JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 7, March 2010

Air Vehicle Enviroment in C++: A Computational Design Environment for Conceptual Innovations

Maxwell Blair*
U.S. Air Force Research Laboratory

DOI: 10.2514/1.25880

Aerospace technology planners in the air force research laboratory are actively developing processes that produce a priority order for corporate investments. Technology assessment is one such process. Aerospace technology can be evaluated in the context of computationally intensive designs of innovative vehicle concepts. Computational design optimization traces technology performance metrics to system level design objectives and constraints. The air vehicle environment in C++ will support this process in the form of a pilot code currently under development. Design innovators will benefit from air vehicle environment in C++ at one of three levels. These three levels benefit a) the end user through interactive operations and file input/output, b) the object-oriented programmer through a compiled library of properly documented and inheritable objects, and c) the air vehicle environment in C++ developer who wishes to enhance air vehicle environment in C++ capability with modifications to the source code. The pilot environment described here focuses on parent–child relationships, automated dependency management, geometry, meshing, analysis, and eventually gradient-based optimization. All together, the overall capability leads to design variant management that will populate a database for either surrogate (response surface) modeling or high-level Pareto optimization. The proposed test case targets a suite of high-altitude-long-endurance aeroelastic concepts with geometric nonlinearity and follower forces.

I. Introduction

Air vehicle environment in C++ (AVEC) is a pilot effort for supporting aerospace concept designers who require computational physics to extend model fidelity. As envisioned, AVEC leads to a computational design optimization environment that spawns computational models at a rate associated with the speed of conceptual design. The environment objective is to rapidly converge on an optimal design with sufficient credibility to warrant an expensive test and thereby advance the technology readiness level (TRL).

This paper, written by an aerospace engineer, is reaching out to the computer programming community with software needs that support computational aerospace design. Innovative aerospace concepts will arise with innovative software programming. The desired environment will require an integration of engineering and computer science. This paper highlights desirable software features that address innovative and optimized aerospace design concepts.

The unified modeling language (UML) is a standard for developing and communicating software requirements for object-oriented projects. UML diagram types are developed in sequence, starting with simple user needs and ending in detailed requirements that object-oriented programmers can follow. Successful software development and maintenance depend on rigorously documented software requirements.

Received 15 June 2006; revision received 13 January 2010; accepted for publication 20 January 2010. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States. Copies of this paper may be made for personal or internal use, on condition that the copier pay the $10.00 per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923; include the code 1542-9423/08 $10.00 in correspondence with the CCC. This material is a work of the U.S. Government and is not subject to copyright protection in the United States.
[*] Research Aerospace Engineer, Air Vehicles Directorate, MultiDisciplinary Technology Center, AFRL/RBSD, 2130 Eighth St, maxwell.blair@wpafb.af.mil, Associate Fellow AIAA.

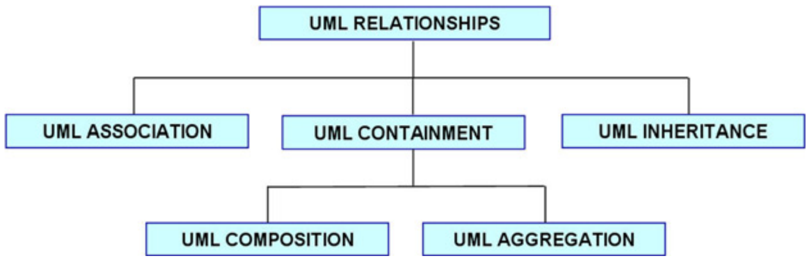


Fig. 1 Types of UML relationships.

Various UML relationships are discussed throughout. Figure 1 puts these relationships in perspective. These various forms of diagramming are relevant to the design of software that will generate a large number of optimized system variants representative of any number of permutations of technology choices.

A design environment is appreciated when it captures and optimizes design models of challenging concepts that serve operational needs. A computational design environment is appreciated in the following ways:

- *Data are timely*: Mundane operations are automated, thus the data process is accelerated and thereby freeing the designer to focus on intuitive aspects.
- *Data are relevant*: System performance is extracted from integrated models that address all physics with relevant fidelity and an appropriate level of uncertainty to account for different levels of geometric fidelity (e.g. a wing structure using intrinsic beam theory).
- *Data are reliable*: Data are persistent and processes gracefully recover from errors.
- *The number of design variables is not a constraint*: The ability to manage a very large family of designs significantly reduces uncertainty. The designer is constantly searching in the broadest and most comprehensive design space possible.
- *Designs are optimized*: Going the next step beyond design insight to include an automated search of parametric design space.
- *Discovery is achievable*: The choice and form of design variants is not restricted to historical precedent. Also, novel technologies are readily integrated.

Within the Air Force Research Laboratory (AFRL), the need for computational design arises from a requirement to assess and prioritize a portfolio of technology developments. AFRL maintains a number of air vehicle concepts that serve as integration concepts for any number of technologies. These flight concepts provide a context for both the technology developer and the technology investor. These concepts address a variety of missions that include long-range strike, space access, mobility, and others.

The AFRL MultiDisciplinary Science and Technology Center (MSTC) envisions a computational design modeling capability that supports the AFRL technology assessment process, beginning with war fighter needs as described in [1,2]. Reference [1] reviews the SensorCraft study that led to the joined-wing example case described in Sec. II. In recent years, quantitative technology assessment (QTA), as described in [2], has been in development at AFRL. QTA mirrors technology assessment with computer models. Reference [2] used ModelCenter as the integration medium between flight optimization system and high-altitude stability and control system, an internally developed data repository and JMP, a response surface generator[†]. The results have been viewed favorably with room for improvement. Reference [3] was funded by AFRL and describes a capability-focused technology assessment methodology for systems-of-systems.

Technology assessment processes at NASA are described in [4–6]. Reference [4] describes a technology engineering framework with coordinated systems and technology developments. This framework helps NASA prioritize technology developments in quantitative terms related to return-on-investment, benefit, critical need, probability, etc. Example applications include the High Speed Research (HSR) Program, Reusable Launch Vehicle (RLV) Program and subsonic concepts in the vehicle systems program. Reference [5] builds on [4] with a detailed description of the

[†] http://en.wikipedia.org/wiki/JMP_(statistical_software)

Exploration Systems Architecture Study (ESAS). As with AFRL, the high-level ESAS process is based on a suite of missions (International Space Station (ISS), lunar lander, Mars, etc.) and favors technologies that provide the greatest payoff. Reference [6] focuses on technology assessment in the context of NASA's Hypersonics Project. This paper reaches down to the level where computational design can be appreciated. Indeed, computational design is critical to their success. The environment for hypersonic design produces optimized configurations with acceptable system level risk. The tools range from geometric modeling to computational physics in the context of trajectory modeling.

References [7, 8] describe recent developments in the field of optimization that depend on the ability to generate a large number of computational design models. Reference [7] focuses on commercial development of communication satellite constellations. Reference [8] applies visualization techniques to wing design in terms of Pareto-optimized strategies that balance performance with cost. Reference [9] describes a computational propeller design with cascaded uncertainty.

This paper presents a number of tested software concepts in the form of a open and inheritable library of C++ class structures that lead to rigorous software requirements such as automated dependency management. This pilot code is preceded by a number of noteworthy developments that also feature automated dependency management. ICAD[‡] is described in [10] as a pioneer example of a proprietary object-oriented design framework based on a generative model that transforms input specifications into a product design through a number of relevant procedures. Two early efforts leading to computational design are described in [11,12]. Reference [11] synthesizes design models using a quasi-procedural method (in Fortran) that is roughly replicated in this work in the form of an adaptable dependency management class. Reference [12] prototypes dependency management in a framework for the optimization of an object-oriented conceptual design model. In more recent years, we see [12] leading to a capability for the optimization of complex systems described in [13]. The computational design capability described in [14] is a more tightly integrated environment with a focus on aerospace concepts. Reference [15] represents a significant computational design capability based on scripted freeware (free software) with application to the design of a hypersonic concept. The design model of [16] for the optimization of a high-altitude-long-endurance (HALE) concept with joined-wings was based on a commercially licensed scripted environment. This aeroelastically trimmed design model was complex with geometric nonlinear structures and follower forces. Certainly, a number of other computational design environments are also in development as they guide developmental decisions leading to future flight concepts. These environments serve the designer as an integration medium but with significant differences in their program structure and data process mechanisms. References [10–15] build on commercial or proprietary software libraries. The MSTC mission requires source code to meet research goals identified below.

Reference [17], provided an in-depth overview of the author's motivation for a computational design environment. HALE flight systems are inherently flexible and fly in extreme atmospheric conditions. Aeroelastic nonlinear mechanics have become a critical aspect of HALE design. Multifunctional antenna technology is pushing unconventional design configurations that appear promising. Successful configuration design of nonlinear aeroelastic systems with multifunctional technologies will benefit with the development of a computational design environment.

The history of computational design in the AFRL MSTC might begin with [18] where the requirements for an aerospace technology assessment system are put forth. This technology assessment environment requires the ability to measure technology benefit in terms of system performance. The recommended requirements focus on the pieces of the environment, but do not address specific processes leading to optimal system designs. One of these pieces is object-oriented programming with inherited dependency management in the form of dependency tracking and demand-driven calculations. Dependency management adds accountability to a QTA process with traceable metrics in the form of sensitivities and uncertainties. Class structures facilitate the integration of geometric modeling, mesh generation, mass modeling, and analysis all supporting configuration design optimization. Reference [19] represents a technical success for the MSTC. However, the MSTC business model for realizing the ATAS has not been particularly successful. The business model that led to [19] focused on fast commercial development while sidestepping established software standards and documentation. The research goals of the MSTC were not facilitated. In contrast, the nonproprietary AVEC software environment is attractive to a community of design engineers who benefit from software programming standards that address the capabilities outlined in [18].

[‡] http://en.wikipedia.org/wiki/ICAD

AVEC should be compatible with framework developments described in [20,21]. SORCER is described in [20] as a service-oriented grid space that locates published services (e.g. a structural analysis) and manages the process. FIPER in [21] exemplifies a best in class approach with providers supporting a service-oriented grid space. In contrast, AVEC will normally operate on a single platform and thereby provide a relatively simple environment for drafting simple design problems (as a front end for SORCER) such as are described in Sec. II. However, AVEC should also serve as a C++ client service under SORCER.

Industry designers are developing proprietary computational design capabilities. Examples were described in [15, 22]. Yet research opportunities are still ripe for the taking. AVEC is a pilot for a nonproprietary research tool that will lead to the desired computational design capability.

MSTC center research goals supported by AVEC: In the larger context of AFRL technology assessment, AVEC targets a computational design niche while addressing any number of innovative aerospace technologies. Example QTA studies might address adaptive systems, multifunctional structures, new material systems, etc. To meet the goals of the MSTC, the environment should also accommodate unfamiliar sciences, often nonlinear in nature. For example, the sciences associated with flapping wings are not well developed. Micro air vehicle designers will benefit with a programming environment that effectively integrates geometric design with complex vortical flows and nonlinear structures. The environment should generate and manage any number of locally optimal variants in support of design of experiments. New design variants will benefit with automated dependency management along with sensitivity and simple uncertainty tracing (no nonlinear crossterm interactions are possible).

AVEC is a library of classes that can be programmed to form an environment and therefore should be adaptable to different design strategies (such as a convergence on multifidelity geometry to be explained later). As the title implies, the author envisions using AVEC to support computational conceptual design innovations over a broad range of design space starting with simple geometric forms but with high fidelity physics. High fidelity physics can take the form of equivalent beam or shell theory for simple one- and two-dimensional geometric forms, for instance [23]. Geometric detail can be added as design solutions converge. As additional geometric detail is added, traditional finite element methods (e.g. a system of solid elements) are required to represent an exceptional number of design variables. AVEC will be designed to support the creation of a large database of locally optimized design variants (all with consistent geometric fidelity) that can be post-processed in terms of customer-driven objectives, and thereby sharing the QTA process with the stakeholders. Design space can be narrowed as customer interest warrants higher geometric fidelity.

II. SensorCraft Computational Design

SensorCraft is an AFRL integration concept that supports the larger technology assessment process. This section proposes a SensorCraft design optimization process with various UML diagrams starting with the use case diagram. A series of UML class diagrams lays the system out with some sense of association. To be sure, the associations described in these class diagrams provide some sense of organization. In AVEC, the design model follows a tightly managed dependency trail. The dependency trail is exemplified in Section II-D, ‘Process Representation’.

The AFRL SensorCraft concept is described in [24]. It is a conceptual flying antenna farm whose design intent is to replace several flight systems (currently in service) with a single integrated system. The AFRL Sensor-Craft technology-development program delivers next-generation ISR (intelligence, surveillance, reconnaissance) technologies in the context of system level integration of a HALE concept. SensorCraft also provides a context for development of design software technologies such as AVEC (Fig. 2).

A high-level computational design environment such as the envisioned AVEC system is required to manage, analyze, and optimize the myriad of disciplines, technologies, and design variants that play in the overall system

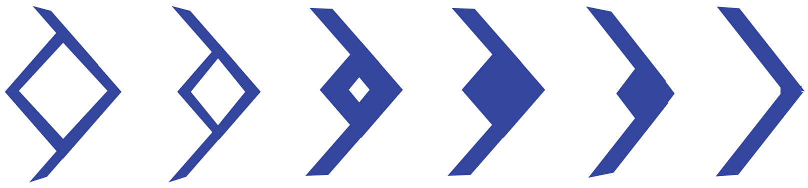


Fig. 2 Sample of SensorCraft design variants.

optimization of any SensorCraft concept. Figure 2 depicts a range of configuration variants that address the SensorCraft mission. At the highest level, two objective functions for SensorCraft are the ability to gather the most sensor data at the least cost. For this proposed study, sensor data are not addressed. The objective is the least fuel expenditure with a configuration design model that allocates space and mass to account for any number of technologies, including radar systems. Fuel expenditure will be significantly influenced by aerodynamic drag over a suite of missions. In conceptual design, aerodynamic drag is accounted for in terms of viscous drag and induced drag. Induced drag is a direct function of trim angle of attack which in-turn is a function of vehicle weight. The analysis models proposed for this study are far from state-of-the art in terms of viscous transonic flow and composite structural failure theory.

The unconventional joined-wing concept is an example in which a maximum endurance objective is constrained by global geometric nonlinearity that dominates the critical structural failure modes. Consequently, a structural weight penalty is normally associated with nonlinear mechanics. Rather than dismissing the joined-wing, we have elected to view nonlinearity as an opportunity to create a weight-competitive design. A thorough understanding of practical joined-wing design requires a computational design environment to drive a daunting certifiable design study without a major investment.

The baseline model described here, and the software classes that drive the design variant analyzes go far beyond traditional conceptual design methodology. In fact, the joined-wing design model is unprecedented and can NOT be addressed with historical regression, with or without linear analysis.

