12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference
10 - 12 September 2008, Victoria, British Columbia Canada

AIAA 2008-6069

The Development of an Open Source Framework for Multidisciplinary Analysis & Optimization

Kenneth T. Moore[1] and Bret A. Naylor[2]
Wyle Information Systems, Cleveland, Ohio

and

Justin S. Gray[3]
NASA Glenn Research Center, Cleveland, Ohio

This paper presents the motivation for the development of an open source framework for performing Multidisciplinary Analysis and Optimization (MDAO) to aid in the design of unconventional aircraft. While a number of frameworks already exist, critical and desirable requirements are still not satisfactorily met. For developing a new framework, an open source development is the most logical choice, in particular because it provides the means to rapidly develop a community of users who can also contribute enhancements, components, and knowledge to the framework. Presently, the framework development project has finished the requirements gathering stage and has begun prototyping some of the capability in the cross-platform scripting language Python, which already provides many of the basic building blocks needed to create an MDAO framework.

Nomenclature

ASTROS	=	Automated Structural Optimization System
BLAS	=	Basic Linear Algebra Subprograms
CDS	=	Conceptual Design Shop
DAKOTA	=	Design Analysis Kit for Optimization and Terascale Applications
GPL	=	GNU General Public License
GUI	=	Graphical User Interface
LAPACK	=	Linear Algebra PACKage
LGPL	=	Lesser GNU Public License
MDAO	=	Multidisciplinary Analysis and Optimization
NASA	=	National Aeronautics and Space Administration
NOSA	=	NASA Open Source Agreement
NPSS	=	Numerical Propulsion System Simulation
ROSE	=	Repetitive, Object-oriented Simulation Environment
SWIG	=	Simplified Wrapper and Interface Generator

I. Introduction

IN order to survive the present and future economic and environmental challenges facing air transportation, aviation design must expand its focus beyond today's conventional "wing-body-tail" aircraft and investigate more advanced and efficient vehicle concepts. The ability to design revolutionary aircraft to improve mobility and air transport efficiency is of paramount importance and is expected to deliver additional benefits, such as the reduction

[1] Senior Systems Engineer, MDAO Branch, Mail Stop 500-105, NASA Glenn Research Center, AIAA Senior Member
[2] Senior Software Engineer, MDAO Branch, Mail Stop 500-105, NASA Glenn Research Center
[3] Aerospace Engineer, MDAO Branch, Mail Stop 5-11, NASA Glenn Research Center, AIAA Member

of fleet fuel costs, the minimization of the environmental impact, as well as the improvement of mission capability in support of national defense. However, traditional design methods rely on the accumulated experience of the designer and the availability of data, and are not necessarily well suited for unconventional concepts. Test data and design experience are inherently limited for unconventional aircraft, thus there is a strong need for the incorporation of more sophisticated computational models and methods, in particular, physics-based Multidisciplinary Analysis and Optimization (MDAO), into contemporary and future design practices.

To address this need, NASA's Fundamental Aeronautics Program has undertaken the development of Multidisciplinary Analysis and Optimization (MDAO) capabilities. The goal of these capabilities is to facilitate a seamless transition between single-discipline and multidisciplinary analyses by providing systematic processes and an intelligent computational environment for managing multidisciplinary variable-fidelity tools that enable system analysis and optimization at primarily the conceptual and preliminary design stages for all flight regimes of conventional and unconventional vehicle classes. The MDAO capability will be compatible with state-of-the-art computing architectures, comply with established standards, support distributed heterogeneous platforms, and reduce overall setup and runtime.[1]

Two distinct categories of MDAO tools will be developed as part of this work: the discipline-specific analysis tools and the framework that interconnects them. The analysis tools generally include a number of specific discipline software codes of varying levels of fidelity that are pertinent to the aircraft conceptual design (for both conventional and unconventional vehicles). These tools also include an assortment of algorithms suitable for solving a wide range of optimization problems appropriate to a specific application. One of the most important tools is the geometry engine, which allows the automatic generation and re-generation of geometries (and their associated discretized meshes) at varying levels of fidelity, based on a persistent parameterized realization of the design concept. The software framework provides the underlying support for all of the analysis tools and facilitates co-operation between the discipline codes. A good framework minimizes the computer science burden and allows the engineer to focus on the pure engineering aspects of the analysis.

Currently, there are a small number of commercial MDAO frameworks; some of these are under development, while others are in a more mature state. At first glance, it might appear that the best solution for achieving NASA's goals in the development of an MDAO capability for aircraft conceptual design would be to choose such an existing framework and work directly with a group or company to incorporate additional capability therein. However, there are some significant advantages to developing a new MDAO framework under the open source initiative. One of the most readily evident advantages of an open-source software project is the price; such a framework would be free for anyone to use. Of course, this does not mean that the analysis codes, some of which may be commercial, proprietary, or otherwise restricted (e.g., International Traffic in Arms Regulations (ITAR)), would also default to an open-source status. The software framework itself (along with a selection of optimizers and solvers from open literature) would be open source, and it would be up to the discretion of the owner of any specific analysis component to decide how to handle its distribution. Another advantage of the open-source route–one that is more important than the cost–is the potential for extensibility of the core capabilities. Researchers and engineers from other organizations, academia, and industry would be able to contribute to the continuing development of the framework as well as submit their new analysis techniques as components. In particular, it should be noted that even though the current discipline toolsets are being developed to address a need in the design of conventional and unconventional aircraft, nothing inherent in the framework will restrict them to that use. Thus, this framework will be useable for any application in the milieu of multidisciplinary analysis and optimization. One additional benefit from an open-source framework is the ease of maintainability. Once the development reaches a certain level of maturity, an open-source community should begin developing around the framework. This community, with some help from NASA, should be able to maintain the code long after the majority of the initial development team has moved on to other activities.

II. Frameworks

When developing any new MDAO capability, careful consideration must be given to the software infrastructure that will support the analysis. This software infrastructure is the support structure that allows an MDAO problem to be solved, and as such, it includes the tools necessary to control communication and manage the passing of data between components (i.e., the analysis tools). It also provides the capability to set up, manage, and execute a variety of different design problems (e.g., process models) ideally using both textual and graphical interfaces. To facilitate the process of adding a new capability, the software infrastructure should also provide a reasonably powerful scripting capability. The infrastructure should also take care of managing any overhead associated with distributed computing, including the efficient mapping of processing tasks onto the available resources. Beyond these basic capabilities, a number of auxiliary capabilities and tools are also desirable, including plotting and data visualization, design space exploration tools, sensitivity analysis, surrogate (approximation) generation, and a functional driver interface that allows a variety of optimization problems to be solved. All of these capabilities comprise what will be referred to here as *the MDAO framework*.

