematical objects defined by the optimization problem (2) and the various MDO formulations. The main base class is `MDOProb`, which represents an MDO problem and contains one or more instances of the `OptProb` class. All MDO method classes are derived from `MDOProb` by inheritance. The relationships between the various classes are illustrated in Figure 1, the details of which we describe in this section. The figures throughout this section use the standard unified modeling language (UML) representation for class diagrams [O'Docherty 2005].

4.1 The `OptProb` Class

The `OptProb` class represents a single optimization problem. This class can use an arbitrary optimization package (shown as "Optimizer" in Figure 1) as long as it is wrapped with Python. In this work we have used SNOPT, which is a Fortran SQP package developed by Gill et al. [2002]. The code was wrapped with Python and encapsulated into the `OptProb` class [Alonso et al. 2004].

A UML representation of the `OptProb` class is shown in Figure 2. The main attributes of this class are a dictionary of variables (`vars`), which contains the design and state variables of the problem, and the objective function value (`obj_value`). The class methods include the evaluation of the objective and constraints (`eval_objective` and `eval_constraints`, respectively), as well as the computation of sensitivities and the main call to the optimizer (`optimize`).

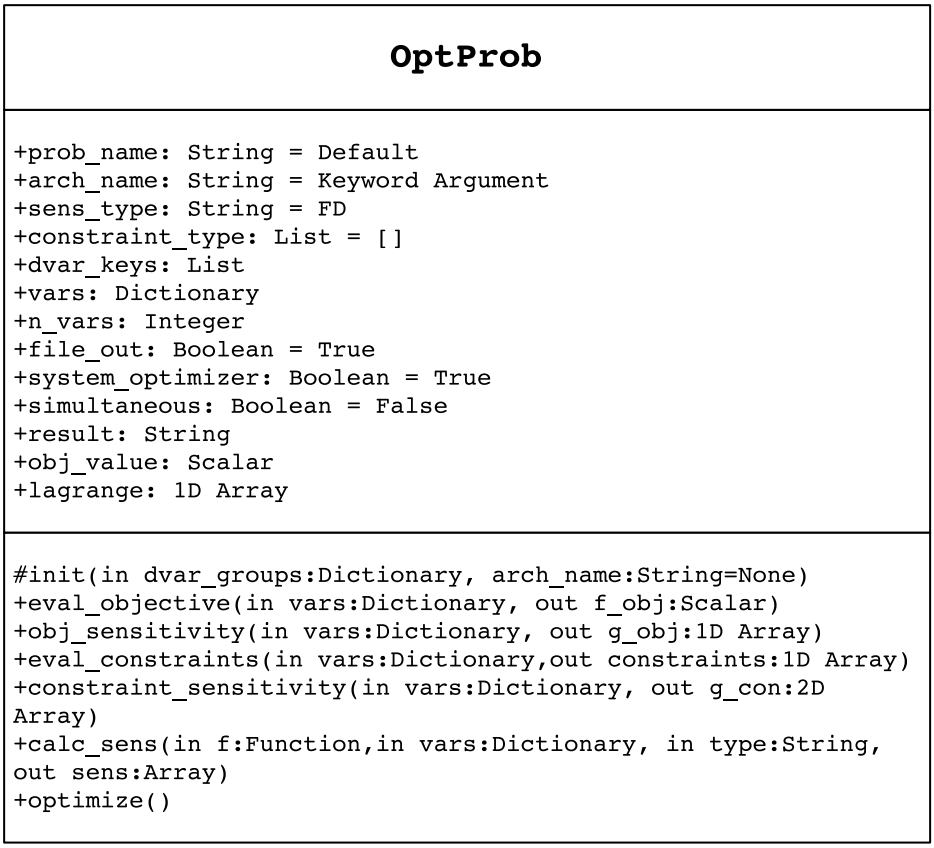

Fig. 2. UML diagram for the `OptProb` class.

The default sensitivity analysis method is finite differencing, but users are free to overwrite the default method definitions and provide their own sensitivities.

4.2 The `Discipline` Class

As previously mentioned, MDO problems typically require multiple simulations, each of them representing a discipline. To make use of pyMDO, the simulations can be programmed in any language as long at they are wrapped with Python using the `Discipline` class. The UML diagram for this class is shown in Figure 3. The attributes include lists of input and output variables, as well as a list of the variables that are local to the discipline. This information is used by specific methods to reorganize the MDO problem and also to decide which variables to read from the main vars dictionary and apply as inputs or outputs to the analysis.

The definition of these input and output variables for each discipline is very important. The input variables are fixed within a given discipline and can be states given by other disciplines or functions of those states. A given discipline outputs its own states, or functions of those states. Furthermore, each

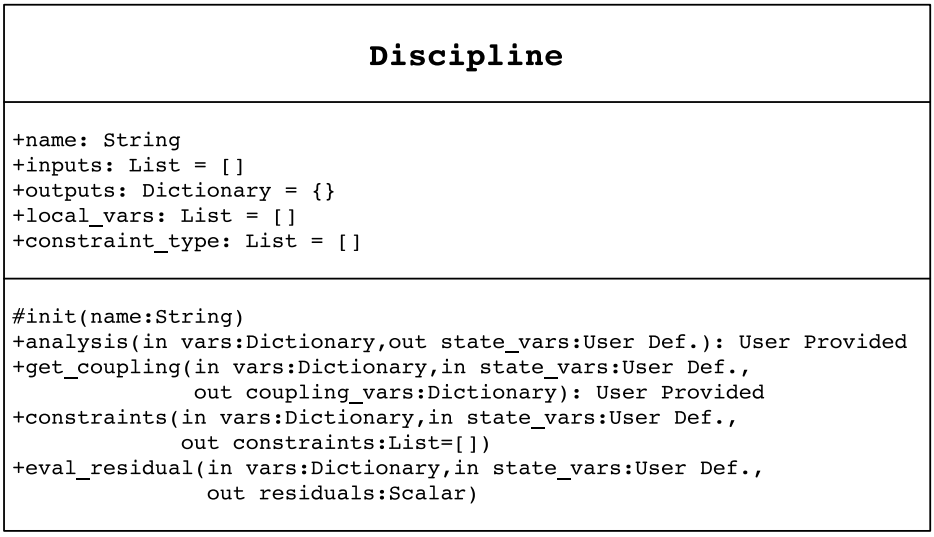

Fig. 3. UML diagram for the `Discipline` class.

discipline might have a number of local design variables as well as local constraints. All this information is essential at the MDO problem level to perform decomposition when the chosen method demands it.

4.3 The `RS` Class

The `RS` class provides a simple yet effective surrogate model—in the form of a quadratic response surface—to the pyMDO framework. Used by CSSO and certain implementations of CO [Sobieski and Kroo 2000], this class provides an approximation of the output of a given function based on the variables passed to it. The UML diagram for `RS` class is shown in Figure 4.