A. SensorCraft Optimization Use Case

A UML use case diagram documents a sequence of activities. Figure 3 outlines a high-level use case for SensorCraft configuration optimization featuring the equivalent static load method in [25].

In the case of SensorCraft design, the end user is tasked with populating and managing configuration design space with a large number of endurance optimal design variants. Subsequently, the end user is tasked with exploring the configuration design space and selecting the optimal configuration. In Fig. 3, the user starts with a baseline configuration that is analyzed and structurally optimized to meet the critical aeroelastic gust condition. Process loops in Fig. 3 are indicated with curved arrows. This optimized baseline is saved in a database. The baseline is subsequently

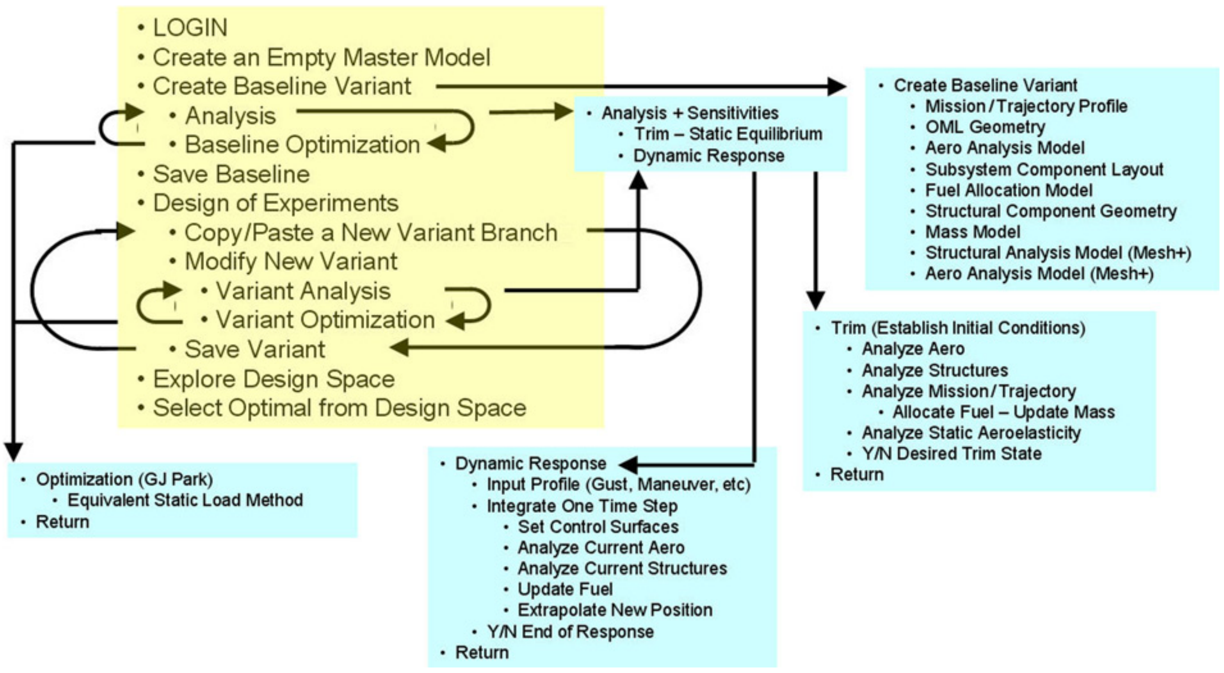

• LOGIN
• Create an Empty Master Model
• Create Baseline Variant
• Analysis
• Baseline Optimization
• Save Baseline
• Design of Experiments
• Copy/Paste a New Variant Branch
• Modify New Variant
• Variant Analysis
• Variant Optimization
• Save Variant
• Explore Design Space
• Select Optimal from Design Space
• Analysis + Sensitivities
• Trim – Static Equilibrium
• Dynamic Response
• Create Baseline Variant
• Mission/Trajectory Profile
• OML Geometry
• Aero Analysis Model
• Subsystem Component Layout
• Fuel Allocation Model
• Structural Component Geometry
• Mass Model
• Structural Analysis Model (Mesh+)
• Aero Analysis Model (Mesh+)
• Trim (Establish Initial Conditions)
• Analyze Aero
• Analyze Structures
• Analyze Mission/Trajectory
• Allocate Fuel – Update Mass
• Analyze Static Aeroelasticity
• Y/N Desired Trim State
• Return
• Optimization (GJ Park)
• Equivalent Static Load Method
• Return
• Dynamic Response
• Input Profile (Gust, Maneuver, etc)
• Integrate One Time Step
• Set Control Surfaces
• Analyze Current Aero
• Analyze Current Structures
• Update Fuel
• Extrapolate New Position
• Y/N End of Response
• Return

Fig. 3 Use case diagram for SensorCraft optimization.

copied and modified to create a new structurally optimized variant. The process is repeated until design space is sufficiently filled for the user to either create a Pareto front or to interpolate a multidimensional design surface.

B. Unique Design Requirement: Nonlinear Analysis

Any HALE concept is inherently large, light, and flexible. Three nonlinear contributors dominate HALE design: structural geometric nonlinearity, large deformations, and follower forces. Any span-braced (e.g. joined-wing, strut-braced wing, and tail-braced wing) configuration comes with compressive loads along unconventional paths. These compressive loads provide the possibility of static structural instabilities related to global buckling with an aeroelastic component. In [16], it is clear that nonlinear mechanics and follower forces contribute to the successful design of a joined-wing SensorCraft concept.

Figure 4 comes from [16] and is based on a built-up membrane finite element method (FEM) model. Of course no membrane is unsupported, thus precluding the possibility of local panel buckling in the nonlinear analysis. Figure 4 depicts the critical buckling mode for an optimized structure with trimmed aeroelastic follower forces. Note, although the aft wing is shown in the buckled state, it would be more accurate to say the leading edge of the aft wing is buckled. During the optimization process, forward wing buckling is also a possibility along the leading edge for upward loads and along the trailing edges for downward loads.

C. Managing Configuration Design Variants

It stands to reason that technology assessment requires a comprehensive computational design model of the entire system that addresses any permutation of a system of technologies. The goal for AVEC is to populate and manage a large database of optimized design variants that feed any number of design visualization tools such as a Pareto diagram as indicated in Fig. 9. A Pareto diagram can simply communicate the worth of a technology in terms of customer objectives.

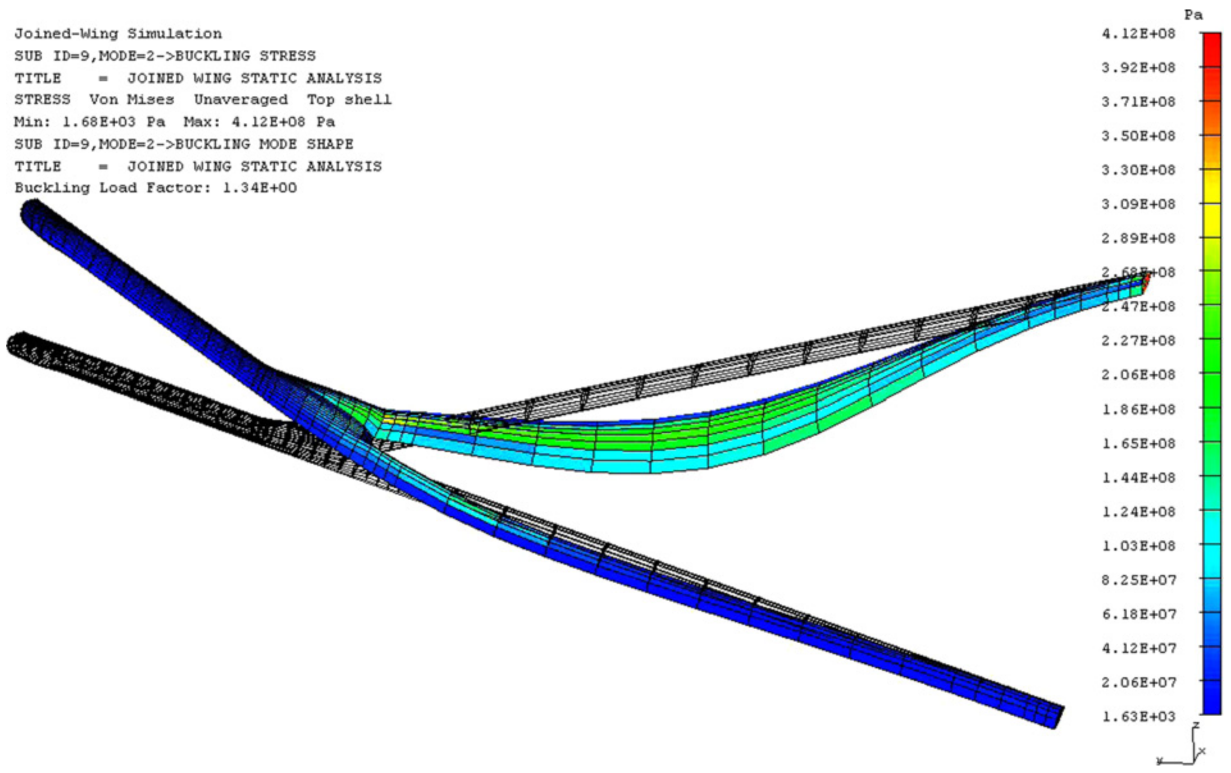

Joined-Wing Simulation
SUB ID=9,MODE=2->BUCKLING STRESS
TITLE = JOINED WING STATIC ANALYSIS
STRESS Von Mises Unaveraged Top shell
Min: 1.68E+03 Pa Max: 4.12E+08 Pa
SUB ID=9,MODE=2->BUCKLING MODE SHAPE
TITLE = JOINED WING STATIC ANALYSIS
Buckling Load Factor: 1.34E+00
Pa
4.12E+08
3.92E+08
3.71E+08
3.50E+08
3.30E+08
3.09E+08
2.89E+08
2.68E+08
2.47E+08
2.27E+08
2.06E+08
1.86E+08
1.65E+08
1.44E+08
1.24E+08
1.03E+08
8.25E+07
6.18E+07
4.12E+07
2.06E+07
1.63E+03
Z
Y
X

Fig. 4 Critical buckling of the fully stressed nonlinear FEM.

AFRL SensorCraft technologies are being evaluated (by major US airframe manufacturers) in the context of three basic configuration (or system) variants. These configuration studies are valuable in putting technology development into a system context. However, one is challenged to objectively identify which configuration is best suited for any combination of technologies. This requires a comprehensive study involving data from many assessments of many design variants driven by a myriad of variables—a computational design study. We still struggle to develop computational design as a means to optimize a family of configurations with nonlinear mechanics. This is the goal of the AVEC development that will focus on simple geometry driving high fidelity physics.

The joined-wing concept in Fig. 5 is characterized by the unusual load paths discussed above. With geometrically nonlinear mechanics and follower forces in play, a novel aeroelastic design optimization process must be made practical for a comprehensive computational design study. Although [16] was successful in completing a design optimization process for one configuration, the design convergence process was very cumbersome and not practical to address an entire family of designs. Reference [26] was successful in generating a response surface approximation for a family of 74 design configurations. Each of the 74 structurally optimized configurations was based on buckling criteria. However insufficient time and resources were available to closely examine each of the 74 designs. The emphasis was on identifying a process for optimizing the response surface approximation.

Design variable values can be managed in two ways, either parametrically or as discrete design variants. A variable value can be treated parametrically with automated master/slave dependency (discussed in detail later). With automated master/slave dependency, a trail of design sensitivities can be constructed to support gradient-based design optimization. At the end of the optimization process, the vector of variable values become a set of design variants that comprise a system variant. The system variant serves as a baseline model for launching a new system variant starting with a copy and paste operation on the baseline. Often, an optimized design is only a local optimal point, and not worth saving in any permanent fashion once a global optimal point has been achieved. New design variants can be initiated (and subsequently optimized) with the insertion of new technologies, as indicated in the discussion on technology assessment. Usually, this is not a simple variant in variable value, but involves all new geometric structures. It is important that the design environment and database work together in managing all kinds of design variants as a system of parametric (continuous and discrete) and boolean (on/off) variables.

A design variable will be wrapped in a class structure. The class data contain variable value, master/slave status, and variable sensitivity (derivatives). It is important that the design model be designed for computational efficiency, including the ability to switch sensitivity calculations on and off depending on whether simple baseline analysis (assessment) is in effect, or optimization is in effect. More will be discussed later.

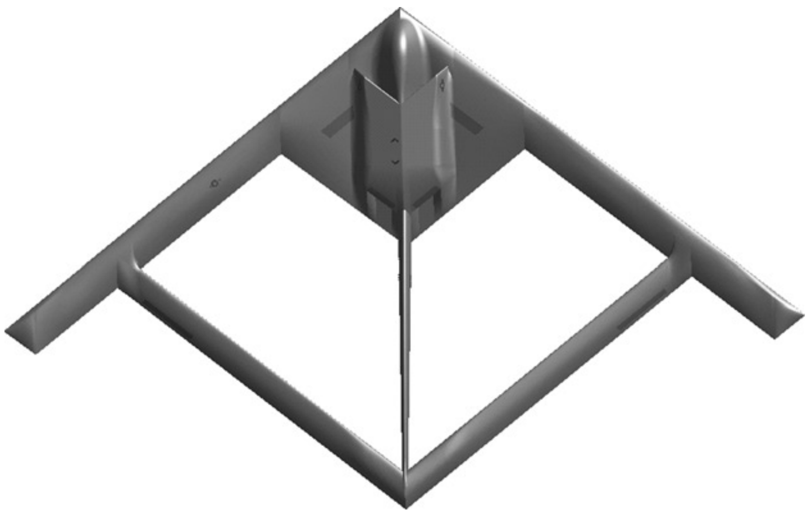


Fig. 5 Joined-wing.

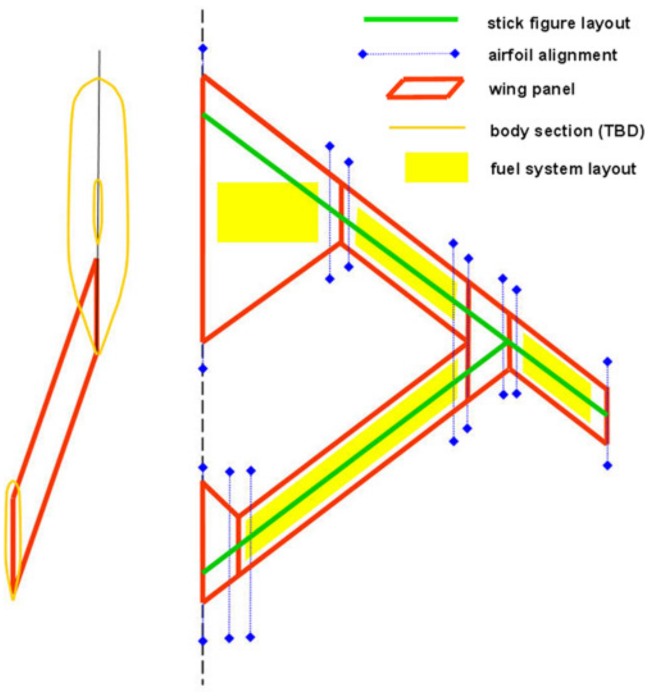


Fig. 6 Stick figure layout.