A sophisticated software framework containing all of the attributes described above may not be needed for every application. A group of researchers at Tohoku University performed a design study of the wing shape for a proposed high-performance regional jet where they varied nine design variables of a parameterized wing cross section using a genetic algorithm to optimize three performance parameters.[2] The integration between the structural code (NASTRAN) and the aerodynamics code (an in-house unstructured grid solver called TAS-Code) was handled by a set of Unix shell scripts and operated by a human-in-the-loop. This is actually quite reasonable and manageable for these types of problems, i.e., where a code executes a small number of times but has a lengthy run time, or where human inspection of the flow results at each iteration may be required. However, even in this study, a few other tools were used for data mining the results, including Viscovery SOMine, a software program for reducing the higher-dimension parameter/objective space onto a 2-D self organizing map.

Other researchers have gone a step beyond shell scripting and have developed their own frameworks as part of their MDAO efforts. J. Alonso and J. Martins developed an MDAO framework called pyMDO using the scripting language Python.[3] Using this framework, they were able to easily integrate a set of analysis codes that included three-dimensional flow analysis, structural analysis, and parametric geometry to investigate the aero-structural coupling of a supersonic business jet. The modular design of their framework facilitates the incorporation of new modules as well as the swapping of different solvers and optimizers for new ones. The pyMDO effort is still ongoing at the Multidisciplinary Optimization Laboratory at the University of Toronto, with some new results being presented concurrently with this paper.[4]

In 2006, S. Padula and E. Gillian compiled what may be the first and only survey of framework development efforts, and in doing so, put together a history of MDAO frameworks spanning 20 years.[5] One of the earliest frameworks was a structural optimization framework called ASTROS (Automated Structural Optimization System) developed by the Air Force. While the analysis was low fidelity by today's standards, ASTROS integrated a simple finite element code with aerodynamic loads, an aeroelastic stability tool, and a closed-loop controller to perform design space exploration as well as constrained optimization.[6] Another fairly early example was COMETBOARDS (Comparative Evaluation Test Bed of Optimization and Analysis Routines for the Design of Structures), which was developed at NASA Glenn Research Center.[7] COMETBOARDS is primarily a sandbox for investigating optimization algorithms where the analysis codes are rather loosely coupled.

More recently, a number of commercial MDAO frameworks have become available, the most well known of which are iSight, by Engineous Software (recently acquired by Dassault Systemes), and ModelCenter, by Phoenix Integration. The iSight framework, which was officially announced in 1998 at the 7th AIAA Multidisciplinary Analysis and Optimization conference, provides the user with a Graphical User Interface (GUI) for assembling a process flow diagram that describes the execution order for components and the driver loop.[8] This is particularly useful for nesting optimizers and solvers and for setting up conditional logic for branching. The ModelCenter framework has also been commercially available for several years, and it includes a GUI for defining a problem by specifying the overall data flow between the modules. Apart from this fundamental difference in the two products, both of them include most of the capabilities listed at the beginning of this section.

Another particularly relevant framework is DAKOTA (Design Analysis Kit for Optimization and Terascale Applications), which was developed at Sandia National Laboratories for the simulation of complex systems pertinent to nuclear energy.[9] This framework covers disciplines such as structural dynamics, fluid mechanics, heat transfer, and shock physics. Because the researchers at Sandia often found themselves rebuilding the same kinds of interfaces between codes, in 1994 they decided to build a software framework to automate some of these tasks. DAKOTA has most of the features found in many of the above mentioned frameworks, including a variety of optimization methods, data visualization, sensitivity analysis, and solid support for parallel computing. What sets it apart from some of the other efforts is that it has been released as an open source project under the GNU General Public License (GPL). This means that not only is DAKOTA free to use (within the terms of the GPL), but its code base also benefits from the contributions of a community of developers. This concept of open source development will be discussed in greater detail in Section III–Open Source Software.

More recently another open execution framework option, ROSE (Repetitive, Object-oriented Simulation Environment)[10] has been under development at NASA Glenn Research Center. ROSE provides a flexible and extensible software specification for implementing model execution tools (e.g., optimizers, solvers, surrogate generators). Since it is primarily a software specification, ROSE can be implemented in any object-oriented language. ROSE can also provide support for parallel computing. In support of the current MDAO framework development effort, a ROSE implementation has been created in Python, named PyROSE. A set of basic optimizers and a Design-of-Experiments execution tool have been created for PyROSE; these have been tested on turbine engine models built using the Numerical Propulsion System Simulation software. Current development is focused on implementing parallel computing features in PyROSE using the ROSE specification.

The Numerical Propulsion System Simulation (NPSS)[11,12,13], mentioned above, is another object-oriented modeling framework developed, in part, at NASA Glenn Research Center. NPSS is an extensible framework for performing analysis of complex systems throughout the entire development life-cycle. The initial development of NPSS focused on airbreathing aircraft engines, but the resulting NPSS framework may be applied to any system, for example: aerospace, rockets, hypersonics, power and propulsion, fuel cells, ground based power, and even human system modeling. Through a Space Act Agreement and the NASA Industry Cooperative Effort (NICE), three levels of teaming combined with a formal software development process enabled involvement of NASA, industry, and DOD from planning stages through development and implementation. One of the key features of NPSS is its ability to make use of both compiled and scripted analysis tools. This capability enables NPSS to have both the computational efficiency of a compiled code and the flexibility of a scripting language.

A. Defining the Requirements for an MDAO Framework

Given the number of commercial and research grade MDAO frameworks that are available, a process needed to be defined for evaluating the alternatives and selecting the best one. NASA's current MDAO effort is the successor to earlier NASA projects such as the Conceptual Design Shop (CDS) and the Numerical Propulsion System Simulation (NPSS). CDS was a project similar to the present work, and was carried out by NASA in 2004 and 2005.[14] Originally intended to be a 6-year project, CDS ended after the second year, leaving the overall goals unrealized. However, while the project was active, a good body of useful work was produced. The requirements for the CDS project were defined based on the input of a number of discipline experts; this resulted in a set of a few hundred requirements covering specific discipline tools as well as the MDAO framework. A set of fairly high-level framework requirements has been condensed from the full set and is shown in Figure 1.