The main attributes of the `RS` class are the `foi`, or function of interest, a list of the input variables `opt_keys`, and the coefficients and bounds of the response surface `coefficients` and `RS_bounds` respectively. The methods include `evaluate`, which returns the approximate function value for given input variables, and `update` and `regenerate`, which modify or completely regenerate the approximation model in response to motion within the design space. The class structure of pyMDO allows this quadratic model to be easily replaced with, or run concurrently with, other approximation methods.

4.4 MDO Methods

4.4.1 *The* `MDOProb` *Base Class.* Since the MDO problem is an optimization problem, it is only natural that the `MDOProb` class (shown in Figure 5) inherits its basic attributes and methods from the `OptProb` class. Additional attributes are needed to define the MDO problem, most of which are contained in the dictionary of `Discipline` objects.

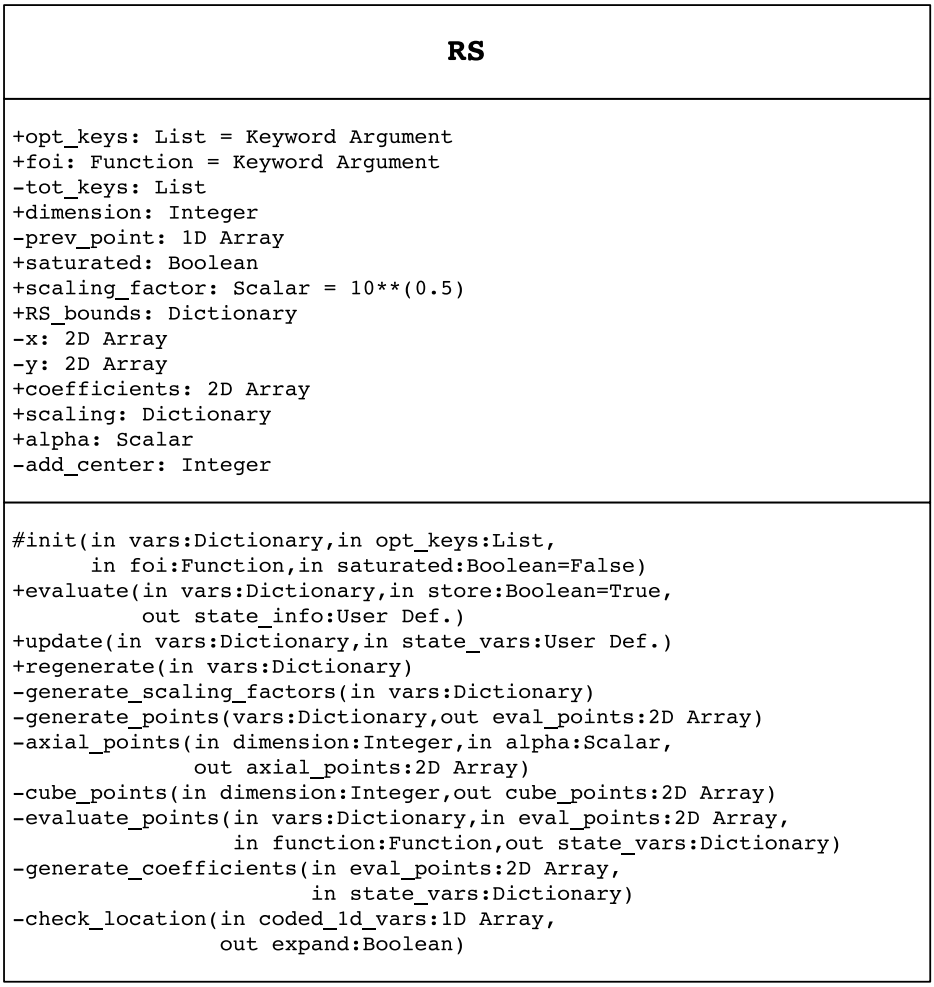

Fig. 4. UML diagram for the RS class.

In Figure 1, we can also see that MDOProb is the base class for the various MDO method classes. This is an abstract class and is therefore never instantiated. The optimize method, for example, is specific to each method and can only be defined once the class is specialized.

For the bilevel methods (CO and CSSO), a minimum of three instances of the OptProb class are necessary. They are instantiated in the top level optimize method and the exact number of instances depends on the particular method and the number of disciplines in a given problem.

4.4.2 *The* MDF *Class.* The MDF class inherits all attributes and methods from both OptProb and MDOProb. As with all specific method classes, several methods are overloaded. However, this method is the one that most closely resembles a single discipline problem and thus little specialization is necessary.

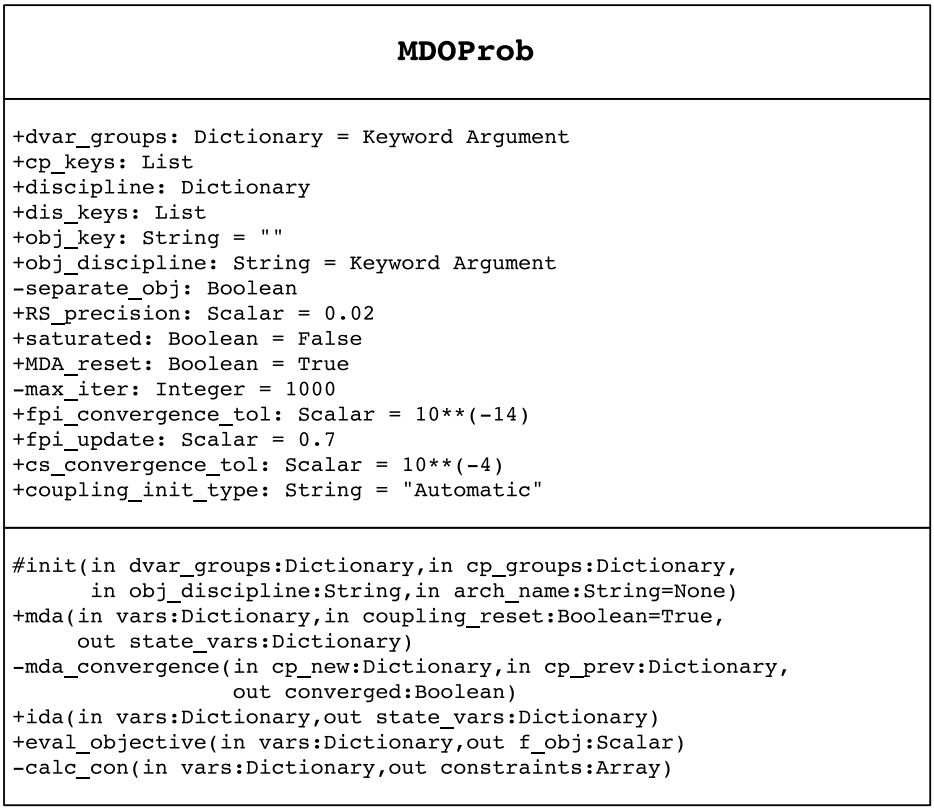

Fig. 5. UML diagram for the MDOProb class.

