2001-01-3034

Development of an Object Oriented Vehicle Library for Automated Design Analysis

Julien Scharl [*], **Dimitri N. Mavris** [†]
Aerospace Systems Design Laboratory
Georgia Institute of Technology
Atlanta, GA 30332

ABSTRACT

In today's emerging parametric and probabilistic design environments, disciplinary or multidisciplinary analysis data are represented efficiently with the use of metamodels. Each metamodel is an efficient replacement for a particular design analysis tool. An object oriented library is developed in this paper to represent vehicle configuration in a generic manner and assist the analysis data collection for the metamodeling process. The library is used to produce input files for design analysis tools. It can also be used to create preprocessors for integration environments used in the design process. This allows for smoother integrations of analysis programs within such environments as the environment now needs only replace data in one central input file rather than a file for each analysis tool.

INTRODUCTION

Breakthrough design methods being developed today center on system modeling and simulation around parametric design space exploration and uncertainty assessments [1–3]. These methods rely on the generation of parametric and probabilistic system models through the synthesis of lower-level or disciplinary models whose goal is to capture the essence of a design discipline in a simplified form. Each design discipline related to the design problem at hand should be modeled to enable physics-based engineering analyses used to generate design knowledge and data upon which design decisions are made.

The most common form of a physics-based disciplinary model is a computer program instantiating the physical principles pertinent to that discipline. A typical example is the use of linear aerodynamic models based on lifting line theory or potential flow theory to estimate aerodynamic characteristics. The fidelity of such models is proportional to the degree to which the computer program captures the physical phenomena at play. Naturally, the highest degree of fidelity is always desired to reduce the overall uncertainty in the design process. However, one quickly finds that execution time of analysis programs grows exponentially with increasing fidelity requirements. As a result, their inclusion within a large parametric design architecture comes at the intolerable cost of longer cycle times.

The classical tradeoff in design modeling and analysis is between efficiency and accuracy. One seeks the most efficient way of obtaining data of highest fidelity. An innovative way of obtaining accurate data efficiently is through the use of metamodels. A metamodel is a model of a model; it is an efficient replacement of the model itself. Metamodels seek to form a more compact representation of the functional relationships between the inputs and outputs of a model. Each disciplinary model must have a corresponding metamodel for efficient inclusion in the system model synthesis process. Consequently, a metamodel of each disciplinary analysis program is sought.

The metamodeling process is equivalent to function approximation: given a set of data points (model inputs $\vec{x}$, model outputs $\vec{y}$) what is the function f that best approximates the relationship $\vec{y} = f(\vec{x})$? Many techniques exist for this problem, varying from linear regression based on polynomials [4], to local model networks [5] and neural networks [6]. Independently of the metamodeling technique used, the metamodeling process starts with data collection. In multidisciplinary design, this corresponds to execution of disciplinary analysis tools, a process referred to as "virtual experimentation".

Each discipline involved in the design process relies on its own analysis tool for the generation of disciplinary data that are used for the metamodeling process. Each tool has different means of input and output and may operate on different computing platforms, which may or may not physically be at the same location. Means of linking analysis programs in a concise, central, computational in-

[*] Graduate Research Assistant
[†] Professor, Dir. ASDL, Boeing Chair in Advanced Aerospace Systems Analysis

frastructure for the purpose of generating model data is needed.

BACKGROUND

Research in this area in both academia and industry has resulted in environments such as IMAGE [7], iSIGHT® by Engineous Software, and Model Center® by Phoenix Integration. These integration environments allow for a graphical representation of data flow between analysis tools, automatic parsing of analysis output, and wrapper tools for multiple executions supporting such techniques as design of experiments, Monte Carlo simulations, Fast Probability Integrations, etc. Figure 1 conceptually depicts such environments. Elements shaded in gray are enabling elements:

- *Internal representation of design data*
  Design data needs to be internally represented so that it can be passed to the pre-processing tools
- *Pre-processing tools*
  These tools are responsible for replacing data in the analysis input files corresponding to current design data. This can involve text substitution and logic if needed.
- *Execution environment*
  The central element in the integration environment is the execution environment. This should allow for a graphical representation of the flow of analysis execution as well as tools to enable different means of execution (serial, parallel, networked etc.)
- *Parsing tools*
  Analysis output needs to be parsed to recover analysis data (responses) of interest.
- *Internal representation of analysis results*
  Analysis output needs to be internally represented if any manipulation of output or logic involving output is needed (such as unit conversion, transformations, etc.)
- *Multiple executions wrappers*
  The ability to parametrically evaluate analysis output with design data requires multiple executions of the analysis codes. This corresponds to “virtual experimentation”, which is the essential first step of the metamodeling process. Multiple execution may also be used for probabilistic assessments of responses with Monte Carlo simulations or Fast Probability Integration.

The effort presented in this paper originated from the need for an automated panelizing pre-processor to HASC, a potential flow based aerodynamic analysis tool [8]. Integration environments typically replace data in the input files of analysis tools corresponding to current design data. This is not possible in the case of HASC since its input file needs a panelized representation of the vehicle configuration. An Object Oriented (OO) approach was taken to create a pre-processor to HASC that would automatically create the panel-based configuration representation within its input file. The JAVA language was chosen for portability and the possibility of seamless Internet integration.

The OO approach to this HASC pre-processor quickly turned into a generic representation of vehicle configuration. After several improvements and modifications, the project resulted in a OO vehicle library that is now used to create any analysis input from a single input file, referred to as the *metafile*. When used within an integration environment, the library allows for the creation of preprocessors that include all the necessary logic to create each analysis tool’s input file. This allows the environment to be shielded from the individual analysis tools’ input files. Instead of individually pre-processing each input file separately, the integration environment now only needs to change one central file. As a result, the OO library renders the pre-processing block of Figure 1 implementable much faster and without having to write logic in an unfamiliar or proprietary way.

OO VEHICLE MODELING

In Object Oriented Design (OOD), system components are described as objects with encapsulated data and methods. This allows for abstraction and separation of the *what* (class data) from the *how* (class methods) [9]. Classes are computational blueprints of system elements. Objects are created as instantiations of a given class. Classes are related to each other through hierarchy, where common elements (data and/or methods) within a hierarchy structure are shared through inheritance. Encapsulation, instantiation (constructing and destructing objects) and inheritance are three fundamental concepts of OO programming.

ASSOCIATION Object Oriented modeling of a fixed-wing vehicle configuration starts with the identification of common individual components and their association. A single fixed-wing configuration can be separated into the following components:

- Fuselages
- Lifting Surfaces (wing, tails, canards)
- Control Surfaces (flaps, slats, spoilers), and
- Nacelles

Component (object) association for a fixed wing vehicle is shown in Figure 2. A configuration represents a collection of fuselages, lifting surfaces and nacelles. Lifting surfaces and nacelles are attached to fuselages; control surfaces and nacelles may be attached to lifting surfaces.