Figure 6 depicts the variables that control the SensorCraft joined-wing concept configuration. The green line segments determine the layout of the wing panels that are outlined in red. Wing depth is interpolated from a set of airfoils positioned as depicted with blue line segments. The fuel is managed between a number of fuel containers indicated with yellow. Figure 7 represents the structural analysis model in terms of equivalent beam structures in green. Cross-section geometry for each beam element is meshed. Internal geometry is parametrically driven with respect to aerodynamic airfoils interpolated along the span. The wing skin thickness is indicated in Fig. 8 could be enhanced to include more substructural detail. Geometric sensitivities will support gradient optimization.

Combined, Figs. 9–12 comprise a high-level UML class diagram. The idea is to focus on things and relations, not the process. These figures contain UML associations, as indicated in Fig. 1. The data contained in the diagram in Figs. 10–12 are processed according to subsection II-D, 'Process Representation'.

Figure 9 indicates a SensorCraft design environment communicating with a design variant database and a graphical user interface (GUI) all in support of surrogate modeling (e.g. response surface approximations) and data visualization (e.g. Pareto optimization).

- *System variant database*: Stores system variants in a convenient retrievable format. Specific database requirements have not been identified for AVEC. The following descriptions will identify types of data involved.
- *SensorCraft design environment*: Manages design analysis and optimization of a concept. Facilitates extraction of a system variant from the database, a user-defined component modification, a gradient-based optimization process, and insertion of the new optimized system variant into the database.
- The GUI will be discussed in Sec. IV in the context of the AVEC end user. The GUI is designed by the AVEC programmer to guide an end user in the creation of the system variant database.
- *Data visualization*: Commercial software is readily available to serve the need to examine a large set of system variants and thereby engage in technology assessment at a high level in collaboration with technology customers.

One of the challenges is to efficiently manage analysis data for a large number of design model variants. A design study must have an efficient strategy that spans the maximum design space with accurate physics in the context of a suite of missions. This process starts with simple physics that drive high fidelity mechanics. System variants

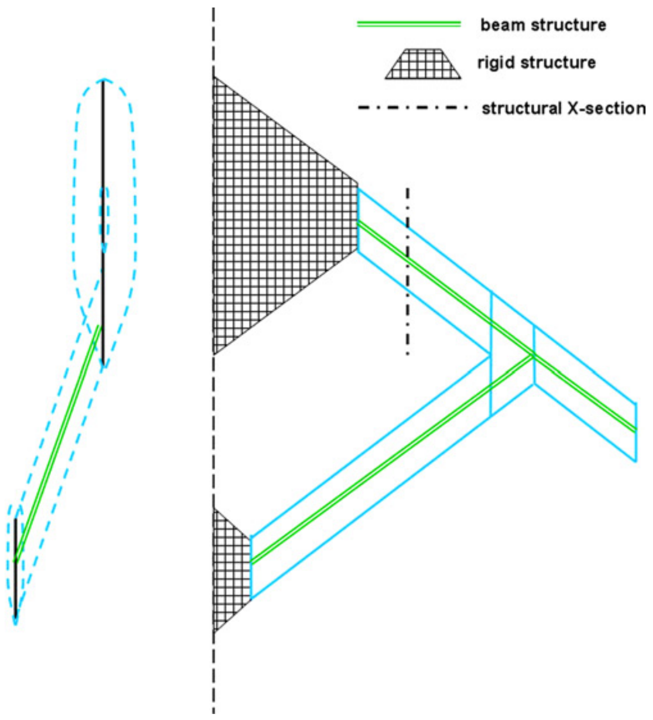


Fig. 7 Idealized equivalent beam structure.

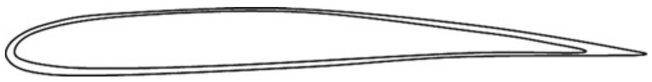


Fig. 8 Structural X-section.

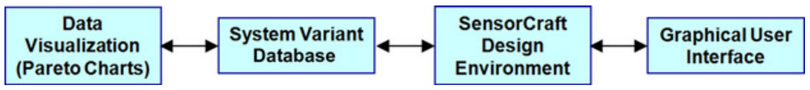

Data Visualization (Pareto Charts)
System Variant Database
SensorCraft Design Environment
Graphical User Interface

Fig. 9 Top-level class diagram for the AVEC design system.

should be grouped according to sensible collections of component variants. Each system variant, representative of a unique technology, is gradient-optimized, thereby relegating design variants in the near field, irrelevant. Still, it is not reasonable to analyze every permutation of all component variants and some design intuition should be exercised.

The SensorCraft design environment in Fig. 9 is expanded in Fig. 10. We see the SensorCraft design environment has two classes at the top level, Model Root and Mission Suite. Each mission in Mission Suite contains data related to trajectory and orientation. A yellow box contains a system variant that is stored in the design variant database in Fig. 9. A green box contains a particular set of variable variants that defines a system variant. System variants are stored in the design variant database in Fig. 9. The components of the class diagram of Fig. 10 are described as follows:

- *Model Root*: Model Root is a container for a system design variant comprised of configuration, optimization and analysis classes. Thus, when Model Root object is copied/pasted during runtime, all the data in Analysis and Configuration are included along with the connections in optimization. The connections are established in the baseline design variant as part of the class constructor for Analysis.

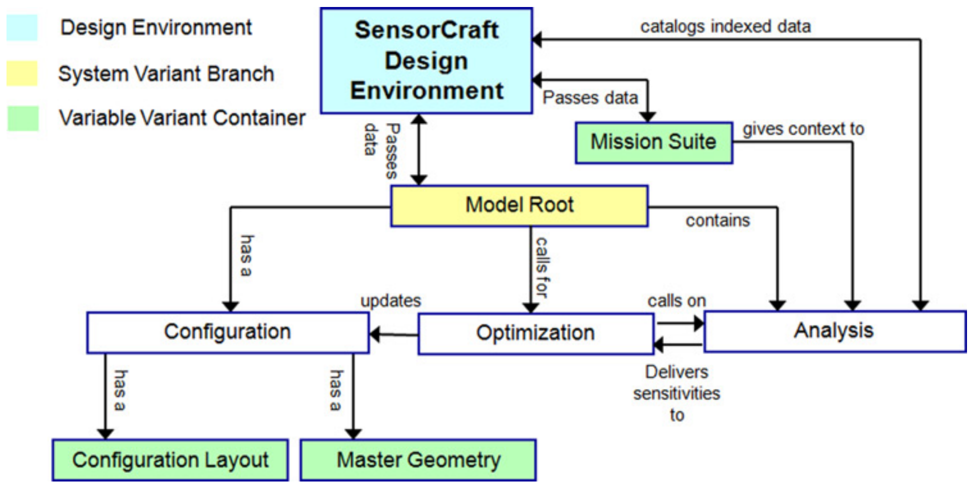


Fig. 10 Medium-level class diagram (see Fig. 12 for detailed breakouts).

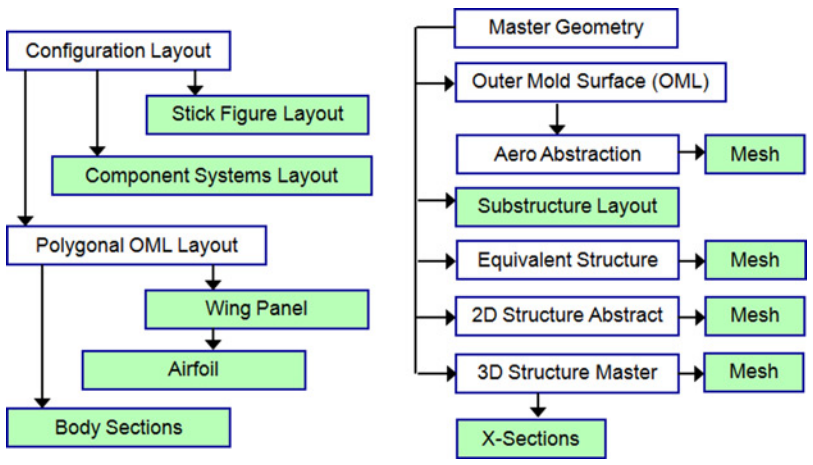


Fig. 11 Low-level class diagram for SensorCraft geometry (builds on Fig. 10).

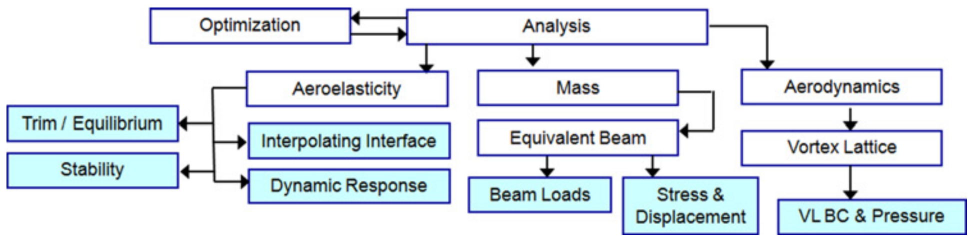


Fig. 12 UML low-level class diagram for SensorCraft analysis (builds on Fig. 10).

- *Mission Suite*: This is not a design variant itself. But it does contain a list of mission cases, each of which drives the analysis models in terms of maneuver loads, atmospheric conditions, etc.
- *Configuration_Layout*: Provides an abstract geometric rendering of the parametric state. Configuration_Layout is expanded in Fig. 11.

- *Master_Geometry*: Points to Configuration Layout and generates highest fidelity geometry required to achieve analysis and realize optimized design. Master_Geometry is expanded in Fig. 11.
- *Analysis*: Retains a pointer to Configuration for computational meshes etc. The pointer to Configuration is passed through Model_Root. Every instance of Analysis will always point to the same instance of Mission Suite. The analysis box is a UML container for every analysis model (input), a list of all analysis variables, pointers to any analysis package, and pointers to output files generated by analysis package. Analysis container provides value and sensitivity data for the optimization box. Figure 12 expands the analysis container with additional detail tailored to the SensorCraft application. Each analysis function contains input and output files and a pointer to executable analysis code.
- *Optimization*: This box is placed between 'Analysis' and 'Configuration'. The optimization box is a UML container for an input file in the standard form (objective, constraints, sensitivities), and a pointer to an optimization solver, roughly orchestrated according to the example use case in Fig. 3. Here, optimization is monolithic (not distributed) to begin. Intermediate design model data are disposed until convergence is reached. Optimization operates on design variable value sensitivity trails (generated in the Analysis container) performs gradient optimization and thereby converges to the optimal design model in an automated process. The purpose here might be to optimize each Pareto point in Fig. 9 so the customer does not become distracted with sub-optimal system variants. However, when global optimization is a challenge to reach, the purpose might be to optimize a limited set of design variables for a subsequent response surface optimization.

For SensorCraft configuration optimization, a number of sensitivity trails run through a number of analysis procedures, some of which are commercial packages and some of which are specialized. Each analysis requires a mix of design variables that does not line up (one-for-one) with the design variable boxes and does not necessarily line up (one-for-one) with any one system variant. For instance, if the design process depends on an established airfoil variant, the airfoil data are shared among many system variants. If parametric airfoil geometry is a variable in the optimization process, then the parametrics will drive a number of analysis packages, including aerodynamics and structures. In other words, the mix of design variants and design analysis models can become a challenge to manage without some form of automation proposed for AVEC.

A complete top-level analysis may be required for ANY combination of variants in the data tree. In addition, analysis class should be programmed so it can branch off at any level of design variance assuming the model is invariant at a higher level. For example, this could make sense if loads are fixed and we wish to study the composite ply layup sequence. Composite laminate layups could be smeared as parametric design variables as part of a gradient-based optimization process. Composite design space is characteristically bumpy with local minima. Thus, the designer will need to work on composite material design optimization as a side issue without creating entirely new system variants in the database Fig. 9. But, for this study, there is a single analysis container that manages analysis at the system level and interacts with the database through the design environment.

Although Analysis class will only examine (read-only) the contents of Configuration class, Optimization class will modify specific portions of Configuration, specifically, Layout and X_Sections classes indicated in green in Fig. 11. There is an assumption in play here: every instance of Model Root has a uniquely optimal design for a specific set of design variables including planform (with substructure layout) and cross-section geometry (airfoil shape and parametric structural thickness). The optimal can be achieved using gradient optimization applied to the baseline model, or alternatively applied to a surrogate (e.g. response surface approximations) representation or some combination of both. The set of all the locally optimized design variants are captured in a very large design-of-experiments study (which produces a surrogate model) from which the global optimal configuration is identified. If each system design variant (representative of a suite of technology interests) is globally optimized, then QTA can be achieved by the customer in a Pareto-optimal sense. In all cases, a design environment, such as AVEC, is required to spawn and manage a large number of system design variants.

Configuration in Fig. 10 contains Configuration Layout and Master Geometry with additional breakout details provided in Fig. 11. Green boxes contain a vector of variable variants that defines a system variant as was indicated in Fig. 10. Configuration Layout is decomposed into its class members, which are depicted in Fig. 6. (Each of these classes will inherit from display class as described in Sec. V for AVEC programmers.) The breakouts are described as follows:

- *Stick Figure Layout*: This class contains configuration parameters (sweep, span, etc.) that drive every analysis model. These high-level geometric variables are typically the basis for design space in a conceptual design study [27] based on regression models using historical data. This creates a meaningful baseline model from which to converge on a baseline optimized system variant in the database.
- *Component Systems Layout*: Contains geometric and mass attributes of all pieces that are not load-bearing structure in flight. Examples are fuel volumes, propulsion mass, radar systems, landing gear, etc. Wing Panel, Airfoil, and Body Sections must contain every member of Component Systems Layout.
- *Wing Panel*: Starting with Stick Figure Layout, trapezoidal panels are laid out and joined. Example attributes are chord and twist for every panel. Wing Panel will be used to position Airfoils that define and control the wing thickness shell.
- *Airfoil*: Contains geometric attributes related to wing thickness. Primary aerodynamic effects controlled with airfoil camber. Structural effects are realized with wing box depth. There is strong correlation between airfoil shape and drag (including the effect of structural weight on drag).
- *Body Sections*: Contains geometric attributes related to main body of the airplane. As with Airfoil, Body Sections control the fuselage cross-sections from which the OML is interpolated. Body Sections drives a number of analyzes, including aerodynamics and structures.