To evaluate the objective function (eval_objective in Figure 6), a multidisciplinary simulation must be performed. The default algorithm used to converge the multidisciplinary system is a block Gauss–Seidel iteration. The same is true for the constraints, which are assembled from every discipline and enforced simultaneously in the single optimization problem.

4.4.3 *The* IDF *Class.* As with the MDF class, the complete set of constraints is assembled by gathering the constraints from each discipline. However, additional constraints involving the coupling variables are added according to the IDF formulation (5). The set of design variables is also augmented to include the coupling variables.

The objective and constraint functions can be evaluated without converging the multidisciplinary simulation, that is, only uncoupled simulations are required. The attributes and methods shown in Figure 7 are identical to the ones shown for the MDF class. The differences reside in the underlying definition of methods.

4.4.4 *The* SAND *Class.* In this class (Figure 8), the optimizer enforces the simulation-based constraints as part of the optimization problem. As described

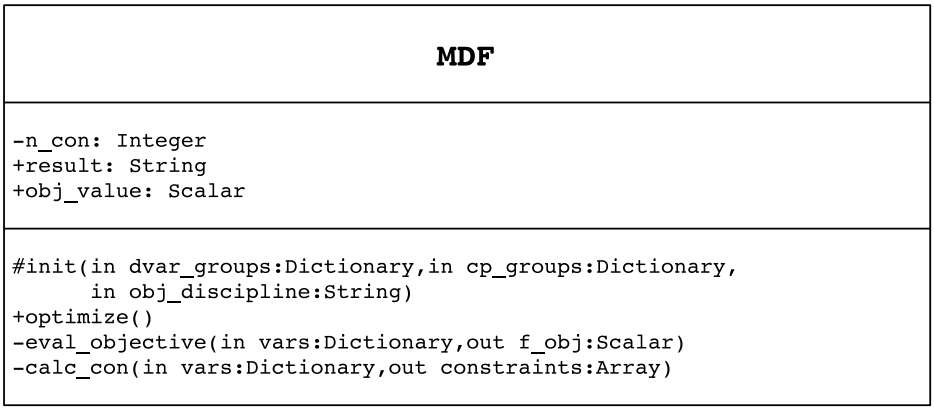

Fig. 6. UML diagram for the MDF class.

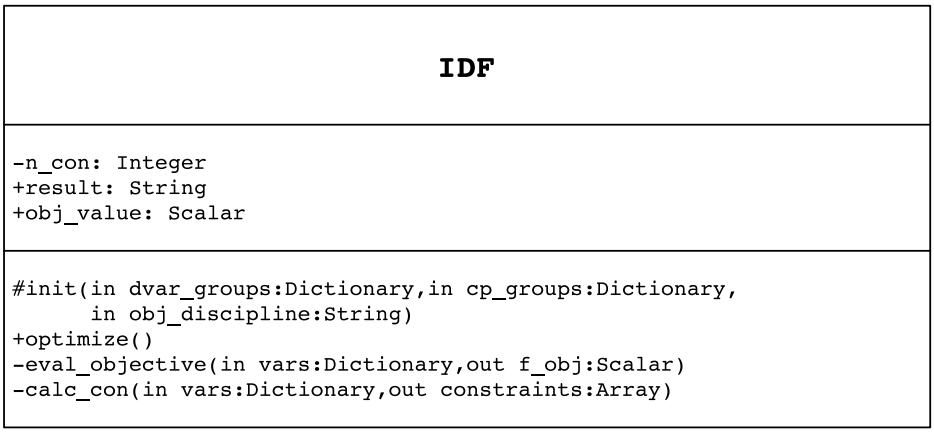

Fig. 7. UML diagram for the IDF class.

previously, this is done by enforcing equality constraints corresponding to the residuals of the governing Equations (1). Thus an additional method for residual evaluation of each simulation must be provided for the SAND class.

Similarly to IDF, SAND augments both the set of design variables and constraints relative to the MDF method.

4.4.5 *The* CO *Class.* This class (Figure 9) requires more extensive redefinition of methods and addition of new ones relative to the MDOProb base class. Since CO is a bilevel method, the optimize method includes calls to the system level optimization as well as subspace optimizations corresponding to each of the disciplines.

An OptProb instance is created for each discipline instance in the MDO problem. The design variables are then distributed among the different optimization problems according to their status as global or local variables. In the case of local variables, they are the design variables of the optimization problem

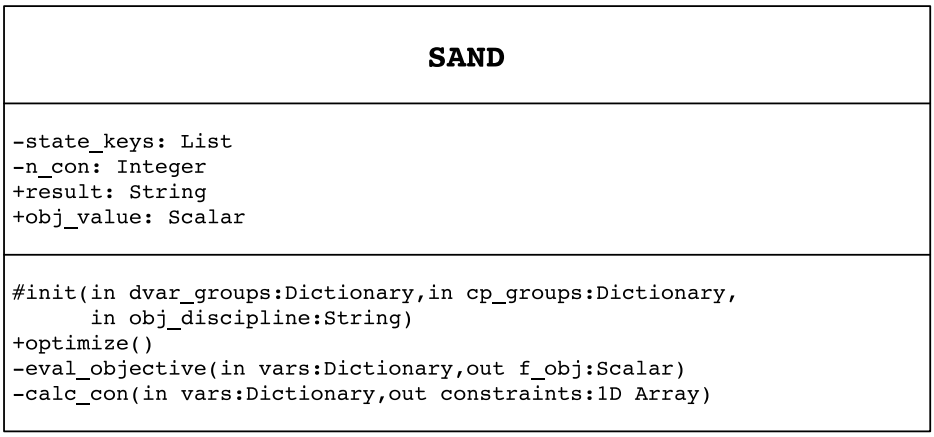

Fig. 8. UML diagram for SAND class.

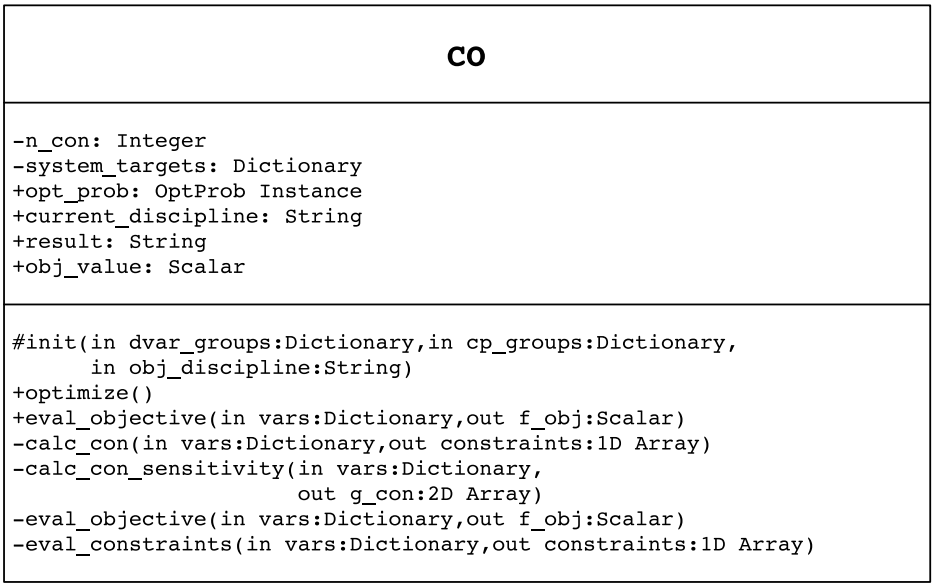

Fig. 9. UML diagram for the CO class.

corresponding to their respective disciplines. For each optimization problem, individual eval_objective, eval_constraints, and analysis methods are defined corresponding to the discipline sub problem. As with IDF, coupling variables are required to be design variables at the system level.