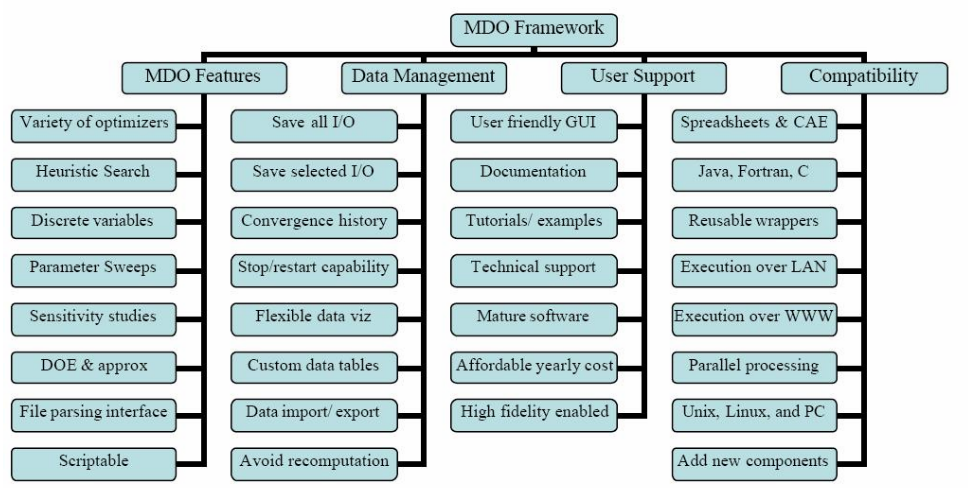
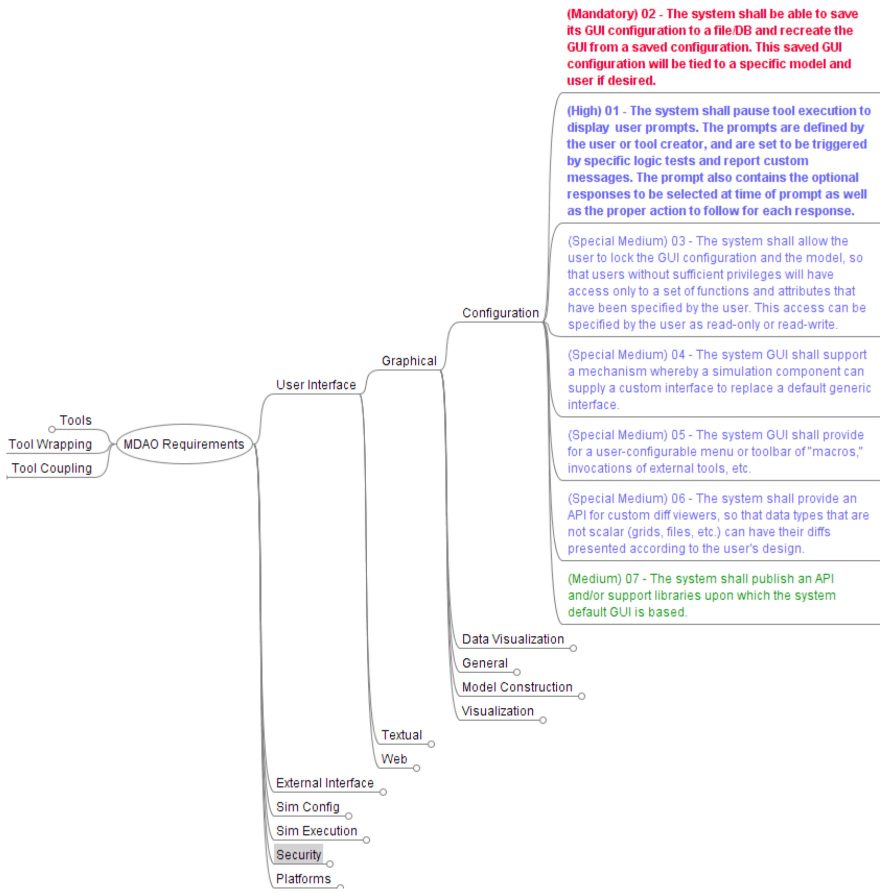


Figure 1. Characteristics of a good MDAO framework as envisioned by the CDS (Conceptual Design Shop) project.[5]

To define the MDAO requirements, an iterative requirements development process is being followed as described by Wiegers; the process consists of four stages: elicitation, analysis, specification, and validation.[15] During elicitation, the CDS requirements along with some additional requirements from NPSS were seeded as the initial requirements and through a process of formal inspections[15] by discipline experts, a new set of 577 functional requirements was produced covering both discipline tools and the MDAO framework. These requirements were then prioritized using the following categories: Mandatory, High, Medium, and Low. The priority levels were primarily based on an assessment of 1) how the requirement impacted the overall vision for MDAO in aircraft conceptual design and 2) how early the requirement needed to be implemented. An extra designation of Special Medium was added to indicate requirements of medium or lower priority which would impact the overall design and which must be considered in the architecture design from the start. In total, there are 284 requirements in the Mandatory and High categories, of which 105 are framework requirements. A small selection taken from the set of requirements is shown in Figure 2, where the set of requirements has been organized hierarchically in FreeMind[16] and sorted by priority.



(Mandatory) 02 - The system shall be able to save its GUI configuration to a file/DB and recreate the GUI from a saved configuration. This saved GUI configuration will be tied to a specific model and user if desired.

(High) 01 - The system shall pause tool execution to display user prompts. The prompts are defined by the user or tool creator, and are set to be triggered by specific logic tests and report custom messages. The prompt also contains the optional responses to be selected at time of prompt as well as the proper action to follow for each response.

(Special Medium) 03 - The system shall allow the user to lock the GUI configuration and the model, so that users without sufficient privileges will have access only to a set of functions and attributes that have been specified by the user. This access can be specified by the user as read-only or read-write.

(Special Medium) 04 - The system GUI shall support a mechanism whereby a simulation component can supply a custom interface to replace a default generic interface.

(Special Medium) 05 - The system GUI shall provide for a user-configurable menu or toolbar of "macros," invocations of external tools, etc.

(Special Medium) 06 - The system shall provide an API for custom diff viewers, so that data types that are not scalar (grids, files, etc.) can have their diffs presented according to the user's design.

(Medium) 07 - The system shall publish an API and/or support libraries upon which the system default GUI is based.

Figure 2. A selection from the MDAO framework requirements, prioritized and displayed hierarchically using the FreeMind[16] mapping software.

In addition to the functional requirements, a set of non-functional requirements was also derived to describe the characteristics of the MDAO framework as ultimately envisioned. While the functional requirements describe what a software component does as part of its function, the non-functional requirements describe in high-level terms the characteristics or attributes of the component and system as they function. Non-functional requirements are actually closely related to Software Quality Attributes[17], which are a set of qualifier nouns often called “ilities.” A list of the Software Quality Attributes that were derived from the project’s vision is shown in Table 1 generally sorted by importance.