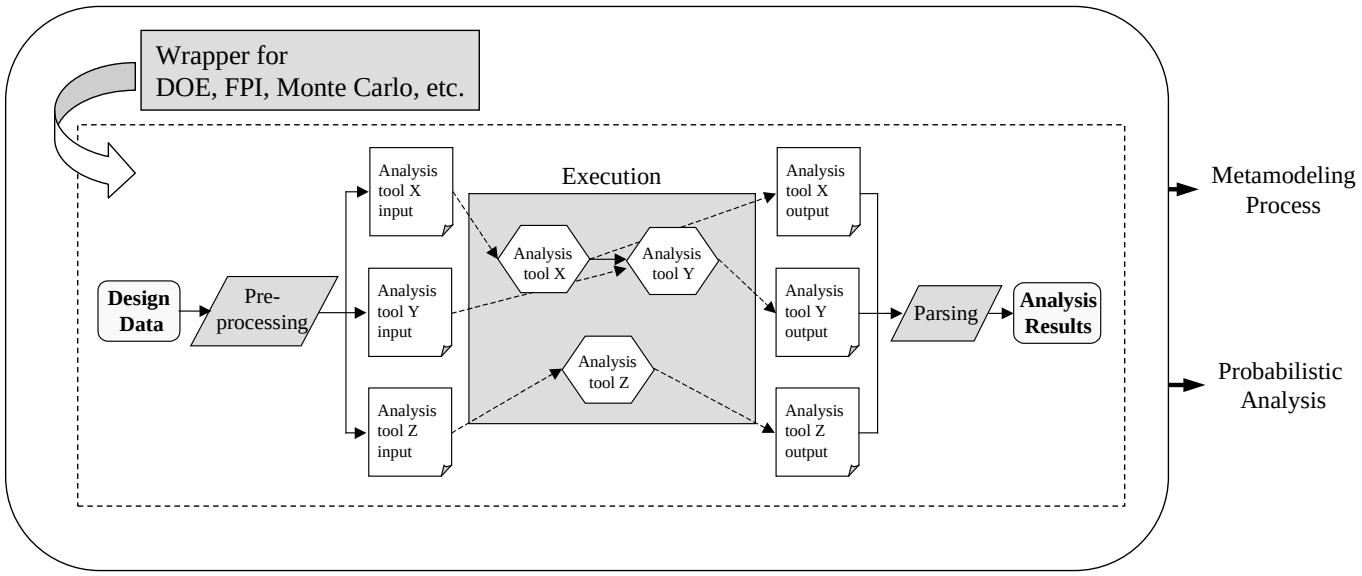

Figure 1: Elements of integration environments for design analysis

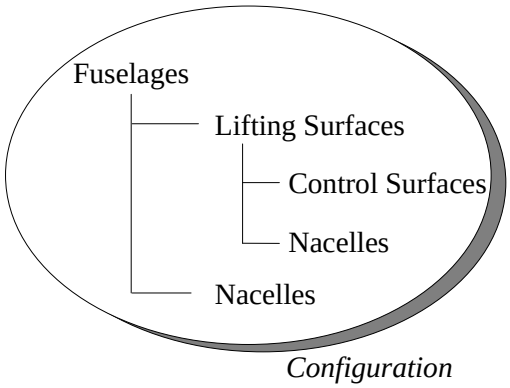

Figure 2: Structural breakdown of a fixed-wing vehicle configuration

ABSTRACTION Although each configuration element in the above list and in Figure 2 is easily identified as an object, its properties may be defined differently by different people. For example, designer A may describe a wing in terms of aspect ratio, taper ratio, sweep and area while designer B will think of it in terms of root chords, tip chords, sweep and span. Moreover, both these definitions are valid for simply tapered wings but need to be extended for cranked or kinked surfaces.

As a result, a description of the fundamental properties of each object in the most abstract sense is required. This results in a set of abstract classes (classes that cannot be instantiated) that are used as the foundational blueprints to create individual implementations through inheritance. For example, the abstract lifting surface class is used to define a simply tapered surface (the SimpleWing class); the abstract control surface class is used to define trailing edge surface (the Flap class) or a leading edge surface (the Slat class). Figure 4 describes the abstract classes that are part of the library and examples of simple implementations that were created from them. Each box describes a class (abstract classes are shaded) with the class name on top, class methods in the middle and class data on the bottom.

The Fuselage abstract class In the most abstract sense, a fuselage can be represented as a collection of planar cross-sections ordered in a given way. Basic properties such as length, maximum width and maximum height can be extracted from the collection of cross-sections. All fuselage objects must be able to return such properties as well as add and remove cross-sections.

The LiftingSurface abstract class Abstractly, a lifting surface can be represented by a set of points in a 2-dimensional cartesian space defining its plane, dihedral, and a collection of airfoils distributed in a given manner to define its thickness, camber and twist distribution. Basic geometric properties such as area and mean geometric chord can be extracted from this data. All lifting surface objects must be able to return such properties as well as add and remove points, airfoils and control surface objects. Finally, lifting surfaces may be moved in the x and z direction and installed on a fuselage.

The ControlSurface abstract class Control surfaces may be modeled as quadrilaterals positioned in a prescribed manner on a lifting surface. The size of this quadrilateral may be abstractly defined in terms of a percentage span (pSpan) and root chord (pChord) of the lifting surface it is attached to and a taper ratio. Placement of the control surface on a lifting surface is described in terms of the percentage of exposed span at which the root chord of the control surface is placed on the lifting surface (ypPos). Figure 3 shows how the ControlSurface properties are defined and how they can be used to create ControlSurface objects (here for trailing edge surfaces, created through the Flap class).