All local constraints of the original optimization problem are enforced at the discipline level. Global constraints are handled by the system level optimizer, which also handles the compatibility constraints specific to CO. The sensitivity of these constraints is computed using post-optimality sensitivity analysis of the optimum of each discipline-level optimization after it has converged.

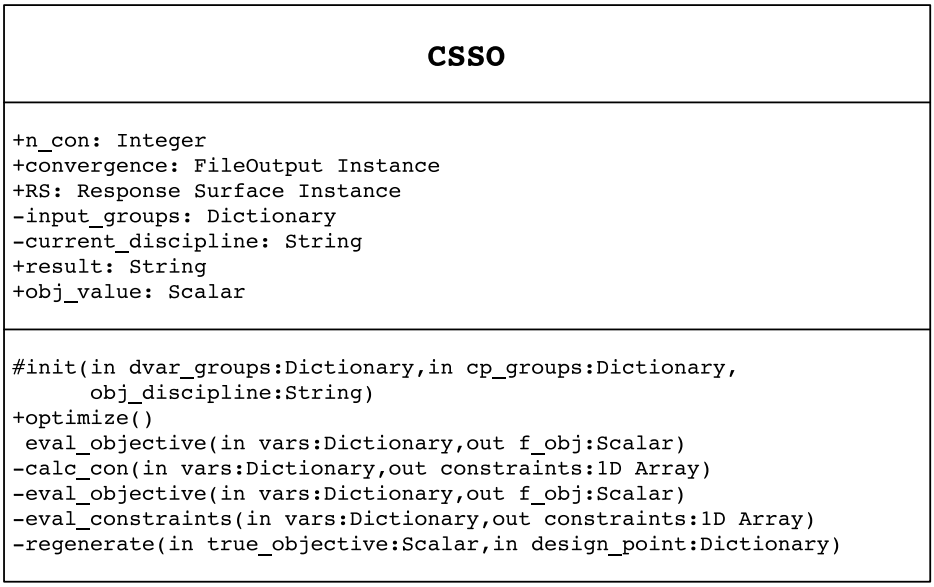

Fig. 10. UML diagram for the CSSO class.

Before any optimization is performed, a multidisciplinary simulation is run to obtain a suitable starting point that is multidisciplinary feasible. This was found to greatly increase the efficiency and robustness of the CO method.

Finally, response surfaces are sometimes used to represent the optimal points of each discipline as a function of the variables of the system-level problem (the global and coupling variables).

4.4.6 *The* CSSO *Class.* The main differentiating feature of the CSSO method (Figure 10) is the extensive use of response surface approximations, in the form of the RS class previously described.

The objective, the constraints, and the set of design variables at the system-level are identical to the ones in the original MDO problem. What is different is the way the coupling variables are computed: these are approximated as functions of the design variables using response surfaces that are modelled on the analysis method of each discipline instance.

As in the case of CO, an instance of OptProb is required for each discipline. Due to the slightly different structure of the CSSO class, an instance of the FileOutput class is contained within CSSO with modified attributes to accurately represent the system level convergence.

5. RESULTS

For the purpose of demonstration, we will now present results for two different problems. Both of the problems are analytic problems that would usually be formulated as single-discipline problems. The disciplines consist in sets of explicit coupled equations that yield the states.

5.1 Example Problem

The following optimization problem was originally formulated by Sellar et al. [1996] and was selected to demonstrate the framework. The problem is defined as,

$$\begin{aligned} \text{minimize} \quad & x_1^2 + z_2 + y_1 + e^{-y_2} \\ \text{with respect to} \quad & z_1, z_2, x_1 \\ \text{subject to} \quad & y_1/3.16 - 1 \geq 0 \\ & 1 - y_2/24 \geq 0 \\ & -10 \leq z_1 \leq 10 \\ & 0 \leq z_2 \leq 10 \\ & 0 \leq x_1 \leq 10, \end{aligned} \tag{13}$$

where y_1 and y_2 are the states of the two disciplines given by

$$y_1(z_1, z_2, x_1, y_2) = z_1^2 + x_1 + z_2 - 0.2y_2, \tag{14}$$

$$y_2(z_1, z_2, y_1) = \sqrt{y_1} + z_1 + z_2. \tag{15}$$

Although this is a simple problem, it exhibits characteristics of larger MDO problems. There are two disciplines, each consisting of one state variable that also functions as a coupling variable (y_1, y_2). There are two global design variables (z_1, z_2) and an additional design variable local to Discipline 1 (x_1). The coupling between the two disciplines is nonlinear as is the coupling between the disciplines and the system level optimizer. Each discipline has a local constraint associated with its state.

With the divisors of the constraint functions formulated as 3.16 and 24, the global optimum of this problem is $(z_1, z_2, x_1) = (1.9776, 0, 0)$, where the objective value is 3.18339 and Discipline 1's constraint is active.

The Python source code corresponding to the implementation of this optimization problem is shown in Figure 11. This problem was solved successfully using all five methods from a number of starting points. Underrelaxation was not required to converge the coupled Equations (14) and (15). Response surface limits were initialized to the upper and lower bounds of the design variables. The convergence tolerance for the MDA module was set to 10^{-15}. All of SNOPT's parameters were left at their default values; therefore the optimality conditions were satisfied to a tolerance of 10^{-6}.

Table I presents the computational performance of each of the methods starting from $(z_1, z_2, x_1) = (1, 2, 5)$ using both finite differences and the complex-step method [Squire and Trapp 1998; Martins et al. 2003] to compute the necessary sensitivities. A step of 10^{-8} was used for the finite differences, and the value for the complex step was 10^{-20}.