Configuration Layout controls Master Geometry from a high level. Master Geometry is the highest fidelity geometry from which geometric abstractions are derived in support of various computational analysis models. Master Geometry is decomposed in Fig. 11. The breakouts are described as follows:

- Outer Mold Surface uses the Configuration Layout in Fig. 11 to construct the highest fidelity representation of the exterior surface, also known as the outer mold line (OML).
- *Aero_Abstraction*: Based on the OML, generate watertight geometry is generated for the purpose of creating an aerodynamic mesh. Some idealization is added to avoid geometric boundary conditions that are either beyond analysis, meshing or design interest (e.g. flap gaps, thick trailing edges, etc.).
- *Aero_Abstraction/Mesh*: These attributes affect the aerodynamic model convergence with mesh refinement. For the vortex lattice model, mesh refinement is required to resolve spatial variations in the boundary conditions and the pressure distribution. For unsteady flow (flutter calculations) mesh refinement is also required to resolve wave motion in time.
- *Substructure_Layout*: Curves and lines are laid out on a plane (e.g. Wing Panel) to place wing substructure (i.e. ribs, spars) body substructure (i.e. bulkheads). These geometric attributes directly affect the structural analysis model in terms of load paths, and indirectly affect the induced drag due to structural mass.
- Equivalent_Structure class serves Analysis with a combination of geometric and parametric representations. Equivalent beam wing geometric properties (I, J, etc.) and design sensitivities are interpolated on the basis of X_Sections under 3D_Structure_Master. Equivalent_Structure becomes the analysis basis for the design-of-experiments study based on a large set of locally optimized configuration variants (described elsewhere). Each locally optimized configuration variant is optimized on the basis of equivalent structure optimization. Equivalent Structure can be either beam or plate idealization. In the case of SensorCraft, idealized beams are appropriate as depicted in Fig. 7 and [23].
- *Equivalent_Structure/Mesh*: These attributes affect the structural model convergence with mesh refinement. Mesh refinement is required to resolve synchronized vibration modes in time and space (short wave lengths with high frequencies).
- 2D_Structure_Abstraction (and its mesh) is not required for the conceptual structural optimization with equivalent beams. However, it does provide for analysis of 2D FEM build ups where higher fidelity analysis is a concern.
- *2D_Structure_Abstraction/Mesh*: Same requirements as Equivalent_Structure/Mesh with the addition of chord-wise structural deformations. The 2D_Structural_Abstraction is not used in the equivalent beam optimization.
- 3D_Structure_Master is the highest fidelity geometric representation of the airplane structure. The 3D Structure Master Mesh is not required for a fast-reaction conceptual structural optimization. But it does provide the option for high fidelity analysis of a optimal configuration variant.

- *3D_Structure_Master/Mesh*: Same requirements as 2D_Structure_Abstraction/Mesh with the addition of detailed structural failure mechanics. The 3D Structural model is not used in the equivalent beam optimization.
- X_Sections provide thickness to two-dimensional structural geometry and thereby serves as design control for 3D Structure Master class. X_Sections will be modified either with manual input or through Optimization (via Configuration). X_Sections class is the basis for interpolation for 3D_Structure_Master Geometry and for Equivalent_Structure (equivalent beam and equivalent plate). X_Sections also contains material properties. These geometric attributes directly affect the structural analysis and mass in terms of structural thickness.

Figure 12 provides a break out of Analysis Class as a part of the diagram in Fig. 10, in the context of design optimization. This diagram provides a hierarchy of analysis models in order that they can be found and retrieved. These components contain the data passed in the waterfall diagram in Fig. 13. However, the data in Table 1 are organized according to data flow and the data in Fig. 12 are organized according to data storage. For QTA, all design variants must be evaluated using identical criteria based on similar analysis requirements. Thus, each system variant will point to the same number and type of analysis models. In concept, the blue boxes in Fig. 12 roughly correlate with some of the data sets identified in Table 1 and Fig. 13.

- *Vortex_Lattice* (*VL*): Points to Aero_Abstraction/Mesh in Master_Geometry. Aerodynamic conditions are extracted from Mission Suite in Fig. 10. Generates data contained in class VL _BC & Pressure.

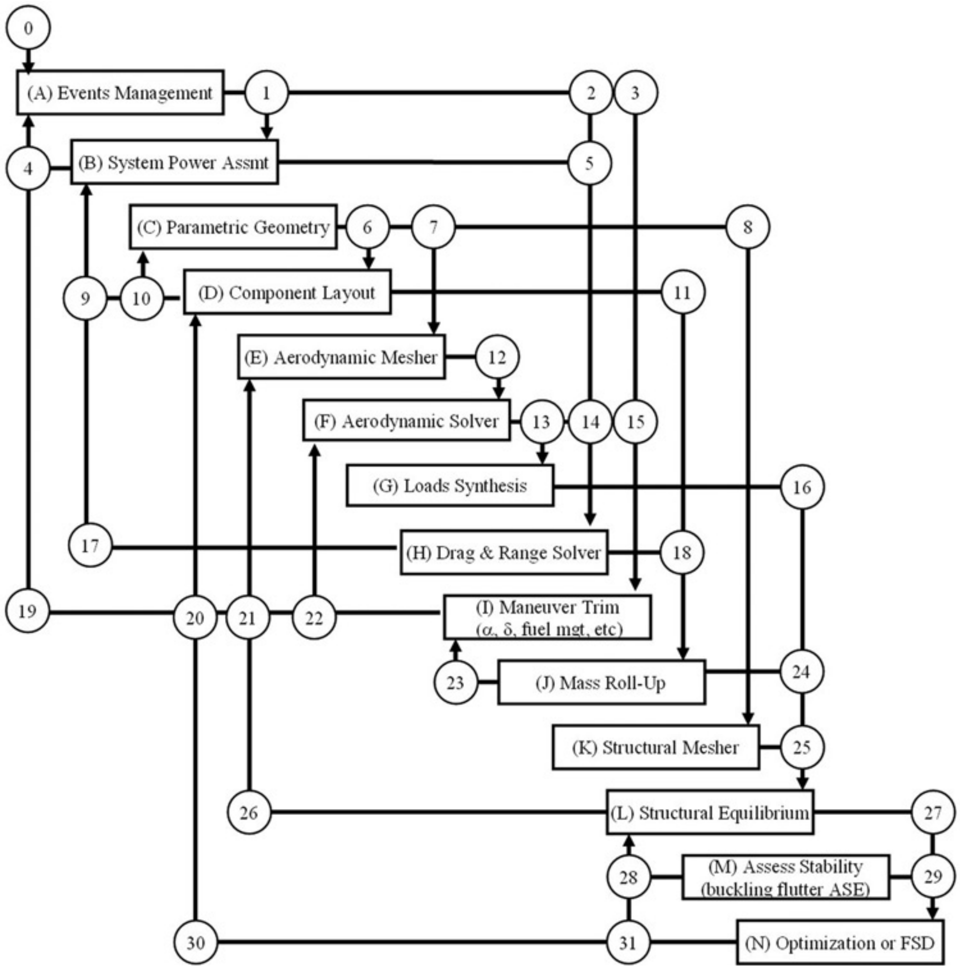


Fig. 13 UML association (N^2) diagram for the design procedure with indices in Table 1.

Table 1 Association indices for Fig. 13

1 Altitude, Mach	17 Fuel Consumed
2 Altitude, Mach	18 Trimmed Fuel Status
3 Altitude, Mach	19 Control Surface Settings
4 Required Thrust	20 Adjustable Mass Locations (e.g. fuel mgt)
5 Fuel Consumption Rate	21 Control Effecter Settings
6 Outer Mold Surface for Component Layout	22 Vehicle Attitude
7 Watertight Mold Surface for Aero Mesh	23 Total Vehicle Inertial Properties
8 Surface and Structure Geometry for Mesh	24 All Masses for Structural Analysis
9 Power System Position	25 Structural Computational Mesh
10 Component Geometry	26 Structural Deformations
11 Component Masses	27 Structural Element Stresses
12 Aerodynamic Computational Mesh	28 Eigenshapes
13 Pressure Distribution for Structural Integration	29 Sensitivities to Flutter, Buckling
14 Pressures for Drag and Range Calculations	30 Structural Element Masses
15 Aerodynamic Sensitivities	31 Structural Element Thickness
16 Aerodynamic Loads for Structural Analysis	

- *VL_BC & Pressure*: Serves as input and output to Vortex_Lattice class. Boundary condition (BC) input points to class Structure (via Aeroelasticity) for orientation and deformation and to Mission_Suite for atmospheric conditions. Pressure output provides loads to Beam_Loads, Stability Derivatives to Aeroelasticity, and (induced) drag sensitivities to class Optimization.
- Mass class accounts for all mass contributions, including structural mass. For the proposed SensorCraft study, structural mass optimization is based on class Equivalent_Beam.
- *Equivalent Beam*: Points to Equivalent_Structures for beam properties, serves Aeroelasticity with deformation. Contributes to mass build-up.
- *Beam_Loads*: Serves as input to Equivalent_Beam. Beam_Loads contain a pointer to Aerodynamics (via Aeroelasticity) for loads.
- *Stress & Displacement*: Serves as output from Equivalent_Beam. Stress sensitivities feed class Optimization in the form of constraints. Displacement output serves Aeroelasticity.
- *Aeroelasticity*: Class which integrates Vortex_Lattice and Equivalent_Beam models into a single aeroelastic analysis model. Addresses static and dynamic aeroelasticity in the form of equilibrium and stability analyzes and design sensitivities through member analysis containers.
- *Interpolating_Interface*: Interpolates Equivalent_Beam Displacement in order to provide boundary conditions for Vortex_Lattice. Also, Vortex_Lattice pressures are integrated and allocated in the form of loads for Equivalent_Beam. For linear analysis, the Interpolating_Interface is invariant with respect to deformation. For nonlinear analysis, large deformations require efficient and accurate algorithms with [23] being a prime example.
- Trim/Equilibrium class supports static analysis, primarily for fuel efficiency analysis, but also as initial conditions for dynamic response analysis.
- *Stability*: This class supports flutter analysis. Again, flutter analysis is optional where general dynamic analysis is addressed. However, flutter may become an active design constrain, in which case it is not optional. Active control must be a consideration, especially where gust load alleviation technology is required.
- *Dynamic Response*: This class is the most important analysis contributor to the critical gust response design criteria. This involves the integration of aerodynamics, structures and controls in an explicit time integration scheme, as either a perturbation about static equilibrium or with large displacements.

D. Process Representation

Table 1 and Fig. 13 work together to define a fairly comprehensive set of design dependencies that are automated to drive SensorCraft design optimization. Table 1 lists a number of indexed data types that are passed about as a design process iterates. These indices are encircled in Fig. 6 and make connections between output from one analysis module and input for another. For example (C) Parametric Geometry feeds (6) Outer Mold Surface for Component Layout which in turn is input for analysis module for (D) Component Layout. Waterfall diagrams such as this are a challenge to follow due to their inherent complexity. However, any design study that has not been traced with a flow diagram is probably not manageable. On the other hand, a waterfall diagram lacks significant detail which must be captured at the software documentation level.

The N^2 (N-squared) diagram in Fig. 13 works with Figs. 3 and 10 to describe an optimization process. Figure 13 describes the streams of dependencies alternating between data and analysis and ultimately defining the streams of automated design sensitivities that are to be managed. Figure 3 is the use-case or instruction set for converging on the optimal design. Figure 10 tells the programmer how the data are to be organized and located in a master-slave hierarchical form.

For instance, the mission class of [16] is represented in Fig. 13 as a combination of (A) Events Manager and (B) System Power Assessment. The output of block (A) is (1, 2, 3) Altitude and Mach which in turn feeds (B) Power Assessment, (H) Drag and Range Solver, and (I) Maneuver Trim. The output of block (B) is (5) which in turn feeds (H) Drag and Range Solver. Block A input includes (4) Required Thrust and (19) Control Surface Settings. Block B input includes (9) Power System Position and (17) Fuel Consumed.

Clearly, a mission class will be developed in AVEC to manage all aspects as described above. The mission class will be designed to interact with the air vehicle class. Actually, the initial mission class will be very simple to construct.

Where we have many design variants to be analyzed in one discipline such as structures, it does not make sense to instantiate a separate solver class for each design. However, it does make sense to instantiate a separate analysis model for each variant. Data from an analysis model will be sent to a common solver object that would manage several cases simultaneously. Thus, block L (Structural Equilibrium) in Fig. 13 could represent a structures model that will be placed in a common queue for solving structures equations. The management of the queue and the associated data will be a challenge to program. The first analysis class to be formulated will address aerospace structures.

Equivalent beams and plates require virtually no computational mesh. Reference [28] is an example of equivalent-plate modeling applied to a joined-wing concept. These "equivalence" methods are closely related to the P-version of finite element modeling. A structural designer working at the conceptual level might consider developing a C++ class for equivalent plate, or more generally, equivalent P elements.

A traditional FEM analysis employs a comprehensive mesh that must be regenerated for any geometric change. As an alternative to the comprehensive mesh, one could mesh parts independently. Interface elements are "sewn" together after-the-fact. This approach is reported in literature as interface elements.

Decisions have yet to take place on methods of meshing. Indeed, AVEC, when matured, may be a wonderful open environment for exploring the various issues involved with meshing such as unstructured solvers, convergence and interface elements. Meshing development is closely linked with geometry development. As indicated, serious geometry development will be enhanced when a geometry kernel is integrated with AVEC.

AVEC will ultimately address various optimization algorithms integrated with geometric nonlinearity, follower forces and aeroelastic trim. There is room to improve on the fully stressed optimization presented in [16]. One would avoid separate serial cyclical convergences procedures if the aerodynamic (loads and trim) model and nonlinear structural model were solved simultaneously.

E. The Management of Large Data sets

Computational design with many design variants requires many analysis models each with potentially large data sets of various sorts. A computational design environment must provide a practical database class to facilitate the various types and functional requirements. The optimal solution is not intuitively obvious.

The data models in AVEC are dependency tracked and therefore require some in-kind dependency-management that extends into the database. The data models in AVEC are hierarchical. This hierarchical form must be efficiently stored in the database, but does not prescribe the database format itself. AVEC classes are somewhat invariant in their form with a fixed number of variables. This feature will facilitate database development. For very large models with many design variants, the database class will benefit with ability to geographically distribute itself among a number of storage devices.

As an interesting note: Reference [29] contemplates the possibilities of managing large data models under XML. Although XML is not appropriate form for storing numerically intense data sets, it might be appropriate as an interface and exchange medium.