Control surfaces also have different means of deflection

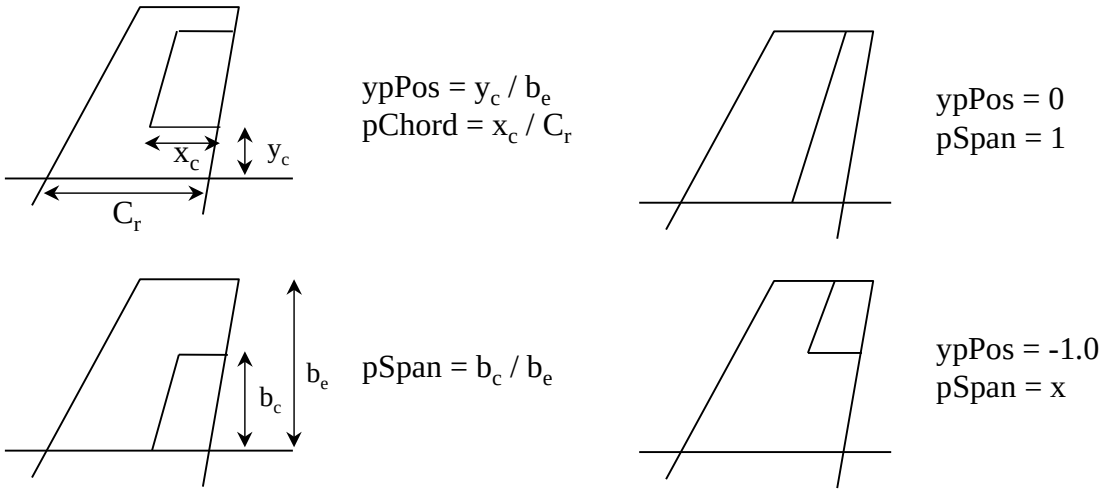

Figure 3: ControlSurface class properties definitions and example

varying from hinged deflection (plain flaps and slats) to translational and rotational deflection (Fowler flaps). Deflections are limited in the positive (maxDef) and negative (minDef) direction.

The Nacelle class Nacelles may also be considered as a set of planar cross-sections. They have basic properties such as length and maximum width and height. Typically, more than one nacelle is part of a configuration. Nacelles that come in pairs are physically the same (for symmetry) but are placed on the wing or fuselage as mirror images of each other. Each nacelle object represents the starboard nacelle if it is part of a pair, and a mirror object is automatically created.

INHERITANCE The abstract classes in Figure 4 allow for the creation of particular implementations through the inheritance process. Individual LiftingSurface implementations contain a different representation of the same basic class. To exemplify the concept, a simply-tapered lifting surface object is created by inheritance from the abstract lifting surface class.

The SimpleWing class Although the abstract LiftingSurface class is merely a set of points, a simply tapered surface is typically defined in terms of a combination of root chord, tip chord, span, sweep, area, aspect ratio, and taper ratio. Means of translating these basic properties into the abstract set of points need to be part of the SimpleWing implementation of the abstract LiftingSurface class.

The SimpleWing class inherits both class data and methods from the LiftingSurface abstract class, as shown in Figure 4. In addition to the inherited data, properties such as root chord, tip chord, sweep, span, etc. are added to the SimpleWing class. Additional methods allow for the addition of kinks. The basic properties of a simply tapered lifting surface must be related to the set of points contained in the LiftingSurface abstract class. This is done through logic contained in the `setPoints()` method.

INITIALIZATION FROM THE METAFILE The key component of the OO vehicle model is the ability to construct the configuration object from one central file. The structure chosen for the metafile is the standard FORTRAN namelist format. The library includes a NamelistsParser class which can be used to retrieve data from and write data to namelist formatted files.