Since an analytic solution can be obtained for this problem, the optimum returned by the optimizer was compared to the exact one. The comparison was made using an l^2-norm of the difference between the optimal design variables and their corresponding exact values, that is,

$$\varepsilon_x = \|x - x_{\text{exact}}\|_2. \tag{16}$$

```
# Import universal framework module
import MDOProb

# Problem setup
dv_groups = {'z1':1, 'z2':1, 'x1':1}
discipline_groups = {'Discipline 1':  {'y1':1}, \
                     'Discipline 2':  {'y2':1}  }
objective_discipline = 'Objective'

# Method selection -- MDF, IDF, SAND, CO, CSSO available.
prob = MDOProb.CO(dv_groups, discipline_groups, objective_discipline)

# Set options
prob.name = "Analytic"
prob.sens_type = "CS"
prob.fpi_update = 1.0
prob.fpi_convergence_tol = 1.e-15
prob.cs_convergence_tol = 1.e-4
prob.coupling_init_type = "Automatic"
prob.RS_precision = 0.01

# Define objective function
def eval_objective(vars):
    return array([vars['x1'][0]**2 + vars['z2'][0] + vars['y1'][0] + \
                  exp(-vars['y2'][0])])

# Define objective dependencies
prob.discipline['Objective'].inputs.extend(['x1', 'z2', 'y1', 'y2'])
prob.discipline['Objective'].analysis =  eval_objective

# Discipline 1 function definitions
def dis1_analysis(vars):
    return array([vars['z1'][0]**2 + vars['x1'][0] + vars['z2'][0] - \
                  0.2*vars['y2'][0]])
def dis1_get_coupling(vars, state_vars):
    vars['y1'][:] = state_vars[:]
def dis1_constraints(vars, state_vars):
    return array([vars['y1'][0] / 3.16 - 1.0])
def dis1_residuals(vars, state_vars):
    return array([vars['z1'][0]**2 + vars['x1'][0] + vars['z2'][0] - \
                  0.2*vars['y2'][0] - state_vars[0]])

# Discipline setup - Only discipline 1 is shown
prob.discipline['Discipline 1'].inputs.extend(['z1', 'x1', 'z2', 'y2'])
prob.discipline['Discipline 1'].local_vars.extend(['x1'])
prob.discipline['Discipline 1'].constraint_type = ['>']
prob.discipline['Discipline 1'].analysis = dis1_analysis
prob.discipline['Discipline 1'].get_coupling = dis1_get_coupling
prob.discipline['Discipline 1'].constraints = dis1_constraints
prob.discipline['Discipline 1'].eval_residual = dis1_residuals

# Setup design variables - Only variable z1 shown
prob.vars['z1'].value[0] = 1
prob.vars['z1'].lower[0] = -10
prob.vars['z1'].upper[0] = 10
prob.vars['z1'].influence.append('Discipline 1')
prob.vars['z1'].RSLowerLimit[0] = -10
prob.vars['z1'].RSUpperLimit[0] = 10

# Solve the problem and retrieve results
prob.optimize()
print prob.result
print prob.obj_value
```

Fig. 11. Python source code that implements the sample MDO problem (13).

Table II shows the error for the various methods, which is within the specified tolerance for all cases.

Table I. Number of Function Calls for the Example Problem; * Denotes Residual Evaluations

	Finite difference		Complex step	
Method	Discipline 1	Discipline 2	Discipline 1	Discipline 2
MDF	346	346	238	238
IDF	61	61	55	55
SAND	1+57*	1+57*	1+49*	1+49*
CO	1,291	729	1,079	587
CSSO	1,250	1,188	1,210	1,148

Table II. l^2-Norm of Absolute Error in Optimal Design Variables

Method	Finite difference	Complex step
MDF	1.1515×10^{-6}	1.1506×10^{-6}
IDF	2.0827×10^{-9}	2.0803×10^{-9}
SAND	7.1353×10^{-7}	7.1356×10^{-7}
CO	6.1643×10^{-6}	7.3389×10^{-6}
CSSO	7.1386×10^{-6}	3.9559×10^{-6}

To compare convergence histories, we used the relative error of the objective function value, that is,

$$\varepsilon_f = \left| \frac{f - f_{\text{exact}}}{f_{\text{exact}}} \right|. \tag{17}$$

The convergence histories for the various MDO methods are shown in Figure 12. For the MDF, IDF, SAND, and CO methods, times are reported each time the optimizer (system-level optimizer in the case of CO) calls the objective function. For CSSO, the time is reported at each system-level iteration and before and after each response surface generation. Flat sections in the convergence history of CSSO represent the time taken to generate the response surfaces.

As can be seen in Table I, the monolithic methods are more efficient than the bilevel methods for this problem. IDF is slightly faster than SAND due to the fact that SAND requires a relatively large computational overhead. Since the structure of the state variables is left to the discretion of the user, SAND must restructure them into a common format each time a user provided function is evaluated. This results in an additional computational effort that is not directly related to solving the optimization problem.

Though the use of complex-step sensitivity analysis resulted in fewer function evaluations in all cases, it did not result in substantially faster execution or a more accurate final variable set. This is because the convergence tolerance of the optimizer during each run was held constant. As function evaluations for this particular problem are inexpensive, the increase in the number of function calls by each of the methods from the complex-step method to the finite-difference method does not impact the convergence time. As function evaluations become more expensive in relation to the computational overhead of each method, the advantages of the complex-step method in reducing the number of function evaluations should become more apparent.

We should emphasize that since the performance of MDO methods strongly depends on the type of the problem being solved, the observations noted above cannot be generalized.

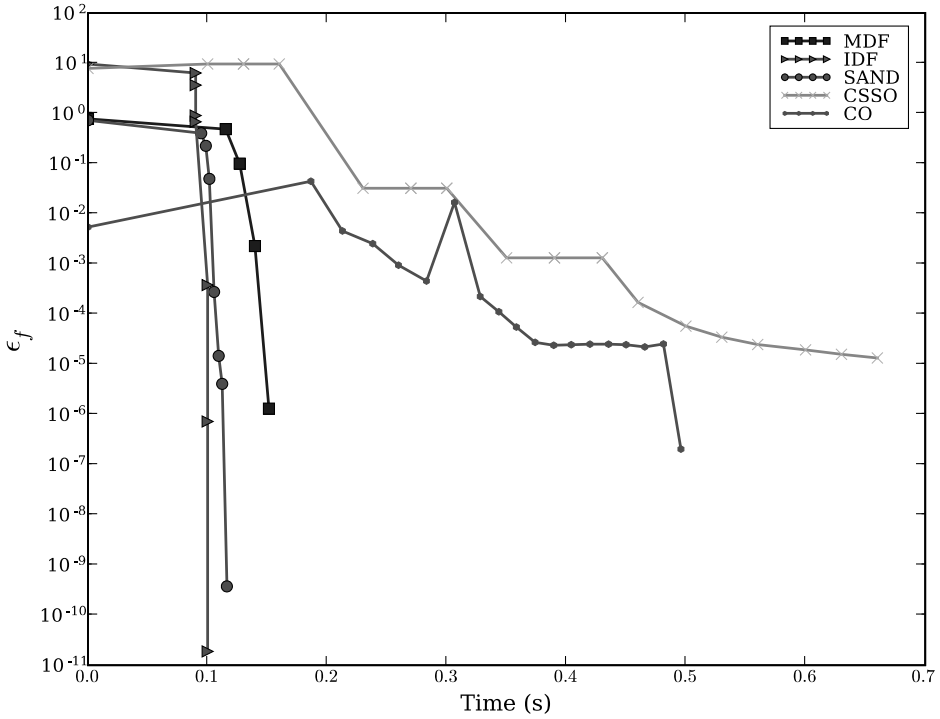

Fig. 12. Example problem convergence history.