1. Extensibility	8. Robustness
2. Flexibility	9. Reusability
3. Adaptability	10. Security
4. Usability	11. Interoperability
5. Efficiency	12. Portability
6. Maintainability	13. Safety
7. Accuracy	

Table 1. Software Quality Attributes for the MDAO Framework development sorted roughly by priority.

These are all general qualities from which specific non-functional requirements can be derived via a process of thought experiments such as described in Barbacci[17]. More importantly, these attributes codify the vision and tone of the project and can be used to help answer some very early questions, such as which language to pick for developing the framework. The qualities that are given the highest importance for the framework are Extensibility, Flexibility, and Adaptability. While each of these has its own technical definition, taken together they describe a framework that is flexible to extend its capabilities and adapt to unanticipated requirements, particularly in its ability to interface with codes or components that may not have been envisioned initially. This is probably the most important overall general quality for an MDAO framework. Codes that need to be included for a particular analysis may have been written in a wide range of languages over several decades. The framework needs to be capable not only of handling all of these codes easily, but it must also be able to adapt to future new languages, hardware, and computational paradigms. Of course, this is not to say that the other qualities are not important. If a framework were grossly inefficient or difficult to use or produced results that were inaccurate, for example, then it would be of questionable value.

Taken as a whole, the functional and non-functional requirements are an important driver in deciding whether to choose an existing MDAO framework, or to develop a new one. If none of the existing frameworks satisfies a reasonable number of the requirements, and if there are sufficient resources, then new development makes sense. Many of the existing frameworks described above were investigated, but only one of them meets the requirement that the framework be open source. This requirement is an important one, and the rationale behind it is explained in Section III – Open Source Software. This requirement ruled out the commercial frameworks, along with pyMDO which is currently not open source, and NPSS, which is also not open source. This leaves DAKOTA as the only viable existing option. However, DAKOTA is licensed under the GNU General Public License (GPL), which places restrictions on how the source can be modified to produce derivative works as well as how the framework can be used in other, possibly proprietary projects, as is explained in further detail in Section III. This is not conducive to our goal of providing a truly open framework option, so the best choice is to develop a new framework.

III. Open Source Software

Often when one hears the term “Open Source,” one of the first things that comes to mind is the Linux operating system, the essentially free non-proprietary and increasingly popular alternative to the Microsoft Windows operating system. Linux is essentially an implementation of the Unix operating system that was developed as Open Source software, and as such, includes contributions from many in the user community. Open source software can at times be free in a monetary sense, and in fact, was originally called “Free Software.” In 1998 this form of development was given a new name that more closely reflects its true definition, namely, that it is a software development strategy that fosters collaborative development while placing little to no restrictions on the final product or knowledge, in this case, the source code.[18]

NASA has recently recognized some of the advantages that can be gained through open source development and has released over 20 different projects as open source. Stated reasons for the interest in open source software are:

To increase NASA software quality via community peer review

- To accelerate software development via community contributions
- To maximize the awareness and impact of NASA research
- To increase dissemination of NASA software in support of NASA's education mission[19]

These reasons summarize a fairly general motivation for developing software using the open source model. Additional reasons why the open source development path is the preferred solution for the creation of a new MDAO framework will be discussed in this section.

A. Advantages of Open Source Software

Possibly the most compelling reason for choosing an open source path for developing an MDAO framework can be summed up as follows: *A framework is only as good as its components*. Here, a component refers both to an analysis tool that operates in the framework as well as to a tool that helps the user carry out the analysis (e.g., optimization algorithms, data visualization, design space explorers). An open source framework, provided that it is well documented, easy to use, and not deficient in some way, will attract a larger pool of users than a commercial framework, primarily because the price barrier has been removed. In particular, this larger pool of users will include researchers and students as well as those at small companies or organizations who may have effectively been "priced out" of using the major commercial codes. A larger pool of users implies that there would be a larger pool of component developers, and hence, a larger pool of components. Those components that represent enhancements to the framework are likely to also be contributed to the open source project for use by the rest of the community, which increases the available functionality for all users. Even more important than the contribution of enhancements to the framework however, is the potential for the contribution of engineering components–specialized analysis tools for the solution of some problem at a particular level of fidelity. Even the addition of more proprietary components (e.g., a code developed by a company for modeling the performance of the engine that it manufactures) contributes to the components available to a particular subgroup, namely, those who have the rights to use that component.

Another advantage that the user gains from using an open source framework is the transparency of the methods inside the code. Since the source is freely viewable, and assuming that a certain standard of readability was followed during development, the user is privy to all of the inner workings of the code. This is especially desirable for the numerical algorithms and data manipulation, where researchers will want to see every step of the problem in order to correctly interpret the simulation results. An ancillary benefit from this transparency is that many more expert eyes will pore over the details of the source code line by line, and this should lead to the discovery and correction of significantly more bugs than a small team of software developers could ever find.

The development of an open source framework also makes sense from the broader perspective of satisfying NASA's need for physics based MDAO for aeronautics. If a commercial framework is chosen, then NASA has effectively tied its whole MDAO analysis capability to one solution provided by one vendor. While NASA may have a solid relationship with that vendor, there are always risks that the company might change its focus or might go out of business altogether at some point, leaving NASA dependent on a product that has no support and no future. Of course, there is also the risk that NASA's framework development effort might end prematurely due to changes in the funding environment, and if that were to happen, an open source project would still have a chance of living on, with the development work being taken over by the open source community.

B. Choice of Open Source License

An important choice that must be made during the early stages of any open source software development is the type of license. There are quite a few different types of licenses, although in general they can be placed in two groups based on whether or not they restrict the use of the software when included in derivative works. Care must be taken to assure that the chosen license does not restrict options or exclude part of the community that is needed to foster future development. The most commonly used license is the GNU General Public License (GPL), which is used in about 60% of open-source projects (as measured by freshmeat.net, a site committed to tracking cross-platform open source software.[20]) GPL is actually the most restrictive license, in that it requires all derivative works to conform to the GPL as well; this makes it practically impossible to use GPL code as part of a closed source project, as any program that includes some GPL code must be entirely GPL. As a result, this creates some incompatibilities between GPL and certain other open source licenses.

There are also licensing options that are less restrictive than GPL, including the BSD-new (Berkeley Software Distribution out of the University of California), and the MIT/X Window System License (developed at the Massachusetts Institute of Technology originally for the X Windows system). These licenses allow the open source code to be modified to produce derived works and to be included in other projects (whether open source or closed source, free or proprietary) without restriction and without imposing requirements on the licensing of the combined work. This option seems to be the one preferred by most open source advocates because code released under one of these licenses is the most free.[21] Most of the remaining licenses, such as the LGPL (Lesser GNU Public License), fall somewhere between these two licensing styles, combining certain aspects of each of them.