III. Overview of AVEC

The above description of a SensorCraft design application provides a motivation for development of AVEC. The UML functionality in Sec. II provides a plan for organizing and passing data in any object oriented environment. Automated dependency management is important to capture in a UML representation. This section provides a description of generic AVEC functionality, including automated dependency management.

Requirements were imposed at the start of the AVEC pilot development. First, the code is compiled. This means the data variables are strongly typed. This will facilitate (in some way) error management, scalability, and compatibility with database managers associated with optimization of computational physics. Second, the code compiles and executes on major operating systems including Unix, Linux, Mac, and Windows. Third, we require the compiled computational design capability to readily adapt to either master (client) or slave (server) status. These three requirements restrict AVEC to two choices of compiled source languages, Java and C++. ANSI C++ was chosen based on personal familiarity. However, the expectation is that the same functionality will also work in Java, assuming C++ pointers can find their equivalent replacement in Java. Aside from global declarations (unit conversions and file paths) all of AVEC falls under a root C++ class that can be instantiated and operated as part of other software developments.

As a side issue, scripted freeware languages (Python, Ruby) are gaining in popularity as the basis for managing design environments. This is discussed in [17]. I believe the choice between compiled and scripted languages will remain a struggle for years to come. Compromise solutions imbed scripted code in compiled systems or imbed compiled code in scripted systems.

The work presented here benefited tremendously with TrollTech QT GUI builder, which also serves as an interface with OpenGL graphical rendering. This is a multiplatform library of utilities that is available under rules for shareware or commercially for proprietary development. Both avenues were used in the AVEC project.

AVEC is an object-oriented environment with three levels of abstraction (i.e. levels of interaction). These are graphically depicted in Table 2.

Level I: End user: Most design engineers will understand AVEC as a graphically interactive design tool with save and retrieve functions in the form of XML file output and input. The end user will appreciate the ability to develop new parametrically-driven models from compiled classes. Additional details of end user functionality with AVEC are provided in Sec. IV of this report.

Level II: Programmer: Object-oriented programmers will immediately appreciate the ability to inherit classes from the AVEC library to create new classes. For instance, with some C++ knowledge, new geometric entities should

Table 2 Three levels of AVEC abstraction

Level	Abstraction	Value
I	*End user*: works interactively (no C++) • AVEC GUI: add, copy, modify, analyze • Link inter-object variables in dependency trail • Create/archive "super-classes" in XML format • Save and restore models with XML format	Tech assessment • Mission effectiveness • Materials evaluation • Concept optimization
II	*Programmer*: engineer proficient in standard C++ • Component class derivatives • Construct virtual functions • Leverage geometry kernel	Configuration innovations • Joined-wing • Micro air vehicles • Morphing air vehicles
III	*Software specialist*: expert in C++ • GUI: new interfaces • Develop new OpenGL features • Mesh and analysis classes • Optimization methods	Design methods research • Nonlinear physics • Coupled physics • Uncertainty management • Distributed design

be fairly painless to add. Indeed, AVEC anticipates the adoption of a standard geometry kernel in the not-too-distant future. This would extend the simple set of geometric entities (line, curve, surface, etc.) already available in AVEC pilot code. Additional details of programmer functionality with AVEC are provided in Sec. V of this report.

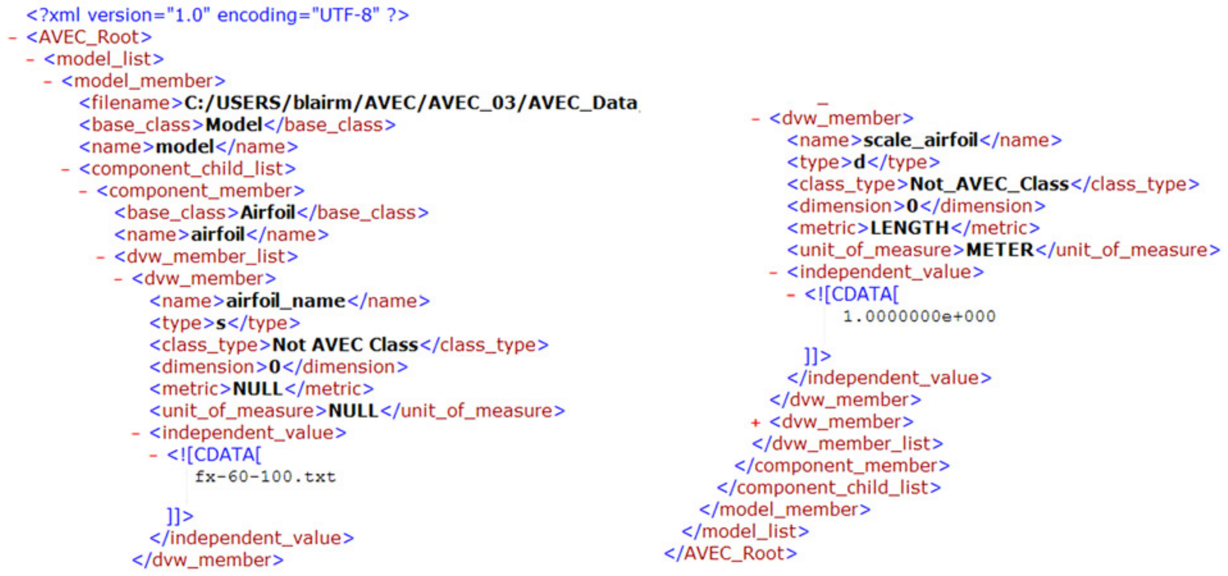
Level III: AVEC collaborator: AVEC collaborators in the form of hard-core C++ programmers may be interested to look into the AVEC source code and either make modifications to existing functionality or enhance current capability. For instance, AVEC is well suited for development of cascaded uncertainty (e.g. [9]) along with the current dependency-management system. From an engineering perspective, this could be applied in numerous situations. For example, this means the design might be extended beyond classical structural reliability-based design to include uncertain loads.

IV. End-User Functionality of AVEC (Level I)

This section describes some of the end user functionality of the GUI in Fig. 9. Alternatively, the end user can initiate a model with textual data in the XML format. The end user will interact with the AVEC graphical interface to develop models and families of models that automatically benefit from native AVEC functionality in the form of dependency-management. These models and family of models can be saved and retrieved in XML format. The expectation is that AVEC data that conforms to an XML schema can be viewed with an XML browser (Fig. 14).

Figure 15 highlights the various features of the main AVEC palette. The main viewing window contains graphical renderings. The FILE menu offers save and retrieve functions. This provides the ability to save interactive model developments on disk storage and retrieve them for later review and enhancements. The VIEW menu offers options for various AVEC viewing functions on the graphical rendering in the main viewing window. Viewing control is affected through three sliders bars. Thus, the sliders control viewing translation, rotation, zoom, and more. The product model tree and associated radio buttons (see Fig. 8) work in tandem. The product tree presents a design model in hierarchical (parent–child) form. The radio buttons provide the end user the ability to add, modify, render, delete, save, and retrieve components in the product tree. Textual input and output are contained in the bottom horizontal boxes.

In addition to the ability to save and retrieve entire models, the end user is able to save composite classes (partial models) that can be reused along with any other AVEC native classes. For instance, the end user can interactively construct and save an airfoil class that can be interactively retrieved as part of user defined wing and tail classes. Data



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <AVEC_Root>
  - <model_list>
    - <model_member>
        <filename>C:/USERS/blairm/AVEC/AVEC_03/AVEC_Data
        <base_class>Model</base_class>
        <name>model</name>
      - <component_child_list>
        - <component_member>
            <base_class>Airfoil</base_class>
            <name>airfoil</name>
          - <dvw_member_list>
            - <dvw_member>
                <name>airfoil_name</name>
                <type>s</type>
                <class_type>Not AVEC Class</class_type>
                <dimension>0</dimension>
                <metric>NULL</metric>
                <unit_of_measure>NULL</unit_of_measure>
              - <independent_value>
                - <![CDATA[
                    fx-60-100.txt

                  ]]>
                </independent_value>
              </dvw_member>
            - <dvw_member>
                <name>scale_airfoil</name>
                <type>d</type>
                <class_type>Not_AVEC_Class</class_type>
                <dimension>0</dimension>
                <metric>LENGTH</metric>
                <unit_of_measure>METER</unit_of_measure>
              - <independent_value>
                - <![CDATA[
                    1.0000000e+000

                  ]]>
                </independent_value>
              </dvw_member>
            + <dvw_member>
            </dvw_member_list>
          </component_member>
        </component_child_list>
      </model_member>
    </model_list>
  </AVEC_Root>
```

Fig. 14 Sample XML data format (two columns).

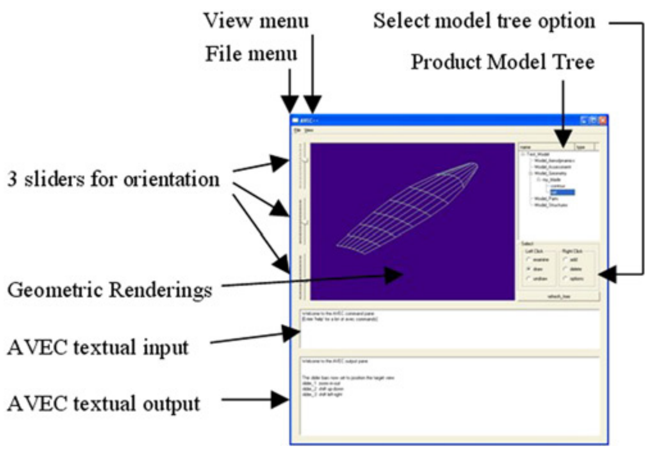
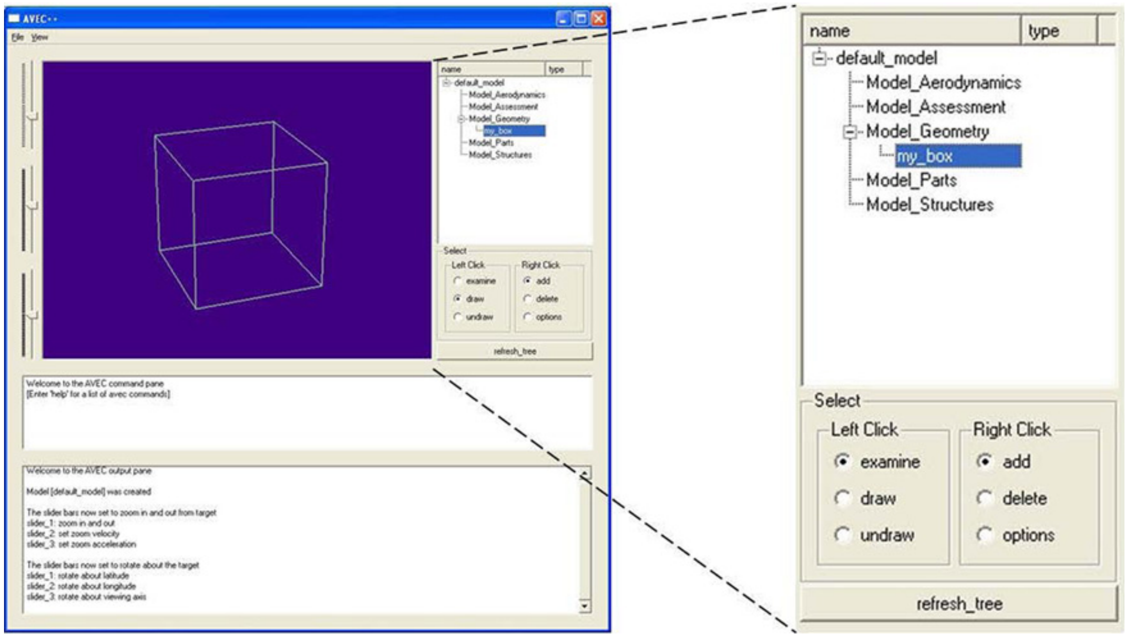


Fig. 15 GUI-view of AVEC for interactive end user—main pallette.

are saved in XML format, the same format as used to save an AVEC model. However, variable values are not saved for derived classes. A number of derived classes will be part of the distribution following the pilot code development.

Figure 15 depicts a partial graphical view of the AVEC desktop as implemented today. For the SensorCraft application, the Product Model Tree will reflect the objects listed in Figs. 10–12. The graphically rendered Micro Wing in Fig. 15 is displayed as part of the derived Blade class and children. The instantiated object tree is depicted to the right of the graphical rendering. Below the object tree is a palette of options that control the action upon selecting any one object listed in the object tree. The textual boxes at the bottom facilitate model modifications with text input and output (I/O). An AVEC help utility is suitable for such user-driven I/O.

Figure 16 expands the Product Model Tree in Fig. 15 to make the Model Tree Options clear. Three radio buttons are positioned under both the “Left Click” and “Right Click” headings. Under “Left Click”, we have button options for “examine”, “draw”, and “undraw”. Under “Right Click”, we have button options for “add”, “delete”, and “options”



```
Welcome to the AVEC command pane
[Enter 'help' for a list of avec commands]
```

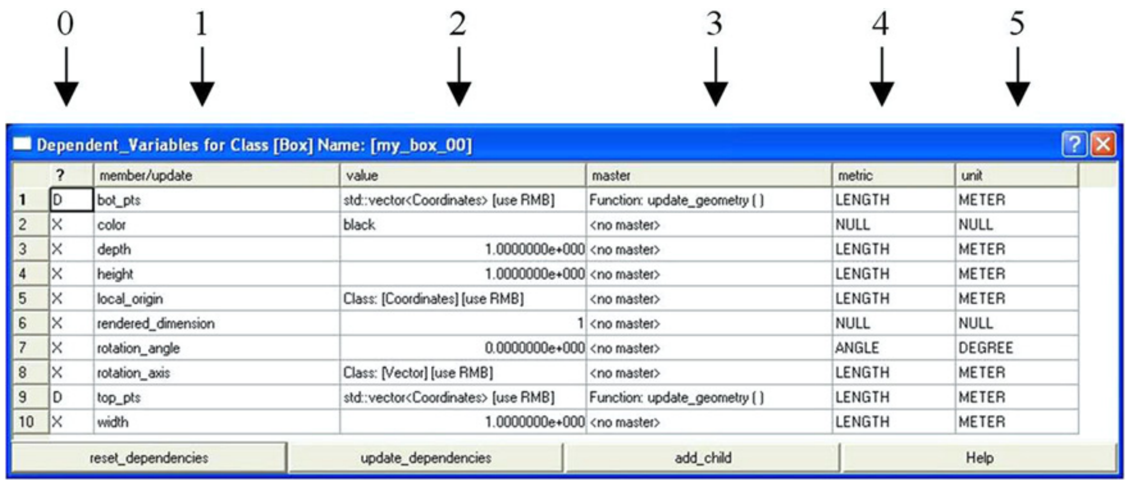
```
Welcome to the AVEC output pane