Figure 13 is a directed graph showing the variable dependence for this problem once the MDF approach is applied. This diagram is automatically generated by pyMDO. The method that produces this figure extracts variable and discipline information from the problem data after the problem has been decomposed. Using the dictionaries of the discipline inputs and outputs as well as the optimizer design variables, the function maps the variable flow for a given method and creates an output file in the GraphViz format [Gansner and North 2000]. Using GraphViz, the structure of the decomposed problem can then be plotted for a given MDO approach, allowing a deeper understanding of the structure of both the problem and the approach. Furthermore, an adjacency matrix can be derived from these graphs and used to modify or simplify the structure of the multidisciplinary problem using graph theory.

5.2 Problem with Large Number of Design Variables

A larger and more complex example is also presented. This problem is a version of a scalable problem that was developed to automatically generate MDO problems with arbitrary dimensionality with respect to number of disciplines, number of global and local design variables, and number of coupling variables [Tedford and Martins 2006a]. The problem is as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{5} z_i^2 + \sum_{i=1}^{5} y_i^2 \\ \text{with respect to} \quad & z, x_1, x_2, x_3, x_4, x_5, \end{aligned} \tag{18}$$

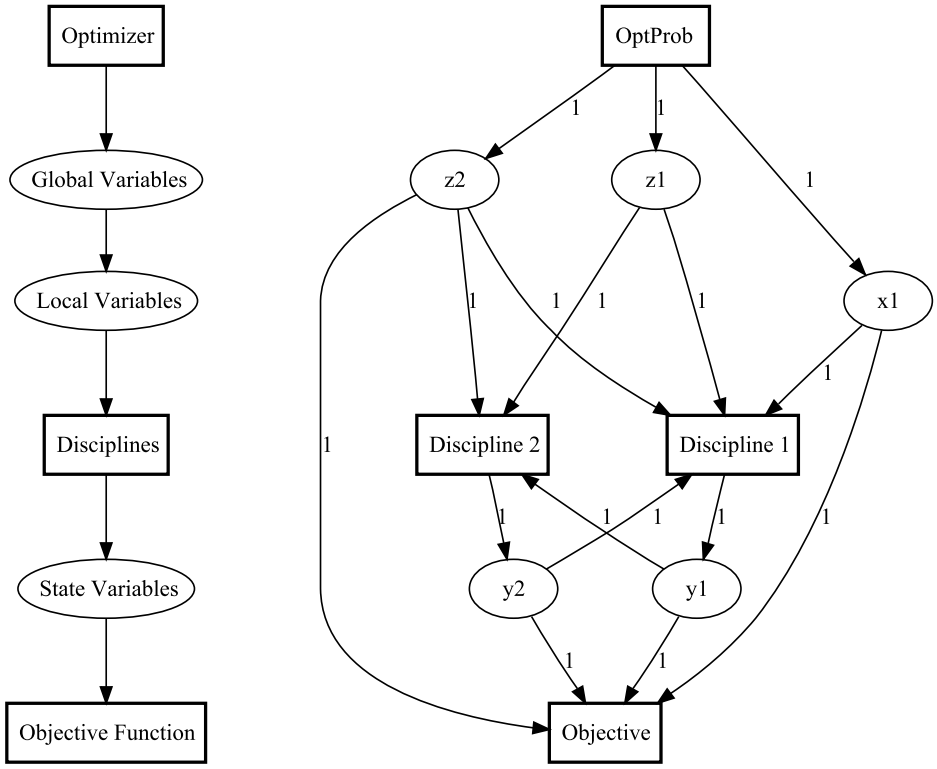

Fig. 13. Graph of variable dependencies for the example problem using the MDF approach.

where y_i represents the output coupling variables of each discipline and is determined by

$$
\begin{aligned}
y_1(z, x_1, y_3, y_5) &= -\frac{1}{C_{y_1}}(C_{z1}z + C_{x1}x_1 - C_{y_3}y_3 - C_{y_5}y_5),\\
y_2(z, x_2, y_1, y_3) &= -\frac{1}{C_{y_2}}(C_{z2}z + C_{x2}x_2 - C_{y_1}y_1 - C_{y_3}y_3),\\
y_3(z, x_3, y_2, y_5) &= -\frac{1}{C_{y_3}}(C_{z3}z + C_{x3}x_3 - C_{y_2}y_2 - C_{y_5}y_5),\\
y_4(z, x_4, y_1, y_5) &= -\frac{1}{C_{y_4}}(C_{z4}z + C_{x4}x_4 - C_{y_1}y_1 - C_{y_5}y_5),\\
y_5(z, x_5, y_2, y_4) &= -\frac{1}{C_{y_5}}(C_{z5}z + C_{x5}x_5 - C_{y_2}y_2 - C_{y_4}y_4),\\
\text{such that} \quad & 1 - \frac{y_i}{C_i} \leq 0 \qquad i = 1, \ldots, 5.
\end{aligned}
\tag{19}
$$

The variable dependencies for this problem are shown in Figure 14. The problem consists of 5 disciplines ($i = 1, \ldots, 5$) and five global design

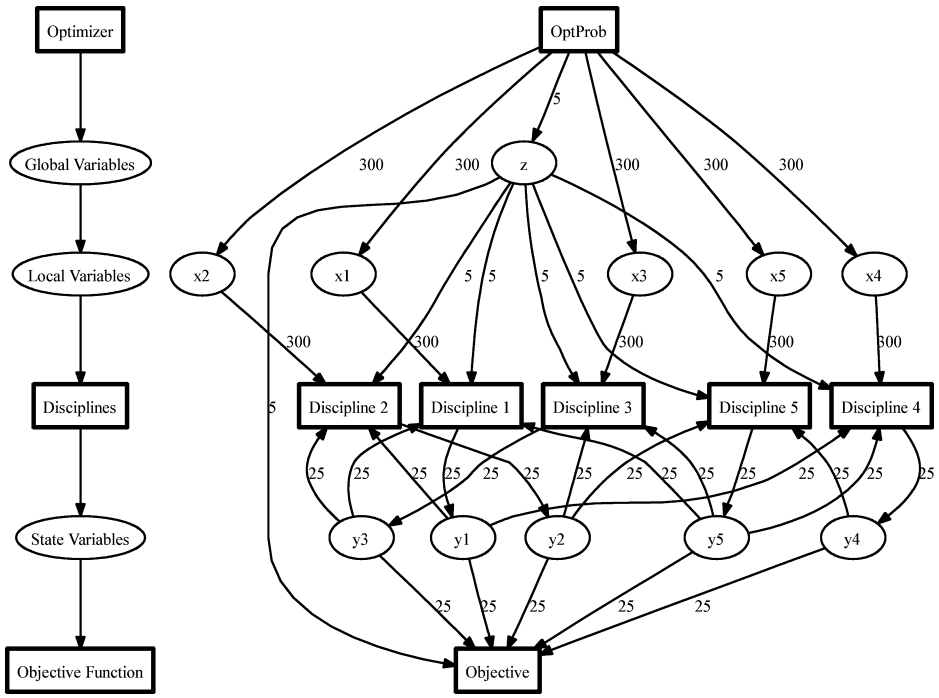

Fig. 14. MDF variable dependency for high-dimensionality problem.