As it turns out, NASA already has its own open source license called the NASA Open Source Agreement (NOSA[22]), and this license is presently used by several projects, including the satellite mapping software World Wind. While it is generally not worthwhile to create a brand new license given the presence of so many established licenses, the NOSA is officially recognized by the Open Software Initiative, so the legwork has already been done. NOSA is generally a less restrictive license than the GPL, in that it allows the open source software to be linked into larger proprietary projects without requiring those projects to adhere to the NOSA for the entire project. At the same time, the NOSA requires derivative works to remain open-source. These were the major motivations for development of this license as described in Moran[21], where the similar Mozilla Public License is recommended. NOSA is currently the only supported option available for developing an open source project at NASA; therefore, under current plans this license will be attached to the MDAO project once the code is ready for initial open source release. There is some room for further investigation however, and if NOSA turns out to be insufficient, different licensing options will be investigated.

One very important point to note is that distributing a framework under an open source license does not imply that every component that is developed for it or every analysis code that is wrapped for use in it will also be open source. A typical MDAO application is likely to use a mixture of free, or open source components, along with those developed under various other kinds of proprietary license. In fact, for these cases it is important for the framework to possess the means to restrict the access to components and models to users who have the appropriate privilege to use them, and this is reflected in the requirements.

IV. Python

A quick survey of existing frameworks reveals quite a variety of development languages, including C++, Java, Python, and even Fortran. The task of choosing a language for developing a new framework is one that should be carried out with careful consideration of the features and limitations of each language. The one language that seems to offer significant advantages over the other choices is Python.[23]

Python is a high level scripting language that was developed in the early 1990s. As a scripting language, it is interpreted instead of compiled and is concise and easy to read, bearing some resemblance to the m-scripting language used in the Mathworks' MATLAB. Python is also well suited for use in an open source environment, particularly since it was also developed (and is still being developed) as an open source project under a custom license that places no restrictions on its end use. Many add-ons and extensions to Python follow this same licensing strategy. Several characteristics of Python make it especially well suited for use as the basis of an MDAO framework, and these will be discussed in this section.

A. Wrapping Codes in Python

While a framework can do a large number of things, the one thing that it must do is facilitate the communication between disciplinary modules. The process of preparing a code to operate in conjunction with other codes in a framework is called "wrapping." Code can be wrapped in a number of ways, the simplest of which is "file wrapping." In file wrapping, the code is executed completely in batch mode and all transfer of data to and from the code is carried out by reading and writing files. Of course, this is the least efficient way to wrap a code, and it can lead to a lengthy overhead when parsing large files inside of a loop. For codes where the source or a library is available, a more efficient option is "function wrapping," where the framework code calls the module via the intermediary wrapper code, and data is passed to and from the code via memory without an excursion to the hard drive.

Python was designed to be smoothly integrated with other languages, in particular C (in which Python was written) and related languages (Fortran and C++).[24] This is particularly important for a scripting language, where code execution is generally slower, and it is often necessary to use a compiled language like C for implementing computationally intensive functions. On top of this native integration ability, the community has developed some excellent tools, such as F2PY (Fortran to Python) and SWIG (Simplified Wrapper and Interface Generator), that simplify the process of building the wrapper for a code. As the name implies, F2PY is a python utility that takes a Fortran source code file and compiles and generates a wrapped object callable from Python. F2PY is actually part of the numerical computing package NumPy, which is discussed in the next section.[25] Another tool with broader application is the Simplified Wrapper and Interface Generator (SWIG), which can be used to generate wrappers for C and C++ functions for execution in a variety of different target languages, including Python.[26] For the most general case, Python has the built-in capability to wrap any shared object or dynamically loadable library (DLL) written in any language. This "ctypes" package is a foreign function interface, and it allows an object to be wrapped without recompiling the library. Of course, care has to be taken when using ctypes to wrap a function that passes data types not native to C.

B. Useful Extensions

One of Python's most significant strengths is the sheer number of capabilities that are available, both as built-in features and as extensions, many of which are actually developed as open source software. In fact, a case can be made that Python, when taken together with all of the community-generated content, actually *is* a framework. There certainly have been some applications where codes that have been wrapped in Python are executed as part of an MDO analysis simply by writing a few scripts in Python. This section describes some of the more useful extensions that should simplify the task of developing a new framework.

Two of the most frequently used extension packages are the mathematics library NumPy[25] and the scientific computing library SciPy[27]. NumPy adds a multi-dimensional array data object with powerful indexing and slicing. This allows matrix operations requiring nested loops in C to be executed as a single line in Python. Users of MATLAB are already familiar with the improvement in code readability that this fosters. The array looping calculations are actually performed in compiled code, which allows the Python code written using arrays to be executed quickly. In addition, NumPy provides a comprehensive math library that includes numerical solution of ordinary differential equations, finite Fourier transform algorithms, convolution, and random number generation, just to name a few. SciPy builds on the capabilities of NumPy and includes a number of specialized packages, such as linear algebra (using the BLAS (Basic Linear Algebra Subprograms) and LAPack (Linear Algebra PACKage) libraries), statistics, optimization, integration, interpolation, a sparse array capability, and a signals and systems package for signal and image processing. Another extension called Scientific Python (ScyPy) adds other useful capabilities, particularly vector transformations, interpolation, nonlinear least-squares fits, and a systematic way of performing unit calculations with automatic unit conversion.[28] More information on using Python for scientific computing can be found in an article by SciPy creator T. Oliphant.[29]

SciPy also contains optimization algorithms via the optimize module, including both gradient-based methods and genetic algorithms. However, any optimization algorithm that is already coded and in the public domain can also be made available to users in the proposed MDAO framework.

There are also quite a few extensions that can be used to add graphics capability to the framework. A cross-platform open source GUI development tool called wxPython[30], which is actually a Python-wrapped implementation of the wxWindows GUI toolkit, could ease the construction of a graphical interface for the framework. There are also a few other GUI development kits, including one based on Trolltech's QT[31] cross-platform Widgets. For 2D plotting, one of the most useful python extensions is matplotlib[32], which provides a plotting capability which reproduces most of what is provided by MATLAB both in style and syntax. There are other options for 3D visualization, including MayaVi2[33], which is a scientific data visualization tool developed for use in Python.