Model [default_model] was created

The slider bars now set to zoom in and out from target
slider_1: zoom in and out
slider_2: set zoom velocity
slider_3: set zoom acceleration

The slider bars now set to rotate about the target
slider_1: rotate about latitude
slider_2: rotate about longitude
slider_3: rotate about viewing axis
```

Fig. 16 GUI-view of AVEC for interactive end user—examine class.

(*note*: AVEC will be enhanced with a radio button labeled “select” as a way to choose from a set of variable variants (the green boxes in Figs. 10 and 11) that specify any particular system variant (yellow box in Fig. 10). Only one radio button can be selected under “Left Click”. With the “examine” radio button selected, the left mouse button (LMB) can be used to select any Component class in the Product Model Tree and the corresponding Component Examination Pane will pop up. For example, an instance of Component Examination Pane is displayed in Fig. 17 for the instance of “my_box” listed in Fig. 16. Instance “my_box” is of class “Box_Class” which inherits from class “Component”. The Component Examination Pane is described in more detail below. Selecting “draw” under “Left Click” in Fig. 16 causes “my_box” geometry to be displayed when “my_box” is selected with the right mouse button (RMB) as indicated in the graphic rendering pane on the left side of Fig. 16. Selecting “delete” and selecting “my_box” with the RMB causes the instance of my_box to be deleted. The “add” option starts a process for adding another derived instance of class Component as a child of “my_box”. The “options” option results in a pop-up with a number of options displayed such as “save derived class”.

The widget (table) in Fig. 17 lists all the dependent variables contained in an instantiated Box object (Box class inherits from Component class). For instance, variable values, “height”, “width”, and “depth” control the box linear dimension and “rendered_dimension” controls the graphical rendering with 0 (points), 1 (wire frame), and 2 (surface) dimensions. This widget is displayed when the examine button (see Fig. 9) is selected and an item (my_box) in the object tree is selected. Columns in Fig. 17 are labeled from 0–5. Column 0 in the table widget identifies the status (D for dependent and valid, X for independent, N for dependent and not-valid). Column 1 lists the variable names. Column 2 displays the data value for simple variables types (double, integer, etc.). Data value can be modified directly in column 2 by selecting with the LMB and simply typing in a new value. For special data types (e.g. large vectors and arrays), a right-click on any box in column 2 brings up a dialogue box with all values listed. If the variable status listed in column 0 is X (Independent), then the list of values can be modified, appended or deleted. Column 3 is dedicated to identifying and modifying dependency on a master variable or just identifying its master function. (Variables and functions are wrapped independently). Column 4 is static and identifies the metric type. Column 5 is variable and identifies the unit of measure. The user can select cells with the right-mouse-button in this column and subsequently select from a list of available units of measure.

An example process is described for instantiating, modifying and saving Box class displayed in Fig. 16. Begin with FILE/New default model. This brings up a default model tree as depicted in Fig. 16. With “add” radio button selected, use RMB to select “Model_Geometry”. A pop-up dialogue box appears and the user can select “Box” class. Another pop-up dialogue box appears and the user can name this instance of Box, “my_box”. Next, refresh the Product Model Tree with the “refresh tree” button. The instance of “my_box” will now appear in the Product Model Tree. Next, to display the box, select “draw” option and select “my_box” with LMB. To examine the variables that control



Dependent_Variables for Class [Box] Name: [my_box_00]

	?	member/update	value	master	metric	unit
1	D	bot_pts	std::vector<Coordinates> [use RMB]	Function: update_geometry ( )	LENGTH	METER
2	X	color	black	<no master>	NULL	NULL
3	X	depth	1.0000000e+000	<no master>	LENGTH	METER
4	X	height	1.0000000e+000	<no master>	LENGTH	METER
5	X	local_origin	Class: [Coordinates] [use RMB]	<no master>	LENGTH	METER
6	X	rendered_dimension	1	<no master>	NULL	NULL
7	X	rotation_angle	0.0000000e+000	<no master>	ANGLE	DEGREE
8	X	rotation_axis	Class: [Vector] [use RMB]	<no master>	LENGTH	METER
9	D	top_pts	std::vector<Coordinates> [use RMB]	Function: update_geometry ( )	LENGTH	METER
10	X	width	1.0000000e+000	<no master>	LENGTH	METER

Fig. 17 Examine component pane.

"my_box", select "examine" option and select "my_box" with LMB. The Component Examination Pane of Fig. 16 will pop up. Double click on column 2 (value) for variable "height" with LMB. Type in a new value (a character string). Next select column 0 for height to translate the character string into a numerical value. The box graphic disappears because it is not valid, based on the new height. Select "draw" option and LMB on "my_box" to redraw my_box with a new height in the display. Select FILE/Save_Model_As and enter a file name. Next select FILE/Close_Model. The Product Model Tree will become blank. The model can be retrieved with FILE/Retrieve_Model.

The end user can interactively change any value listed in the Component Examination Pane.

Figure 18 depicts the process for creating a derived class from a number of compiled AVEC classes. Derived classes are saved in XML format and retrieved. Variable values are not saved in derived classes. The following items (a)–(e) describe icons (a)–(e) in Fig. 18.

a) Begin with clean AVEC palette. From the drop-down file menu select: New_Default_Model
b) Select Add in radio buttons and select Model_Geometry in Product Model Tree
c) Select class airfoil with name: "airfoil"
d) Enter an object name (e.g. "my_airfoil")
e) Pick on "refresh tree" Product Model Tree will be expanded with "my_airfoil".

Figure 19 depicts the process for adding component children. The following items (f)–(j) describe icons (f)–(j) in Fig. 19.

f) Select 'Add' for "my_airfoil" in Product Model Tree

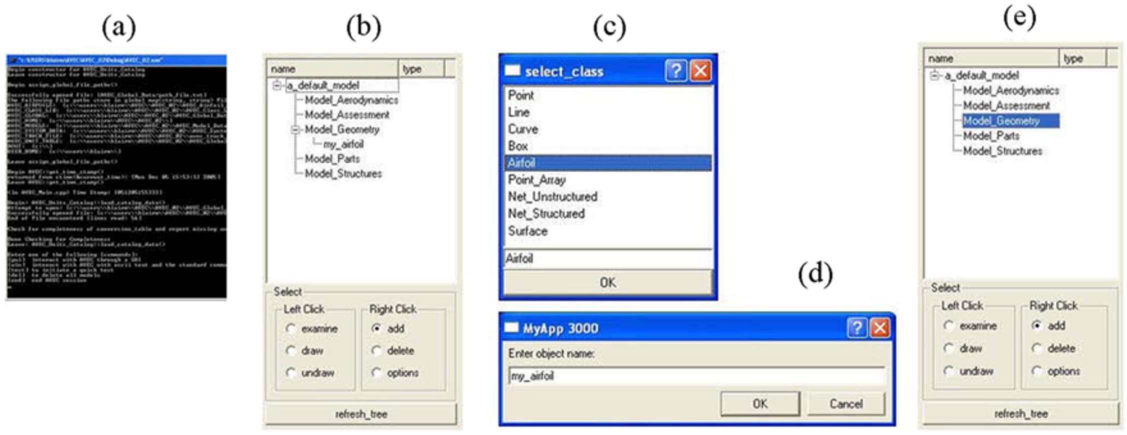


Fig. 18 Derived airfoil class.

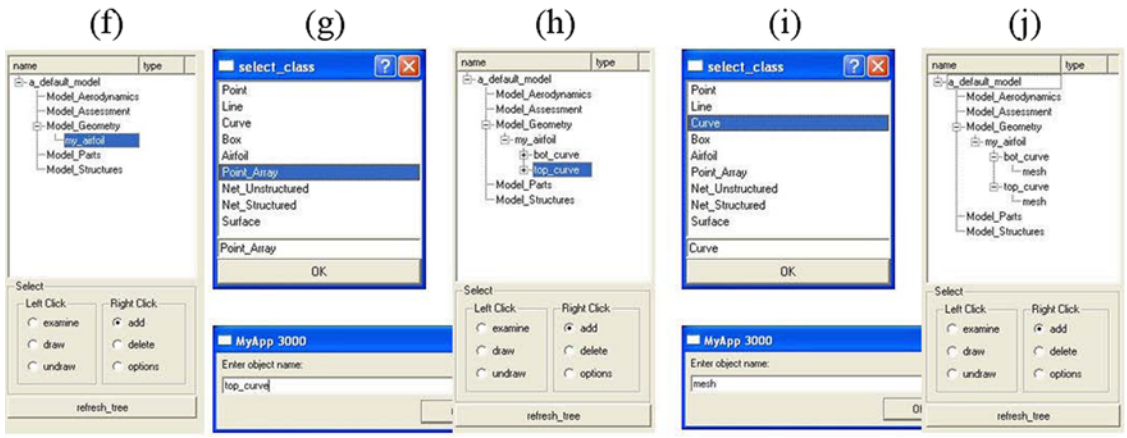


Fig. 19 Derived airfoil class—adding children.

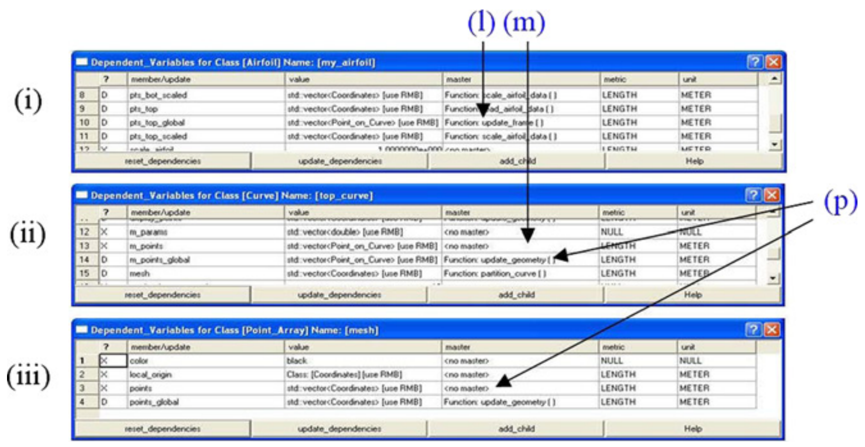
g) Select class “Curve” with name: “top_curve”. Refresh and expand Product Model Tree. Repeat (f) and (g) for “bot_curve”.
h) Select ‘Add’ for “top_curve” in Product Model Tree
i) Select class “Point_Array” with name: “mesh”. Refresh and expand Product Model Tree
j) Repeat (h) and (i) for “bot_curve”

Figure 20 depicts the interactive process for establishing master/slave dependencies. The following items (k)–(r) are depicted in Fig. 20.

k) Select “examine” button for the following Components: (i) “my_airfoil”, (ii) “top_curve”, (iii) “mesh”.
l) [LMB] on column: “master” in row: “my_airfoil/pts_top_global”
m) [RMB] on column: “master” for row: “top_curve/m_points”
n) “top_curve” is now slaved to “airfoil”
o) Repeat process for “bot_curve”
p) Next link “m_points_global” (for “top_curve”) with “points (for “mesh”)
q) “mesh” is now slaved to “top_curve”
r) Update (column 0) “mesh/points”.

Figure 21 depicts the interactive model save and retrieve process. Portions of the following items (s)–(w) are depicted in Fig. 21.

s) Select “draw” and “mesh”



Dependent_Variables for Class [Airfoil] Name: [my_airfoil]

	?	member/update	value	master	metric	unit
8	D	pts_bot_scaled	std::vector<Coordinates> [use RMB]	Function: scale_airfoil_data ( )	LENGTH	METER
9	D	pts_top	std::vector<Coordinates> [use RMB]	Function: load_airfoil_data ( )	LENGTH	METER
10	D	pts_top_global	std::vector<Point_on_Curve> [use RMB]	Function: update_frame ( )	LENGTH	METER
11	D	pts_top_scaled	std::vector<Coordinates> [use RMB]	Function: scale_airfoil_data ( )	LENGTH	METER
12	X	scale_airfoil	1.0000000e+000	<no master>	LENGTH	METER

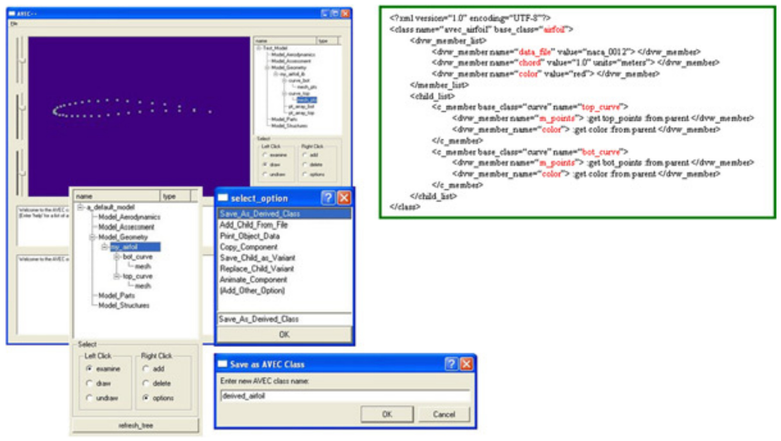
Dependent_Variables for Class [Curve] Name: [top_curve]

	?	member/update	value	master	metric	unit
12	X	m_params	std::vector<double> [use RMB]	<no master>	NULL	NULL
13	X	m_points	std::vector<Point_on_Curve> [use RMB]	<no master>	LENGTH	METER
14	D	m_points_global	std::vector<Point_on_Curve> [use RMB]	Function: update_geometry ( )	LENGTH	METER
15	D	mesh	std::vector<Coordinates> [use RMB]	Function: partition_curve ( )	LENGTH	METER

Dependent_Variables for Class [Point_Array] Name: [mesh]

	?	member/update	value	master	metric	unit
1	X	color	black	<no master>	NULL	NULL
2	X	local_origin	Class: [Coordinates] [use RMB]	<no master>	LENGTH	METER
3	X	points	std::vector<Coordinates> [use RMB]	<no master>	LENGTH	METER
4	D	points_global	std::vector<Coordinates> [use RMB]	Function: update_geometry ( )	LENGTH	METER

Fig. 20 Derived airfoil class—establish master/slave dependencies.