variables, z_i. For each discipline there are 300 local variables, x_i, and 25 coupling variables, y_i. The problem thus consists of 1505 design variables and 125 coupling variables.

In the problem statement ((18) and (19)), all Cs represent a matrix of randomly generated coefficients generated on the first problem initialization, and reused for subsequent optimizations. The various x and y values, x_1, x_2, for example, represent their respective vectors of design or coupling variables. Due to the relatively high dimensionality of this problem, the quadratic response surface implemented in CSSO requires an excessive number of function evaluations making this method orders of magnitude slower than the others. The convergence history of the remaining methods (MDF, IDF, SAND, and CO) is shown in Figure 15. The performance of each of the architectures in terms of function evaluations is shown in Table III.

Similarly to the first test problem, the monolithic methods perform better than the bilevel architectures in this problem. In this case, CO exhibits a much higher convergence rate than the MDF formulation. In related trials, it was found that the performance of CO was heavily dependent on the ratio of local design variables to coupling variables at the discipline level. In this case, the number of local variables is greater than the number of output coupling variables, allowing CO to exploit this freedom in the design space and outperform MDF.

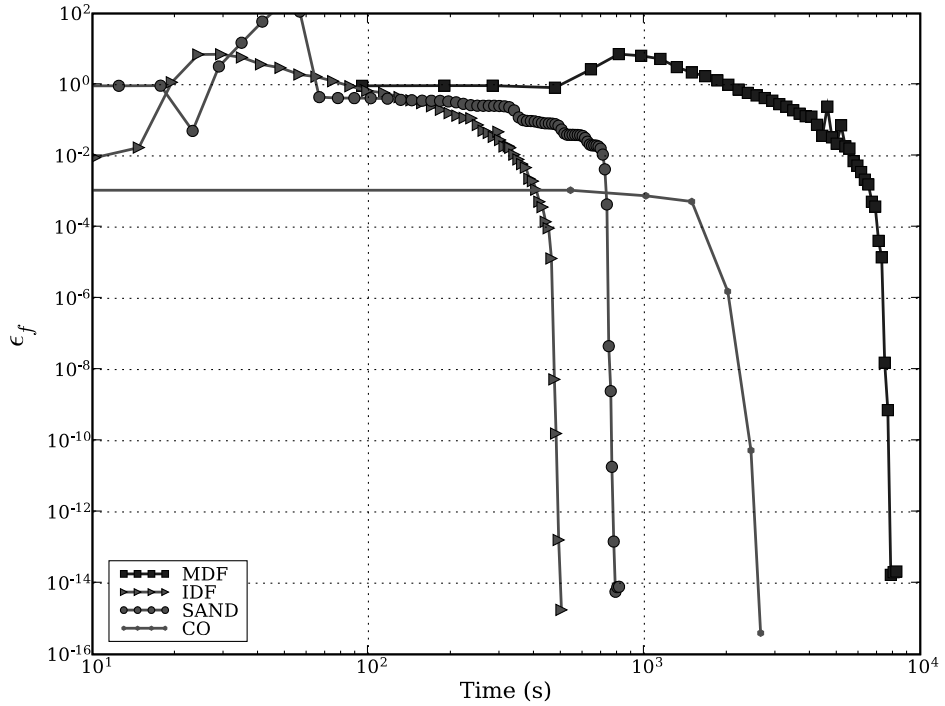

Fig. 15. Convergence plot for high-dimensionality problem.

Table III. Number of Function Calls for the High-Dimensionality Problem; * Denotes Residual Evaluations

Method	Discipline 1	Discipline 2	Discipline 3	Discipline 4	Discipline 5
MDF	3,934,410	3,934,410	3,934,410	3,934,410	3,934,410
IDF	84,816	84,816	84,816	84,816	84,816
SAND	1+114,171*	1+114,171*	1+114,171*	1+114,171*	1+114,171*
CO	765,404	712,716	797,444	746,892	740,484

6. SUMMARY

In this article, pyMDO was shown to be a portable and freely accessible tool for the rapid implementation of MDO methods. The implementation details are automatically managed by the framework: once a problem is described in pyMDO, it can be solved using a variety of methods, optimizers, and response surfaces with reduced effort.

Using carefully designed classes, inheritance, and operator overloading, the pyMDO framework vastly simplifies the structure of MDO software. In the process, it enhances the clarity, reuse, and portability of the resulting program. Additionally, the user interface greatly resembles the algorithms it implements.

pyMDO makes use of a number of classes with a logical hierarchical structure and clear syntax. The use of inheritance and consistent code reuse avoids the need for long lists of parameters and makes the code more intuitive to users

who lack an extensive programming background. In a true expression of the modular software model, all components that are unique to a given algorithm are encapsulated in either functions or classes.

Given the above attributes and the fact that pyMDO is freely available, we think that it is a useful platform for benchmarking current MDO methods and developing new ones.

REFERENCES

ALEXANDROV, N. M. AND KODIYALAM, S. 1998. Initial results of an MDO evaluation survey. AIAA Paper, AIAA, Reston, VA, 98–4884.

ALEXANDROV, N. M. AND LEWIS, R. M. 2002. Analytical and computational aspects of collaborative optimization for multidisciplinary design. *AIAA J. 40*, 2, 301–309.

ALEXANDROV, N. M. AND LEWIS, R. M. 2004a. Reconfigurability in MDO problem synthesis, part 1. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. AIAA, Reston, VA, 2004–4307.

ALEXANDROV, N. M. AND LEWIS, R. M. 2004b. Reconfigurability in MDO problem synthesis, part 2. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. (Albany, NY). AIAA, Reston, VA, 2004–4308.

ALONSO, J. J., LEGRESLEY, P., VAN DER WEIDE, E., MARTINS, J. R. R. A., AND REUTHER, J. J. 2004. pyMDO: A framework for high-fidelity multi-disciplinary optimization. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. AIAA, Reston, VA, 2004–4480.

BEAZLEY, D. M. 2006. *Python Essential Reference*, 3rd ed. Sams, Indianapolis, IN.