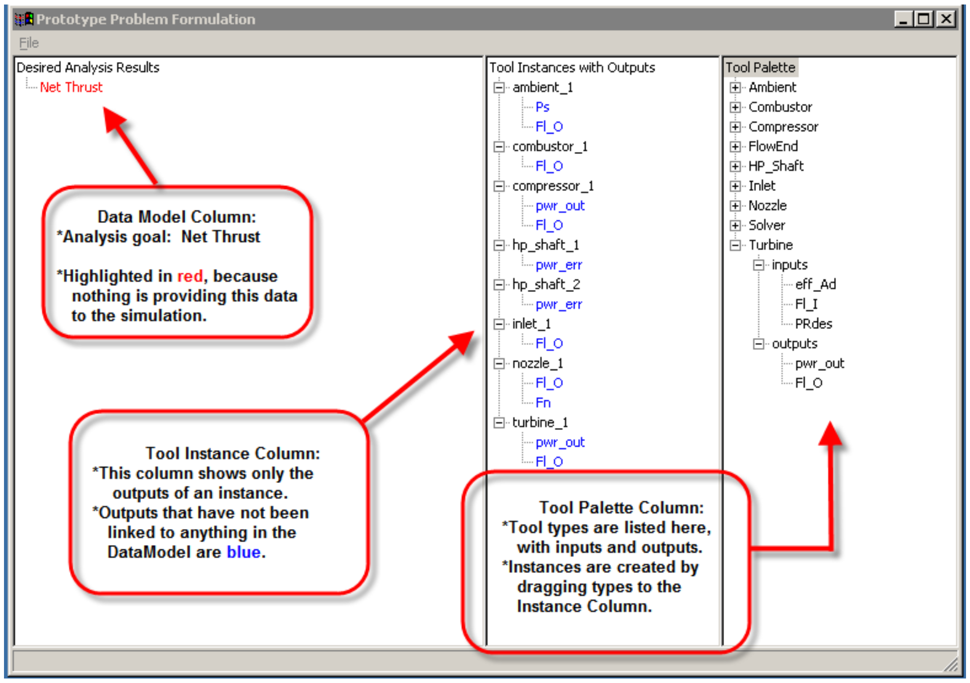
There are literally thousands of other extension packages for Python, covering many more specialized topics and mostly distributed under one of the open source licenses. Taken together, these all make a powerful case for the choice of Python as the best development language for an MDAO framework. Other scripting languages were considered (TCL, Ruby) as well as a compiled language (C++). None of these matched Python in terms of the features it offered or its ability to achieve the quality attributes (e.g., Flexibility, Adaptability, and Maintainability) that are required for this framework.

V. Framework Prototype Ideas

The effort to develop the new open source MDAO framework is still in its very early stages; the requirements have been defined, and currently the architecture is being designed. While it is too early to realistically present a full design, a few prototype ideas covering various aspects of the envisioned framework are being investigated. One idea that has received the most attention so far is actually a tool for helping an engineer formulate an MDAO problem starting with a set of desired output quantities and a list of all the tools that are available. This prototype is referred to here as a data-driven problem formulation tool because the user constructs a solution path by finding tools that supply the open inputs, instead of starting with a given set of tools that are typically used for a given analysis.

Consider a problem where the analysis goal is to calculate the net thrust of an engine. A number of computational tools exist for providing some part of the solution to the engine flowpath (e.g., Compressor, Inlet, Nozzle, etc.), but the exact order of the tool execution is not known at the outset. Moreover, some subset of the tools may cause a circular dependency, or there may be a case where a subset of the tools is interdependent in such a way that an output from that set is also an input. Such a case requires a numerical solver to resolve the dependency. For complicated problems with many tools spanning multiple disciplines, it is not always clear exactly how to set up the problem in an MDAO environment so that it can be solved. This is particularly true if the user tries to build the problem using a typical data flow representation. This is where a new way to view the problem can be very useful.

To test the concept for a data-driven problem formulation tool, a prototype GUI was built in wxPython and is shown in Figure 3.



Data Model Column:
*Analysis goal: Net Thrust

*Highlighted in red, because nothing is providing this data to the simulation.

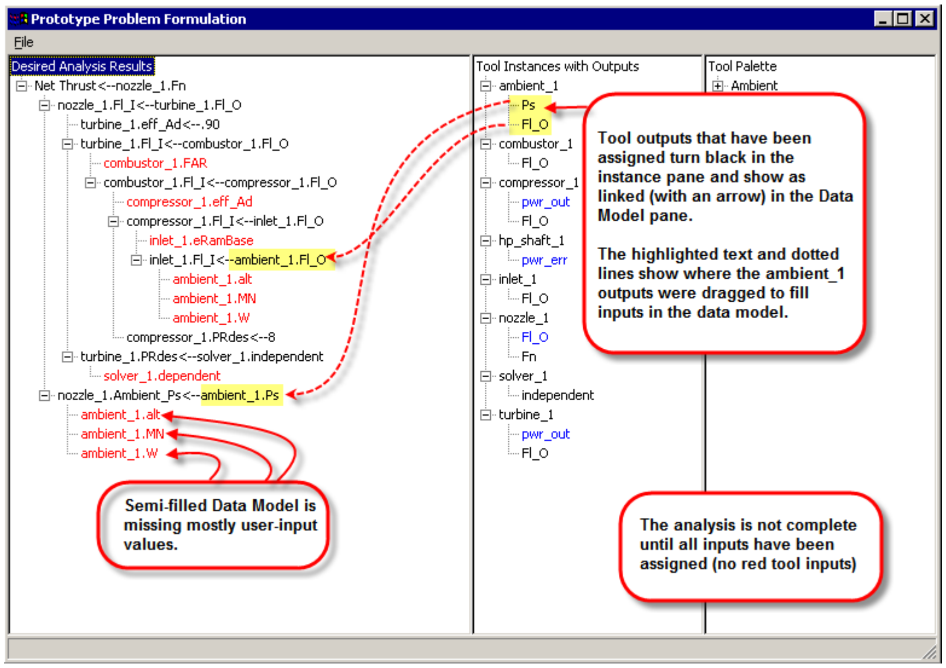
Tool Instance Column:
*This column shows only the outputs of an instance.
*Outputs that have not been linked to anything in the DataModel are blue.

Tool Palette Column:
*Tool types are listed here, with inputs and outputs.
*Instances are created by dragging types to the Instance Column.

Figure 3. Data-driven problem formulation tool at the initial set up of a problem for calculating the net thrust of an engine.

The rightmost panel contains a list of all of the tools that are currently available. Expanding any of these tools reveals a list of variable names that are inputs for the tool and a list of variable names that are outputs when the component is executed. The middle panel contains tool instances; note that there can be multiple instances of a tool. Finally, the left panel is where the data model will be assembled. The starting point for this problem has been entered as “Net Thrust” and is shown in red, indicating that it has not been assigned to a tool output. A key prerequisite for building a problem is that each tool has a well-defined interface. In other words, the inputs and outputs must be defined and given names that are clear and consistent, so that inputs and outputs can be matched correctly.