The Vehicle class allows for the creation of LiftingSurface objects using the SimpleWing class, Fuselage objects using the SimpleFuselage class and ControlSurface objects using the Flap and Slat classes from the metafile. The metafile must contain a namelist for each component of the configuration that is to be created. The following configurations can be automatically generated, depending on the information available in the metafile:

- wing-alone
- wing and tail alone
- wing and canard alone
- wing and fuselage
- wing, fuselage and tail
- wing, fuselage and canard

An example metafile for a wing and tail configuration is shown in Figure 5. The OO vehicle library provides all the tools necessary to create a variety of configurations using the SimpleWing, SimpleFuselage, Flap and Slat implementations either through code or from the metafile. Additionally, the abstract classes provide an unequivocal representation of configuration components that may be used to define simpler or more complete components through the inheritance process.

Figure 6 shows top-view HASC panel representations of two drastically different configurations. The configuration on the left is that of a Boeing 747 with nacelles, flaps, elevators and ailerons. The configuration on the right is

Figure 4: Vehicle OO library classes description

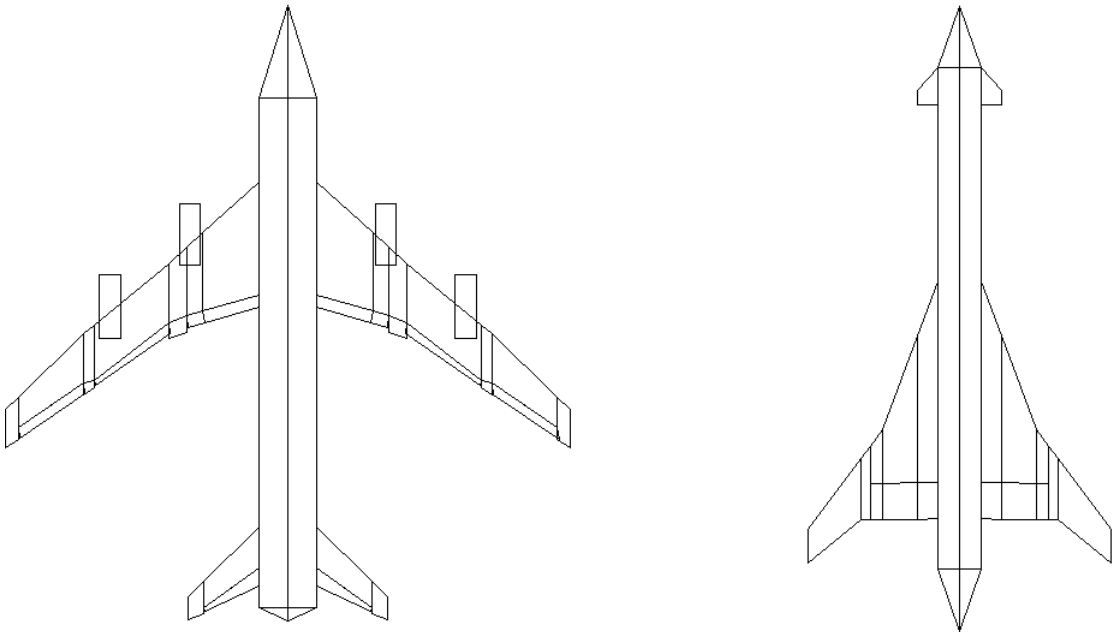

Figure 6: HASC representation of a Boeing 747 (left) and a representative supersonic business jet (right)

```
$Wing
  XW=0.3,CR=21.333,CT=5.333,Span=120.0,Sweep=33.25,
  dihedral=2.0,control1=flap,
$END

$HT
  XHT=0.8,AR=5.3,area=311.8,taper=0.25,Sweep=37.17,
  control1=elevator,
$END

$OPER
  MACH=0.3,ALPHA=0.0,H=0.0,BETA=0.0,PITCHQ=0.0,
  ROLLP=0.0,YAWR=0.0,
$END

$flap
  pspan=0.41,pchord=0.25,type=flap,name=flap,yppos=0.0,
  taper=0.9,asym=false,def=0.0,
$end

$elevator
   type=flap,name=elevator,pspan=0.55,pchord=0.3,taper=0.7,
   asym=false,def=0.0,ypPos=0.0,
$end
```

Figure 5: Example metafile for wing+tail configuration

that of a representative supersonic business jet with all moving canards and a flap on the main wing. All control surfaces on both configurations are shown deflected. Both these configurations were created with minimal effort from metafiles similar to that of Figure 5 (although more complex).