```
<?xml version="1.0" encoding="UTF-8"?>
<class name="avec_airfoil" base_class="airfoil">
    <dvw_member_list>
        <dvw_member name="data_file" value="naca_0012"> </dvw_member>
        <dvw_member name="chord" value="1.0" units="meters"> </dvw_member>
        <dvw_member name="color" value="red"> </dvw_member>
    </member_list>
    <child_list>
        <c_member base_class="curve" name="top_curve">
            <dvw_member name="m_points"> :get top_points :from parent </dvw_member>
            <dvw_member_name="color"> :get color :from parent </dvw_member>
        </c_member>
        <c_member base_class="curve" name="bot_curve">
            <dvw_member name="m_points"> :get bot_points :from parent </dvw_member>
            <dvw_member_name="color"> :get color :from parent </dvw_member>
        </c_member>
    </child_list>
</class>
```

Fig. 21 Derived airfoil class—save and retrieve.

t) Select “option” and “my_airfoil”. Pick: “Save_as_Derived_Class”. An XML file will be generated and saved with name “derived_airfoil” appended with a time-stamp (the XML file depicted is abridged)
u) Select “FILE/Delete_Model”
v) Select “FILE/New_Default_Model”
w) Select “option” and “Add_Derived_Class_as_Child”. Enter “derived_airfoil” as the derived class name

V. Software/Programmer Functionality of AVEC (Level II)

A programmer’s description of AVEC best begins with UML formats. Traditional UML methodology leads to detailed class and object diagrams. UML class diagrams can be generated from a clean sheet or can be reverse-engineered (i.e. extracted) from source code. Numerous codes are available to reverse engineer class inheritance from source code.

UML also addresses object diagrams of instantiated classes. UML object diagrams are appropriate to capture AVEC applications, such as the SensorCraft application described in Sec. II. Again, UML object diagrams can be generated from a clean sheet. The simplified object diagram in Fig. 23 is an example of clean sheet approach. A UML object diagram could conceivably be extracted from a combination of source code and a data model that is designed to support UML object diagrams. However, the AVEC environment is a challenge to capture with UML object diagrams because child objects of various types are instantiated at run time and stored in generic child lists. The good news is that AVEC restores object-oriented models solely based on data stored in XML. Thus, it is conceivable that a UML object diagram could be extracted from a combination of AVEC source code and an AVEC-generated XML data file. Intra-class data dependency could be extracted from AVEC source code. AVEC’s XML schema includes a trail of inter-object data dependency which would support a UML object diagram.

XML is appropriate to save models and derived classes. Saved models contain instantiated data. Derived classes do not contain instantiated data. Example XML data format for model save and retrieve operations was graphically depicted in Fig. 14.

Data variables and functions in AVEC are wrapped independently. Data variables are wrapped in Dependent_Variable_Wrapper described later in the context of dependency management in Sec. V.C. Variable values are saved for independent data variables only. Slaved values point to their master variable.

End users might avoid this section with details of object-oriented source-code. However C++ and Java programmers will be familiar with the concepts used in this section. The information presented here is in the format of a tutorial. In addition, some effort has gone into on-line documentation[§] of AVEC. This interactive programmer’s document (in HTML) is used as a reference manual.

A. Inheritance and Containment in AVEC::Component

This section describes two UML relationships listed in Fig. 1: Inheritance and Containment. The data for the end user airfoil example, beginning with Fig. 18, is organized according to UML diagrams in Figs. 22 and 23.

Component class is depicted in Fig. 22. A set of components can be organized and managed in a categorical hierarchy of parent–child relationships, depicted in the Product Model Tree GUI in Figs. 15 and 16. Note: parent–child relationships are a UML aggregation in the context of Fig. 1 and do not drive master-slave dependencies described below.

The relationship between Component and Dependency_Manager classes in Fig. 22 is an example of UML class inheritance in the context of Fig. 1. Thus the member functions and variables in Dependency_Manager are automatically part of the data structure in the class, Component, and its derivatives. Component derivatives also inherit the ability to relate to other instantiated Components in parent–child relationships. Technical details related to class Dependency_Manager are described at the class level in Sec. V.C.

Component_Children, in Fig. 22, is embedded in Component as a vector of C++ pointers to objects of Component class. In UML, this vector is a Composition. When container class Component is deleted, the contents of Component_Children are also deleted. The Component destructor is designed to avoid dangling pointers and memory leaks (i.e. data are rendered inaccessible for lack of a memory address).

[§] DOxygen is downloadable freeware www.doxygen.org for automated on-line documentation generation.

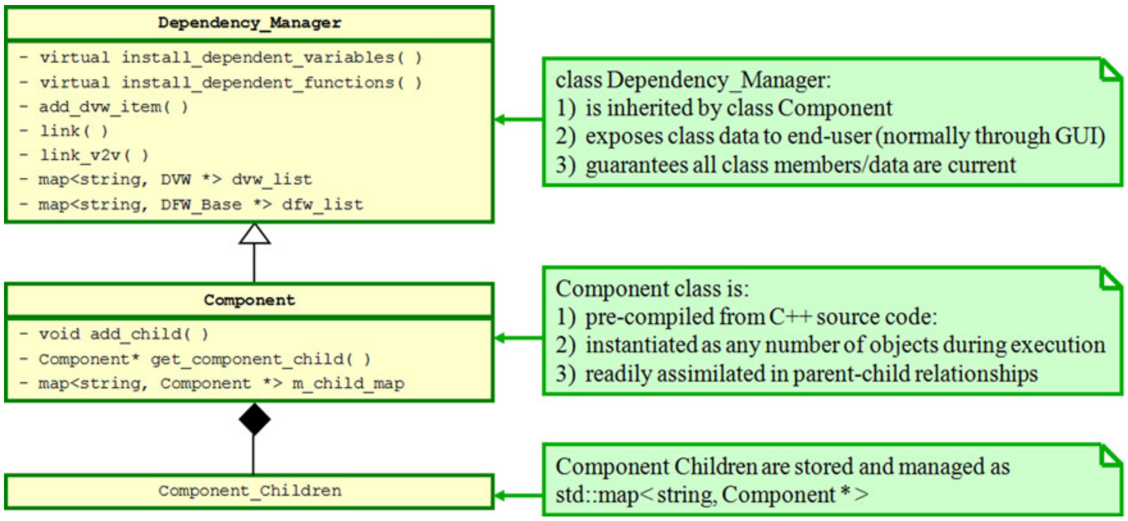

Dependency_Manager
- virtual install_dependent_variables( )
- virtual install_dependent_functions( )
- add_dvw_item( )
- link( )
- link_v2v( )
- map<string, DVW *> dvw_list
- map<string, DFW_Base *> dfw_list
class Dependency_Manager:
1) is inherited by class Component
2) exposes class data to end-user (normally through GUI)
3) guarantees all class members/data are current
Component
- void add_child( )
- Component* get_component_child( )
- map<string, Component *> m_child_map
Component class is:
1) pre-compiled from C++ source code:
2) instantiated as any number of objects during execution
3) readily assimilated in parent-child relationships
Component_Children
Component Children are stored and managed as
std::map< string, Component * >

Fig. 22 AVEC::Component Class Inheritance.

Also note in Fig. 22, only a small portion of the Component class contents are shown. These include two class functions, add_child( ) and get_component_child( ). The one data item listed is child_map. Member variable m_child_map is of type map contained in the C++ standard template library (STL). Among other features, STL significantly facilitates data array/vector management (for ANY data type or class) and map is a member of this library. The map class is essentially a vector in which the programmer can reference individual members in terms of a key character string instead of an integer index.

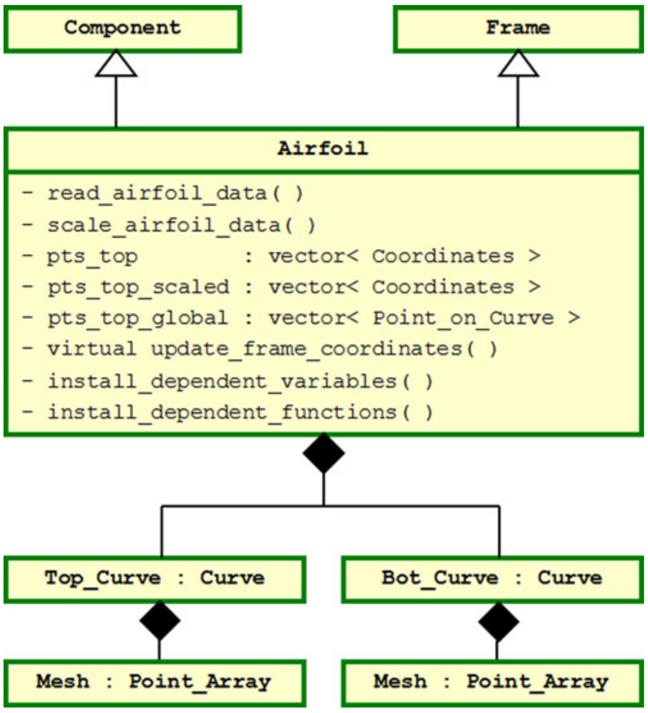

Component
Frame
Airfoil
- read_airfoil_data( )
- scale_airfoil_data( )
- pts_top : vector< Coordinates >
- pts_top_scaled : vector< Coordinates >
- pts_top_global : vector< Point_on_Curve >
- virtual update_frame_coordinates( )
- install_dependent_variables( )
- install_dependent_functions( )
Top_Curve : Curve
Bot_Curve : Curve
Mesh : Point_Array
Mesh : Point_Array

Fig. 23 UML object diagram for AVEC::Airfoil.

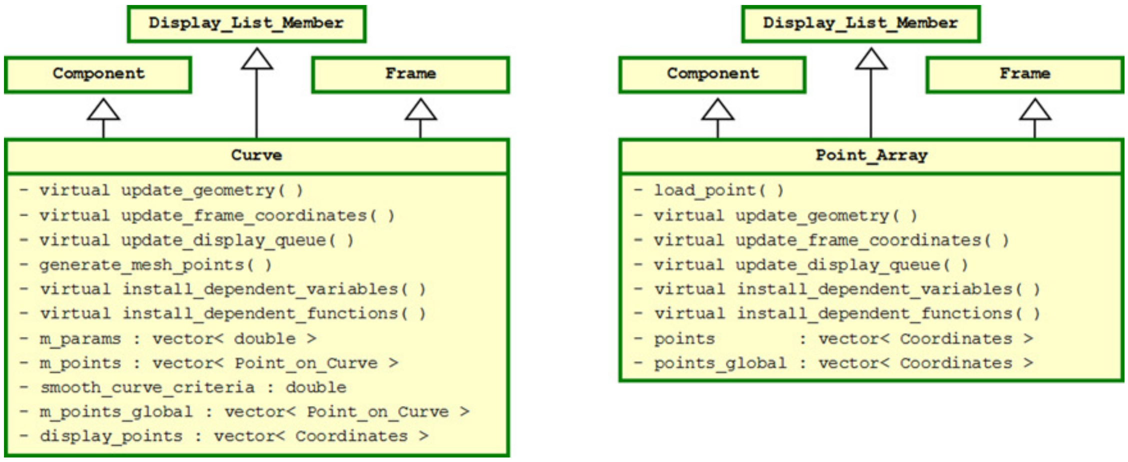

Display_List_Member
Component
Frame
Curve
- virtual update_geometry( )
- virtual update_frame_coordinates( )
- virtual update_display_queue( )
- generate_mesh_points( )
- virtual install_dependent_variables( )
- virtual install_dependent_functions( )
- m_params : vector< double >
- m_points : vector< Point_on_Curve >
- smooth_curve_criteria : double
- m_points_global : vector< Point_on_Curve >
- display_points : vector< Coordinates >
Display_List_Member
Component
Frame
Point_Array
- load_point( )
- virtual update_geometry( )
- virtual update_frame_coordinates( )
- virtual update_display_queue( )
- virtual install_dependent_variables( )
- virtual install_dependent_functions( )
- points : vector< Coordinates >
- points_global : vector< Coordinates >

Fig. 24 Inherited classes behind AVEC::Curve and AVEC::Point_Array.

In Fig. 23, we see Airfoil inherits from Component. Thus, consistent with Component class, Airfoil can instantiate other Component children. In Fig. 23, we see Airfoil class has two children and two grandchildren. The two children are Curve class that allows the airfoil to be discretized and graphically rendered. Each Curve class has one child of class Point_Array that discretizes the curve. Curve and Point_Array classes are depicted in Fig. 24. So, while Airfoil is a native AVEC class, Airfoil alone cannot be drawn (graphically rendered). In Fig. 24, Curve and Point_Array inherit from Display_List_Member. Figure 25 expands Display_List_Member which inherits from class GL_Rendering. The Airfoil assembly depicted in Fig. 23 is an end user-defined (derived) class that can be saved-in and recalled-from a library of user defined classes.

Class Airfoil in Fig. 23 inherits from both Component and Frame. (With C++, the order of inheritance is important.) With Component, Airfoil is automatically prepared to be integrated into a larger Component parent–child data tree at run time. Class Frame is posed as a UML class in Fig. 26. With Frame, Airfoil coordinates can be rotated and translated such that it lines up with other Components. As with Airfoil in Fig. 23, class Curve and class Point_Array also inherit from Component and Frame classes, as indicated in Fig. 24. Curve and Point_Array also inherit from class Display_List_Member, which is tailored, by a programmer, to drive OpenGL graphical rendering. Thus, Curve and Point_Array objects can be drawn in the window labeled “Geometric Rendering” in Fig. 15. In contrast, Airfoil alone cannot be drawn.

B. Programming with Virtual Functions in AVEC

Virtual functions are a standard part of C++. Virtual functions are members of an abstract class that can be specialized to support inherited classes. Table 3 provides a list of virtual functions and their parent class.

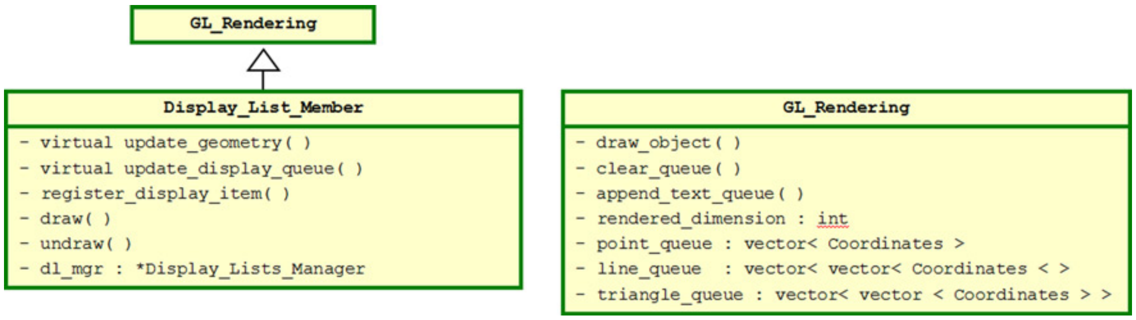
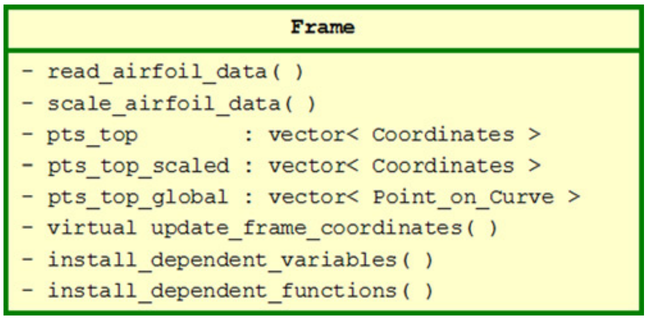

GL_Rendering
Display_List_Member
- virtual update_geometry( )
- virtual update_display_queue( )
- register_display_item( )
- draw( )
- undraw( )
- dl_mgr : *Display_Lists_Manager
GL_Rendering
- draw_object( )
- clear_queue( )
- append_text_queue( )
- rendered_dimension : int
- point_queue : vector< Coordinates >
- line_queue : vector< vector< Coordinates < >
- triangle_queue : vector< vector < Coordinates > >

Fig. 25 Classes to manage geometry in a graphic display.