To begin assembling the problem, an instanced component must be found that can provide thrust. This component is the nozzle via a variable called “fn.” To assign the output of the nozzle to the starting goal, the fn output is simply dragged to the data model panel and onto Net Thrust, and it is connected. Now the inputs that are required by the nozzle code become the next open unassigned variables in the data model. The model is built up further from here by assigning these variables and adding tool instances as needed, until they have all been filled. Figure 4 shows an example of a partially completed problem that includes several tools and a solver.


Prototype Problem Formulation

File

```
Desired Analysis Results
Net Thrust<--nozzle_1.Fn
  nozzle_1.Fl_I<--turbine_1.Fl_O
    turbine_1.eff_Ad<--.90
    turbine_1.Fl_I<--combustor_1.Fl_O
      combustor_1.FAR
      combustor_1.Fl_I<--compressor_1.Fl_O
        compressor_1.eff_Ad
        compressor_1.Fl_I<--inlet_1.Fl_O
          inlet_1.eRamBase
          inlet_1.Fl_I<--ambient_1.Fl_O
            ambient_1.alt
            ambient_1.MN
            ambient_1.W
        compressor_1.PRdes<--8
    turbine_1.PRdes<--solver_1.independent
      solver_1.dependent
  nozzle_1.Ambient_Ps<--ambient_1.Ps
    ambient_1.alt
    ambient_1.MN
    ambient_1.W
```

```
Tool Instances with Outputs
ambient_1
  Ps
  Fl_O
combustor_1
  Fl_O
compressor_1
  pwr_out
  Fl_O
hp_shaft_1
  pwr_err
inlet_1
  Fl_O
nozzle_1
  Fl_O
  Fn
solver_1
  independent
turbine_1
  pwr_out
  Fl_O
```

```
Tool Palette
Ambient
```

Tool outputs that have been assigned turn black in the instance pane and show as linked (with an arrow) in the Data Model pane.

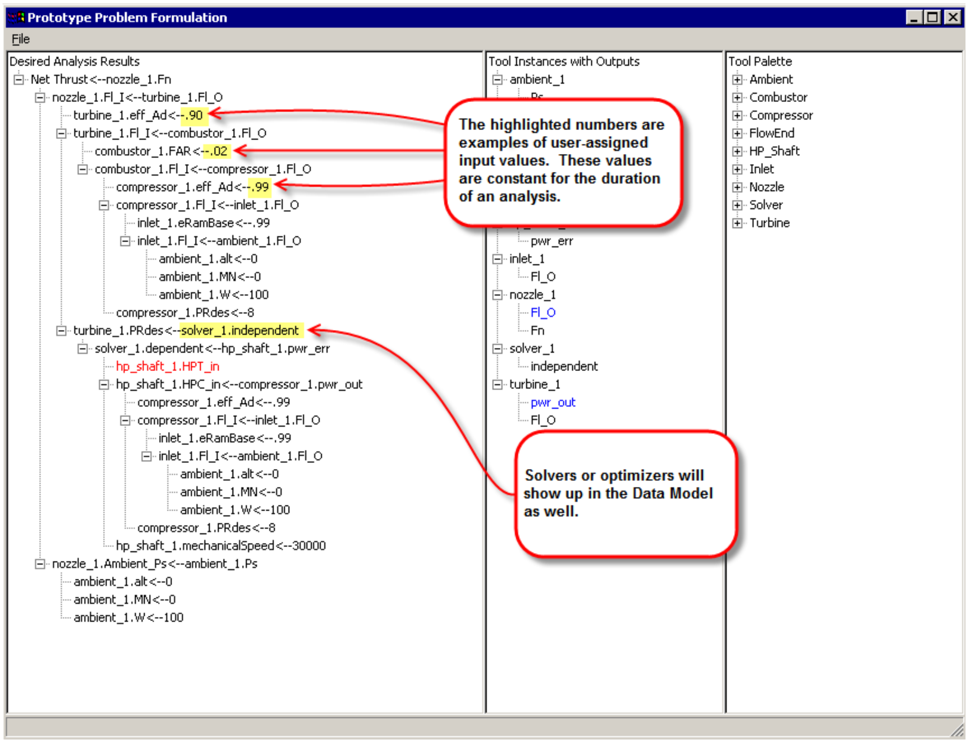
The highlighted text and dotted lines show where the ambient_1 outputs were dragged to fill inputs in the data model.

Semi-filled Data Model is missing mostly user-input values.

The analysis is not complete until all inputs have been assigned (no red tool inputs)

Figure 4. The problem has been partially assembled by dragging tool outputs from the tool instances panel and hooking them up to the unassigned inputs in the data model.

Not every tool input requires a tool to be assigned from the instances panel. Some inputs are merely constants and can be satisfied by entering a value manually. Other inputs may be supplied by some kind of driver component, such as a solver, optimizer, or design space explorer. Once every required input to every component has been supplied in the model, there are no remaining unassigned variables, and the problem can, at that point, be considered formulated and ready for solution. In the example problem, all inputs have been assigned except for hp_shaft_1, as shown in Figure 5.


Prototype Problem Formulation

File

```
Desired Analysis Results
Net Thrust<--nozzle_1.Fn
  nozzle_1.Fl_I<--turbine_1.Fl_O
    turbine_1.eff_Ad<--.90
    turbine_1.Fl_I<--combustor_1.Fl_O
      combustor_1.FAR<--.02
      combustor_1.Fl_I<--compressor_1.Fl_O
        compressor_1.eff_Ad<--.99
        compressor_1.Fl_I<--inlet_1.Fl_O
          inlet_1.eRamBase<--.99
          inlet_1.Fl_I<--ambient_1.Fl_O
            ambient_1.alt<--0
            ambient_1.MN<--0
            ambient_1.W<--100
        compressor_1.PRdes<--8
    turbine_1.PRdes<--solver_1.independent
      solver_1.dependent<--hp_shaft_1.pwr_err
        hp_shaft_1.HPT_in
        hp_shaft_1.HPC_in<--compressor_1.pwr_out
          compressor_1.eff_Ad<--.99
          compressor_1.Fl_I<--inlet_1.Fl_O
            inlet_1.eRamBase<--.99
            inlet_1.Fl_I<--ambient_1.Fl_O
              ambient_1.alt<--0
              ambient_1.MN<--0
              ambient_1.W<--100
          compressor_1.PRdes<--8
        hp_shaft_1.mechanicalSpeed<--30000
  nozzle_1.Ambient_Ps<--ambient_1.Ps
    ambient_1.alt<--0
    ambient_1.MN<--0
    ambient_1.W<--100
```

```
Tool Instances with Outputs
ambient_1
  Ps
  pwr_err
inlet_1
  Fl_O
nozzle_1
  Fl_O
  Fn
solver_1
  independent
turbine_1
  pwr_out
  Fl_O
```

```
Tool Palette
Ambient
Combustor
Compressor
FlowEnd
HP_Shaft
Inlet
Nozzle
Solver
Turbine
```

The highlighted numbers are examples of user-assigned input values. These values are constant for the duration of an analysis.