OO ANALYSIS INPUT FILE MODELING

Once an OO representation of the vehicle is available, it can be used to generate input files for analysis programs using OO modules. Each module, referred to here as an analysis input module (AIM), is represented as a class with data corresponding to input file data and methods to supply the logic necessary to transform configuration data contained in the vehicle object into data appropriate for the specific input file. For example, a HASC input module needs to transform configuration data into surfaces and panels.

AIM objects are simply constructed with a single argument, the vehicle configuration object. The OO vehicle object built with the classes of Figure 4 guarantees that all configuration information is contained within the configuration object, which makes the creation of input modules a fast and simple process. AIMs for HASC, BDAP (a design analysis program used to compute friction and wave drag [10]) and FLOPS (a system sizing and synthesis program [11]) have been developed using the OO vehicle library. These modules are shown on the right side of Figure 9. Each AIM may contain other objects representing sections or parts of the input file (such as cards, namelists, etc.) if needed. Finally, in addition to the logic needed to convert configuration data into input data, each input object needs to have a means to print itself in the correct format.

OO VIRTUAL EXPERIMENTATION MODELING

The OO vehicle and AIM objects permit the creation of input files for each analysis tool from one central file. To allow for the virtual experimentation necessary to collect data for the metamodeling process, wrappers for multiple executions are needed. Integration environments such as IMAGE and iSIGHT® provide such wrappers. Nonetheless, to test the library independently of an integration environment, classes representing elements of these wrappers were created.

One of the key questions when building a metamodel is how much data to collect. Design of Experiments (DOE) is a technique developed to answer this question [4, 12]. DOE methodology helps keep the number of required experiments to a minimum by carefully selecting designs so

```
#Namelist      Variable      Min           Max     ####

WING           XW            0.3           0.55
WING           span          100.0         140.0
WING           Cr            18.0          25.0
WING           ct            3.0           7.5
WING           Sweep         28.0          40.0
HT             XHT           0.65          0.82
HT             AR            9.0           15.0
HT             area          1.5           5.0
HT             sweep         35.0          40.0
OPER           MACH          0.0           0.9
OPER           BETA          -5.0          5.0
flap           def           0.0           40.0
elevator       def           -20.0         20.0
```

Figure 8: Example file for parametric space creation

that the model accounts only for the effects of interest. DOEs are typically used to generate data for polynomial-based metamodels through regression. As a result, DOEs are tailored for a given a priori model. In some cases the a priori model may turn out to be of insufficient predictive power and other models such as neural networks may be used. In those cases, random experiments are needed.

Additionally, a shortcoming of DOEs is that they tend to be corner-based. For large design spaces, the extreme combinations of variables that result at such corners may lead to experiments that are either not accepted by a particular analysis tool or for which no data can be generated. Again, this can be remedied by generating random experiments.

Each "virtual experiment" in the metamodeling data gathering process corresponds to a setting of a number of variables determined either randomly, or by a DOE table. Four classes are necessary to enable object oriented-based virtual experimentation with the OO vehicle library and AIMs: a variable class, a parametric space class, a DOE class and a random experiment generator. These classes are described in Figure 7.

The variable class The variable class represents a continuous scalar variable whose value falls within a given range. Each variable has a unique name and may be part of a namelist in a given file. Each variable object must be able to return its current value, which is assigned externally either by a DOE or random number object.

The parametric space class The parametric space object represents a collection of variables. The parametric space object is constructed from a file that describes the variables that are to be varied, their range and the namelist where the variable can be found in the metafile. An example of such a file for the metafile of Figure 5 is provided in Figure 8. Once the Parametric Space object is constructed, it can set values for individual variables with the help of a DOE table or a random experiment generator through the `nextCase()` method.

The DOE class A DOE table is nothing more than a tabular representation of the variable settings for each experiment. Each row in the table corresponds to one experiment and each column corresponds to a variable. Typically, DOE tables for second order models have three levels or setting for each variable: the minimum, nominal and maximum value. These levels are represented by the integers -1, 0 and 1 so that the columns in the DOE table are uncorrelated. Each DOE object must include methods to normalize and denormalize variable values and to return the real values of each variable corresponding to a given row in the experiment table.