BLEZEK, D. 1998. Rapid prototyping with SWIG. *C/C++ Users J. 16*, 11, 61–66.

BRAUN, R. D. AND KROO, I. M. 1997. Development and application of the collaborative optimization architecture in a multidisciplinary design environment. In *Multidisciplinary Design Optimization: State of the Art*, N. Alexandrov and M. Y. Hussaini, Eds. SIAM, Philadelphia, PA. 98–116.

BROWN, N. F. AND OLDS, J. R. 2006. Evaluation of multidisciplinary optimization techniques applied to a reusable launch vehicle. *J. Space. Rock. 43*, 6, 1289–1300.

CRAMER, E. J., DENNIS, J. E., FRANK, P. D., LEWIS, R. M., AND SHUBIN, G. R. 1994. Problem formulation for multidisciplinary optimization. *SIAM J. Opt. 4*, 4, 754–776.

DEMIGUEL, V. AND MURRAY, W. 2006. A local convergence analysis of bilevel decomposition algorithms. *Opt. Eng. 7*, 2, 99–133.

ELDRED, M. S., BROWN, S. L., ADAMS, B. M., DUNLAVY, D. M., GAY, D. M., SWILER, L. P., GIUNTA, A. A., HART, W. E., WATSON, J.-P., EDDY, J. P., GRIFFIN, J. D., HOUGH, P. D., KOLDA, T. G., MARTINEZ-CANALES, M. L., AND WILLIAMS, P. J. 2006. *DAKOTA: A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis, Version 4.0 User's Manual*. Sandia National Laboratories, Albuquerque, NM.

ELDRED, M. S., OUTKA, D. E., BOHNHOFF, W. J., WITKOWSKI, W. R., ROMERO, V. J., PONSLET, E. R., AND CHEN, K. S. 1996. Optimization of complex mechanics simulations with object-oriented software design. *Comput. Model. Sim. Eng. 1*, 3, 323–352.

GANSNER, E. R. AND NORTH, S. C. 2000. An open graph visualization system and its applications to software engineering. *Softw.—Pract. Exper. 30*, 11, 1203–1233.

GILL, P. E., MURRAY, W., AND SAUNDERS, M. A. 2002. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM J. Opt. 12*, 4, 979–1006.

HAFTKA, R. T. AND GÜRDAL, Z. 1993. *Elements of Structural Optimization*, 3rd ed. Kluwer, Dordrecht, The Netherlands, Chapter 10.

KODIYALAM, S. 1998. Evaluation of methods for multidisciplinary design optimization (MDO), part 1. NASA Report CR-2000-210313. NASA, Washington, DC.

KODIYALAM, S. AND YUAN, C. 2000. Evaluation of methods for multidisciplinary design optimization (MDO), part 2. *NASA Report* CR-2000-210313. Nov. Washington, DC.

KROO, I. M. 1997. MDO for large-scale design. In *Multidisciplinary Design Optimization: State-of-the-Art*, N. Alexandrov and M. Y. Hussaini, Eds. SIAM, Philadelphia, PA. 22–44.

LANGTANGEN, H. P. 2004. *Python Scripting for Computational Science*. Springer, Berlin, Germany.

MARTINS, J. R. R. A., STURDZA, P., AND ALONSO, J. J. 2003. The complex-step derivative approximation. *ACM Trans. Math. Softw. 29*, 3, 245–262.

MEZA, J. C., OLIVA, R. A., HOUGH, P. D., AND WILLIAMS, P. J. 2007. OPT++: An object-oriented toolkit for nonlinear optimization. *ACM Trans. Math. Softw. 33*, 2, 12.

O'DOCHERTY, M. 2005. *Object-Oriented Analysis and Design*. John Wiley and Sons, New York, NY.

PADULA, S. L., ALEXANDROV, N., AND GREEN, L. L. 1996. MDO test suite at NASA Langley Research Center. In *Proceedings of the 6th AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*. AIAA, Reston, VA, 1996-4028.

PEREZ, R. E., LIU, H. H. T., AND BEHDINAN, K. 2004. Evaluation of multidisciplinary optimization approaches for aircraft conceptual design. In *Proceedings of the 10th AIAA/ISSMO Muldisiciplinary Analysis and Optimization Conference* (Albany, NY). AIAA, Reston, VA, 2004-4537.

PETERSON, P., MARTINS, J. R. R. A., AND ALONSO, J. J. 2001. Fortran to Python interface generator with an application to aerospace engineering. In *Proceedings of the 9th International Python Conference* (Long Beach, CA).

SCHMIT, JR., L. A. AND RAMANATHAN, R. K. 1978. Multilevel approach to minimum weight design including buckling constraints. *AIAA J. 16*, 2, 97–104.

SELLAR, R. S., BATILL, S. M., AND RENAUD, J. E. 1996. Response surface based, concurrent subspace optimization for multidisciplinary system design. In *Proceedings of the 34th AIAA Aerospace Sciences Meeting and Exhibit* (Reno, NV). AIAA, Reston, VA, 1996-0714.

SOBIESKI, I. P. AND KROO, I. M. 2000. Collaborative optimization using response surface estimation. *AIAA J. 38*, 10, 1931–1938.

SOBIESZCZANSKI-SOBIESKI, J. 1988. Optimization by decomposition: A step from hierarchic to non-hierarchic systems. *NASA tech. rep.* CP-3031. NASA, Washington, DC.

SOBIESZCZANSKI-SOBIESKI, J. AND HAFTKA, R. T. 1997. Multidisciplinary aerospace design optimization: Survey of recent developments. *Struct. Opt. 14*, 1, 1–23.

SQUIRE, W. AND TRAPP, G. 1998. Using complex variables to estimate derivatives of real functions. *SIAM Rev. 40*, 1, 110–112.

TEDFORD, N. P. AND MARTINS, J. R. R. A. 2006a. Comparison of MDO architectures within a universal framework. In *Proceedings of the 2nd AIAA Multidisciplinary Design Optimization Specialist Conference* (Newport, RI). AIAA, Reston, VA, 2006–1617.

TEDFORD, N. P. AND MARTINS, J. R. R. A. 2006b. On the common structure of MDO problems: A comparison of architectures. In *Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference* (Portsmouth, VA). AIAA, Reston, VA, 2006–7080.

THAREJA, R. AND HAFTKA, R. T. 1986. Numerical difficulties associated with using equality constraints toachieve multi-level decomposition in structural optimization. In *Proceedings of the 27th Structures, Structural Dynamics and Materials Conference* (San Antonio, TX). AIAA, Reston, VA. 21–28.

WUJEK, B., RENAUD, J., AND BATILL, S. 1997. A concurrent engineering approach for multidisciplinary design in a distributed computing environment. In *Multidisciplinary Design Optimization: State of the Art*, N. Alexandrov and M. Y. Hussaini, Eds. SIAM, Philadelphia, PA. 189–208.

Received September 2007; revised December 2008; accepted February 2009