Solvers or optimizers will show up in the Data Model as well.

Figure 5. A solver has been added to the analysis to address a circular dependency.

The problem that was showcased above (really just a conventional jet engine) may be one for which the engineer has at least a general idea about what codes and solution procedures are involved for a typical solution. However, there are other problems, particularly those that arise from unconventional aircraft configurations, for which a significant body of past analyses does not exist. These kinds of problems will benefit the most from tools that help the engineer visualize and understand the best way to find a solution.

VI. Conclusion

While much work lies ahead, some of the ideas that will form the backbone of the open source MDAO framework have been completed. Over the course of the next year, it is hoped that considerable progress will be made towards demonstrating a working framework capable of solving design problems.

Acknowledgments

The authors would like to acknowledge the contributions of the other team members who have expended considerable effort in researching alternatives, defining requirements, and building and investigating prototypes. In particular, they would like to extend their gratitude to Keith G. Marsteller of Wyle Information Systems, Scott Townsend of Artic Slope Research Corporation, Natalia Alexandrov of NASA Langley Research Center, and the MDAO Working Group of the SAD&O (System Analysis Design & Optimization) discipline in the Subsonic Fixed Wing Project under the Fundamental Aeronautics Program. This work has been funded by the Subsonic Fixed Wing project under the Fundamental Aeronautics Program at NASA.

References

[1]"Vision and Scope Document for the Fundamental Aeronautics Program Multidisciplinary Analysis and Optimization (FAP MDAO) Capability," May 18 2007.

[2]Chiba, K., Obayashi S., Nakahashi K., and Morino H, "High-Fidelity Multidisciplinary Design Optimization of Aerostructural Wing Shape for Regional Jet," *23rd AIAA Applied Aerodynamics Conference*, June 6 2005.

[3]Alonso, J. J., LeGresley, P., Van der Weide, E., Martins, J. R. R. A., and Reuther, J. J., "pyMDO: A Framework for High-Fidelity Multi-Disciplinary Optimization," *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, August 2004.

[4]Perez, R., and Martins, J., "An Object-Oriented Framework for Aircraft Design Modeling and Multidisciplinary Optimization," *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Sept 10 2008.

[5]Padula, S. L. and Gillian, R. E., "Multidisciplinary Environments: A History of Engineering Framework Development," *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Sept 6 2006.

[6]Neill, D. J., "ASTROS–A Multidisciplinary Automated Structural Design Tool," *Recent Advances in Multidisciplinary Analysis and Optimization*, NASA CP-3031, 1988, pp. 529-543.

[7]Hopkins, D. A., Patnaik, S. N., and Berke, L, "General-purpose Optimization Engine for Multidisciplinary Design Applications," *6th AIAA Multidisciplinary Analysis and Optimization Conference*, CP9611, AIAA, Reston, VA, 1996, pp. 1558-1567, AIAA-1996-4163.

[8]Golovidov, O., Kodiyalam, S., Marineau, P., Wang, L., and Rohl, P., "Flexible Implementation of Approximate Concepts in an MDO Framework," *7th AIAA Multidisciplinary Analysis and Optimization*, CP9811, AIAA, Reston, VA, 1998, pp. 1969-1979, AIAA-1998-4959.

[9]Eldred, M. S., Giunta, A. A., van Bloemen, B. G., Wojtkiewicz S. F. Jr., Hart, W. E., Alleva, M. P., *DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis*, Version 3.0 User Manual, SDAND2001-3796, April 2002.

[10]Gray, J. G., and Briggs, J. L., "Design of a Model Execution Framework: Repetitive Object-Oriented Simulation Environment (ROSE)," *44th AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, Jul. 2008.

[11]Jones, S. M., "An Introduction to Thermodynamic Performance Analysis of Aircraft Gas Turbine Engine Cycles Using the Numerical Propulsion System Simulation Code," NASA TM-2007-214690, Mar., 2007.

[12]Claus, R. W., Lavelle, T., Naiman, C., Suresh, A., and Turner, M., "Progress Toward and the Promise of Multidiscipline Engine Simulation," *13th ISPE International Conference on Concurrent Engineering*, Sept 2006.

[13]Lytle, J. K., Numerical Propulsion System Simulation: An Overview," *CAS 2000 Workshop*, The Ames Research Center, Feb. 15, 2000.

[14]Mukhopadhyay, V., Hsu, S., Mason, B. H., Hicks, M. D., Jones, W. T., Sleight, D. W., Chun, J., Spangler, J. L., Kamhawi, H., and Dahl, J. L., "Adaptive Modeling, Engineering Analysis and Design of Advanced Aerospace Vehicles," *47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, May 1-4 2006.

[15]Wiegers, C. E., *Software Requirements, Second Edition*, Microsoft Press, Redmond WA, 2003.

[16]http://freemind.sourceforge.net/wiki/index.php/Main_Page

[17]Barbacci, M., Klein, M. H., Longstaff, T. A., and Weinstock, C. B., *Quality Attributes*, CMU/SEI-95-TR-021, Dec 1995.
[18]Fogel, C., *Producing Open Source Software*, O'Reilly, London, Oct. 2005.
[19]NASA Ames, Open Source Software: http://opensource.arc.nasa.gov/
[20]License Breakdown, Top 20: http://freshmeat.net/stats/#license
[21]Moran, P. J, "Developing an Open Source Option for NASA Software," NASA TR NAS-03-nnn, April 7 2003.
[22]NASA Open Source License: http://opensource.arc.nasa.gov/static/html/NASA_Open_Source_Agreement_1.3.txt
[23]Python: http://www.python.org
[24]Langtangen, H. P., *Python Scripting for Computational Science*, Third Edition, Springer-Verlag, Berlin, 2008.
[25]NumPy: http://numpy.scipy.org/
[26]SWIG: http://www.swig.org/
[27]SciPy: http://www.scipy.org/
[28]ScyPy: http://dirac.cnrs-orleans.fr/plone/software/scientificpython
[29]Oliphant, T. E., "Python for Scientific Computing," *Computing in Science and Engineering*, May 2007, pp 10-20.
[30]wxPython: http:// www.wxpython.org
[31]qt: http://trolltech.com/products/qt/
[32]matplotlib: matplotlib.sourceforge.net
[33]MayaVi2: https://svn.enthought.com/enthought/wiki/MayaVi