The random experiment class When DOE tables are not used for specifying the values of the design variables, random experiments may replace them. Random experiment objects include methods to return double precision floats and integers from a given random distribution to build the experiment table. The default random distribution is a uniform distribution bounded by the bounds of a given variable.

A pictorial representation of the virtual experimentation process with the OO library is provided in Figure 9. Each cylinder in the figure represents an object. The left side of the figure shows the OO wrapper objects needed to create multiple analysis files corresponding to a DOE table or a fixed number of random cases. The central part of Figure 9 shows the vehicle object being created for each experiment by a replacement of variables in the metafile from the parametric space object through the `replaceNamelist()` method. The right hand side of the Figure shows the individual AIM objects creating each analysis input file.

CONCLUSIONS

An Object Oriented vehicle library was developed to represent vehicle configurations in a generic manner. The library can be used to develop modules that create input files of analysis tools used in aircraft design. Each such module relies on the OO vehicle model for data and must include logic to create all necessary input in the correct format. Both the vehicle library and the input modules may be used to create a central preprocessor that can be used in an integration environment for design analysis. This integration is necessary for design processes such as metamodeling and uncertainty assessments. With the use of analysis input modules, integration of analysis tools within an integration environment becomes simpler.

Three input modules were created for the HASC, BDAP and FLOPS analysis tools. After the development of these modules, virtual experimentation can begin either within an integration environment or with the OO wrapper tools included in the library. The library was implemented successfully by the authors to develop metamodels included in parametric and probabilistic dynamic vehicle models used in the design of a representative 150 passenger transport [13] and to develop the aerodynamic simulation database for a representative supersonic business jet [14].

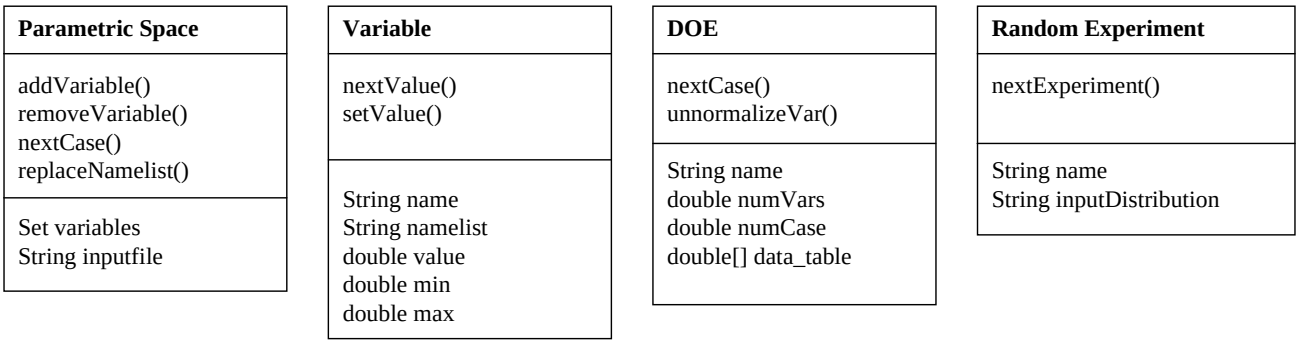

Figure 7: Classes developed for virtual experimentation

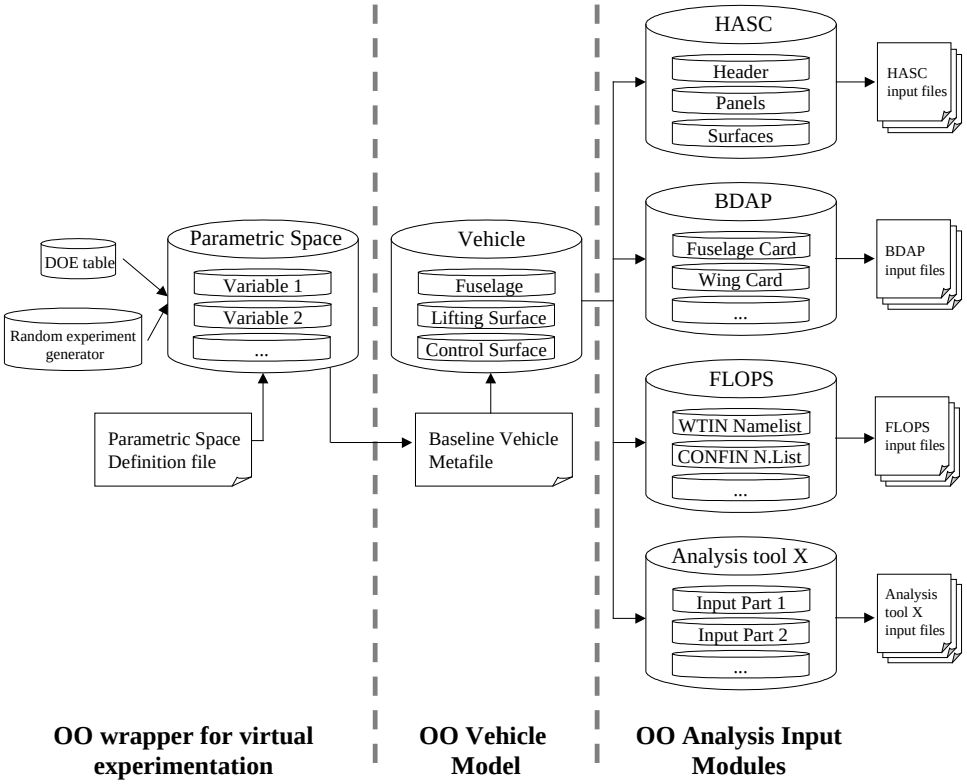

Figure 9: OO analysis input file generation

REFERENCES

[1] Dimitri N. Mavris, Daniel A. DeLaurentis, Mark A. Hale, and Jimmy C. Tai. Elements of an emerging Virtual Stochastic Life Cycle Design Environment. Number 1999-01-5638. AIAA/SAE, 1999.

[2] Daniel DeLaurentis, Dimitri N. Mavris, and Daniel P. Schrage. System synthesis in preliminary aircraft design using statistical methods. ICAS, 1996.

[3] Dimitri N. Mavris, Daniel A. DeLaurentis, Oliver Bandte, and Mark A. Hale. A stochastic approach to multi-disciplinary aircraft analysis and design. AIAA, 1998.

[4] George E. Box and Norman R. Draper. *Empirical model-building and response surfaces*. John Wiley & Sons, 1987.

[5] Susanne Weiss and Frank Thielecke. Aerodynamic model identification using local model networks. Number 2000-4098, pages 364–372. AIAA, 2000.

[6] Martin T. Hagan, Howard B. Demuth, and Mark Beale. *Neural Network Design*. PWS Publishing, 1996.

[7] Mark A. Hale. *An Open Computing Infrastructure that Facilitates Integrated Product and Process Development from a Decision-Based Perspective*. PhD thesis, Georgia Institute of Technology, July 1996.

[8] Alan E. Albright, Charles J. Dixon, and Martin C. Hegedus. Modification and validation of conceptual design aerodynamic prediction method HASC95 with VTXCHN. Technical report, NASA Contract Report 4712, 1996.

[9] Dennis Kafura. *Object-Oriented Software Design and Construction with JAVA*. Prentice Hall, 2000.

[10] Boeing. *Boeing Drag Analysis Program, BDAP*.

[11] Arnie McCullers. *Flight OPtimization System*. NASA Langley. version 5.94.

[12] W. Welch, R. Buck, J. Sacks, and H. Wynn. Screening, predicting, and computer experiments. *Technometrics*, 34(1), February 1992.

[13] Julien Scharl and Dimitri N. Mavris. Building parametric and probabilistic dynamic vehicle models using neural networks. Number 2001-4373. AIAA, AIAA, August 2001.

[14] Julien Scharl, Dimitri Mavris, and Ivan Y. Burdun. Use of flight simulation in early design: Formulation and application of the Virtual Testing and Evaluation Methodology. Number 2000-01-5590. SAE, 2000.