```
Frame
- read_airfoil_data( )
- scale_airfoil_data( )
- pts_top        : vector< Coordinates >
- pts_top_scaled : vector< Coordinates >
- pts_top_global : vector< Point_on_Curve >
- virtual update_frame_coordinates( )
- install_dependent_variables( )
- install_dependent_functions( )
```

Fig. 26 Class to position and orient coordinate frames.

Table 3 Virtual functions in AVEC

Function name	Supports	Abstract class
update_geometry( )	geometry kernel	Display_List_Member
update_display_queue( )	GL graphics rendering	Display_List_Member
update_frame_coordinates( )	geometry orientation	Frame
install_dependent_variables( )	declare dependent variables	Dependency_Manager
install_dependent_functions( )	declare dependent functions	Dependency_Manager

Now we can explain how a virtual function facilitates the drawing of items in the product model tree in Fig. 16. Each item in the product model tree is of class (or inherits from class) Model_Tree_Item. Model_Tree_Item contains a pointer to class Component or a variant of Component such as Box or Curve. Box also inherits from Display_List_Member. So how does a Model_Tree_Item know how to distinguish between its variants? In other words, how does Model_Tree_Item know when to draw a Box and when to draw a Curve?

In Table 3, we see that update_display_queue( ) is a virtual function in Display_List_Member, for example. Class Curve inherits from Display_List_Member. As programmer, a specialized version of update_display_queue( ) was created within Curve class. The specialized version creates OpenGL directives to draw a box using a simple set of OpenGL directives. Now, Curve::update_display_queue( ) is a particular virtual function that can be referenced as if it were a generic Display_List_Member:: update_display_queue( ). The various members of the product model tree in Fig. 15 can be drawn as long as they point to component variants that also inherit from class Display_List_Member.

If this explanation seems a bit convoluted (and it is) and if this points out the complexity of C++ (and it does), then a C++ programmer (Level II) does not need to be concerned. AVEC has already taken care of the complexity and the programmer only needs to create a specialized virtual function update_display_queue( ) for any new variant of Component. The programmer has access to many self-explaining examples such as the source for class Box. The AVEC programmer (Level III) has designed in C++ virtual functions to make the C++ programmer's job (Level II) easy. Subsequently, the C++ programmer's job is to make AVEC easy for the end user (Level I) to operate at the CAD level.

C. Dependency Management in AVEC

Parametric design variables enable design automation and are integral with AVEC. The resulting trail of design dependencies are automated in a way that guarantees all models are consistent as the end user engages in a design exploration mode. This functionality has been achieved in the AVEC pilot code. The proposed AVEC pilot code will enhance dependency management to include design sensitivities using automatic differentiation as described in [30]. The following description provides a view of what has actually been programmed and demonstrated within AVEC.

As indicated in Fig. 22, Component class inherits from Dependency_Manager. Dependency management guarantees that every piece of model data are consistent with respect to its master variable. Dependency_Manager enhances Component class with the ability to declare and manage any set of variables with persistence. Here, persistence means that any change in the data will be carried through the entire model according to a customized (user-defined) dependency trail. Persistence is managed efficiently. The valid/invalid Boolean status persists. But time-consuming calculations are delayed until the user requests dependent data that was earlier declared to be invalid. This feature is called “lazy evaluation” or “demand-driven” in the literature.

Dependency management is an effective tool when properly integrated with a design process. The cost of dependent data storage should be balanced with the cost of dependent data recalculation. Each wrapped variable (or function) comes with significant data overhead to manage its Boolean valid/invalid status and maintain all slave/master dependency declarations. AVEC programmers (Level II) have the freedom to select key variables for the dependency trail and secondary variables buried in member functions.

Virtual functions are listed in Table 3. Dependency_Manager has two virtual functions that are specialized for each inherited class. These are install_dependent_variables( ) and install_dependent_functions( ) as seen in Fig. 22. When a C++ class (e.g. Component class) inherits from Dependency_Manager, these two virtual functions are available to wrap dependent variables and dependent functions in class Dependent_Variable_Wrapper (DVW) and Dependent_Function_Wrapper (DFW) respectively.

The dependency trail can now be automated with a standard procedure. For instance, geometric classes (Curve, Surface, etc.) are used repeatedly throughout a model. We want the parameters in geometry class dependencies to be automatically consistent with all the other objects used in a design. We want to avoid a customized dependency manager for each instance of class Curve or each instance of class Surface. Another advantage of DVW and DFW class is gained with data publishing. One GUI tailored to DVW and DFW can serve all classes that inherit from class Dependency_Manager. Likewise, there is an advantage where models are copied, pasted, saved, retrieved in files or a database. A common Dependency_Manager can be supported by a single format.

AVEC is set up to facilitate C++ programming of specialized Component classes (thus Dependency_Manager through inheritance) with a set of declared dependent variables and declared dependent functions that are compiled as part of the class. Dependencies between declared variables (DVW) and functions (DFW) *within* a class are also compiled as part of the class. In this way, the behavior of the class can be guaranteed by the programmer upon instantiation (loaded during run-time).

Following object instantiation, the end user (Level I) can establish dependencies between variables (DVW) of one object with compatible variables (DVW) of other objects. Recall the user procedure for establishing inter-object dependencies as was outlined in Fig. 20.

Figure 22 features Dependency_Manager class which contains dvw_list and dfw_list. Figure 27 depicts the operation of DVW and DFW in terms of a UML State Machine Diagram. The DVW can be in any of three states: Current/Valid, NotCurrent/InValid, Invalid/Waiting.

Figure 28 depicts some details of the Dependency_Manager class. The reduced version of Dependency_Manager shown contains two public (any user of this class has access to public data) functions and two protected (can only be accessed by class member functions) data items. These two data items, dfw_list and dvw_list, are of type map (C++

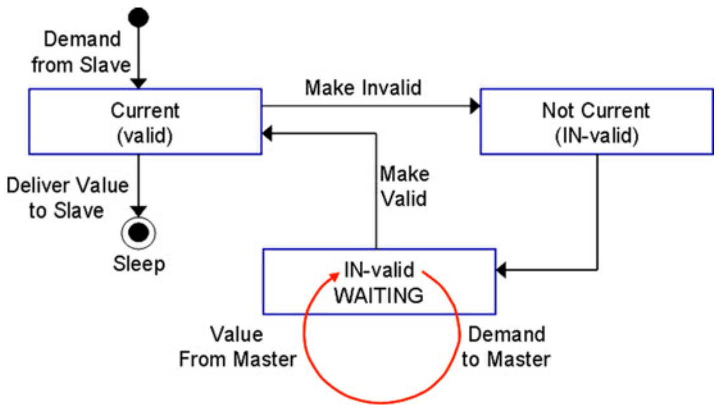


Fig. 27 UML state machine diagram for Dependent_Variable_Wrapper Class.

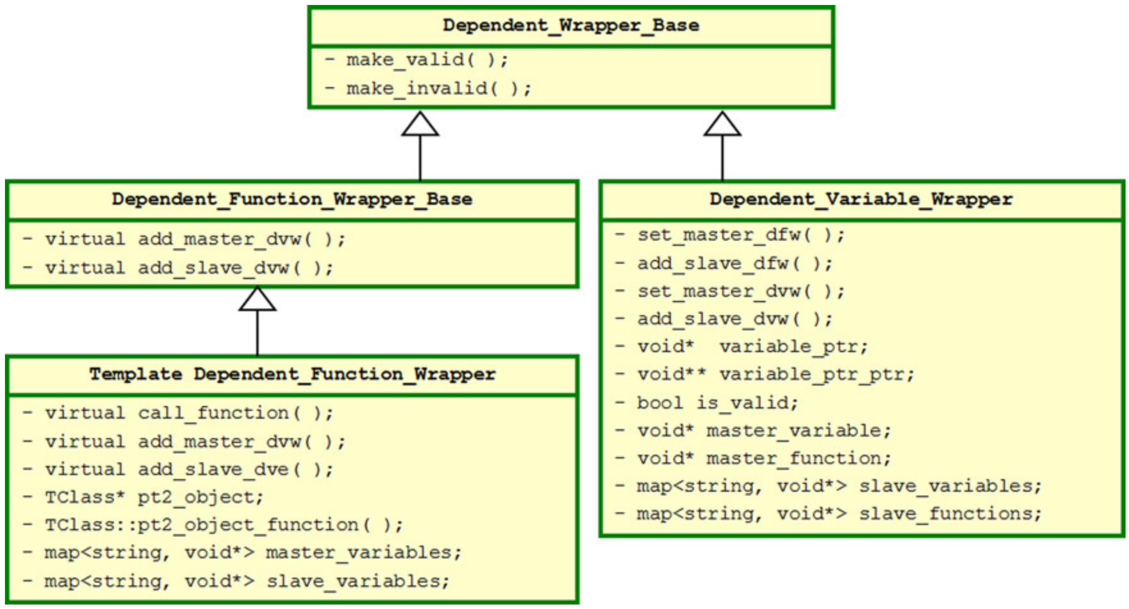

Dependent_Wrapper_Base
- make_valid( );
- make_invalid( );
Dependent_Function_Wrapper_Base
- virtual add_master_dvw( );
- virtual add_slave_dvw( );
Dependent_Variable_Wrapper
- set_master_dfw( );
- add_slave_dfw( );
- set_master_dvw( );
- add_slave_dvw( );
- void* variable_ptr;
- void** variable_ptr_ptr;
- bool is_valid;
- void* master_variable;
- void* master_function;
- map<string, void*> slave_variables;
- map<string, void*> slave_functions;
Template Dependent_Function_Wrapper
- virtual call_function( );
- virtual add_master_dvw( );
- virtual add_slave_dve( );
- TClass* pt2_object;
- TClass::pt2_object_function( );
- map<string, void*> master_variables;
- map<string, void*> slave_variables;

Fig. 28 AVEC Dependent_Wrapper_Base derivatives.

standard template library). The map dfw_list is essentially a vector of class type Dependent_Function_Wrapper. The map dvw_list is essentially another vector of type Dependent_Variable_Wrapper. A source summary of Dependent_Function_Wrapper (DFW) and Dependent_Variable_Wrapper (DVW) are provided in the respective boxes in Fig. 28.

Dependent_Variable_Wrapper class contains seven protected member variables. The variable type void* is peculiar to C++. The * is an indicator that the named variable is to be managed in terms of its position in memory. The * syntax is a declaration for a C++ pointer. The type void is a catch-all data type. The void data type must be type-cast (e.g. as int or float, etc.) wherever void is operated upon. The type void** is a double pointer. Essentially, this is a pointer to a pointer of type void. C++ manuals warn of the dangers of void and pointers. However, with skillful care on the part of the AVEC programmer (Level III), the operation of Dependent_Variable_Wrapper must be guaranteed to behave gracefully. In Fig. 28, we see Dependent_Variable_Wrapper contains the following members:

• variable_ptr	Points to memory location for any type variable of interest. Exception is variable_ptr_ptr.
• variable_ptr_ptr	A pointer to pointer in memory and used to wrap pointers
• is_valid	Declares whether the value behind variable_ptr is consistent with master variables
• master_variable	NULL or pointer to the DVW that controls the value of variable_ptr
• master_function	NULL or pointer to the DFW that controls the value of variable_ptr
• slave_variables	STL map to a list of DVW whose Boolean value for is_valid is set to false when the present (in this context, the master) variable is changed and/or the present value of is_valid is set to false
• slave_functions	STL map to a list of DFW whose variable_ptr is controlled by the present variable_ptr

The two public member functions make_valid( ) and make_invalid( ) shown in Fig. 28 are fairly complex procedures that cascade through an instantiated Component object tree (described above in the sense that class Component inherits from Dependency_Manager).

DFW class structure is a bit more complex (than DVW class). Some of the details are found in [17]. For instance, one might be interested to read about “functors”. Unlike the DVW class, the DFW class is a template. In other words, functions that are to be wrapped by DFW class are always declared upon compilation and never during runtime.

Thus, classes that inherit from Dependency_Manager (and/or Component) in Fig. 22 have the option to declare virtual functions that declare all dependent variables and functions within the class.

- install_dependent_functions( )
- install_dependent_variables( )

As with Airfoil in Fig. 23, class Box in Fig. 29 inherits from Component and Frame classes. Box also inherits from class Display_List_Member.

The benefits of virtual functions in AVEC were discussed in Sec. V.B. It involves a small amount of programming effort for C++ programmers (Level II) with the result that the class becomes very manageable for end users (Level I) who can interactively declare dependencies between variables of different classes. See the example virtual function declared in Fig. 30 for Box::install_dependent_variables( ). A careful observer can notice that the declared variables in Fig. 30 (for class Box) are automatically displayed in Fig. 17.

Dependency management is employed in AVEC with some implicit rules as laid out in Table 4. A DVW can point to either one master variable or one master function. A DVW can point to many slave variables. A DFW can point to many master variables (DVW) and many slave variables (DVW). But a DFW cannot point to another DFW.

With this understanding of dependency management, we can return to Fig. 13 with greater insight. The boxes along the diagonal are roughly representative of instantiated AVEC analysis classes (yet to be developed of course) wrapped in a DFW. Each box in Fig. 13 is labeled according to its primary class function. In AVEC, dependent functions (and associated dependent variables) are compiled as part of a class structure. The circles contain sets of DVW. The connecting lines are representative of user defined links between DFW and DVW. The user process for declaring these variable links was depicted in Fig. 20. The example use case in Fig. 3 orchestrates the flow in Fig. 13. Of course, as indicated in Fig. 3, design processes (i.e. use cases) are cyclical. That is, a design process iterates until it converges to produce desired results. At present, AVEC Dependency_Manager does not address an optimization convergence process. Some thought has gone into designing such a system. However, at present it remains a research topic as discussed in [30].

Reference [18] emphasized the need for dependency management. So, how does dependency management facilitate a computationally intensive design optimization process? The answer continues to evolve with years of experience. As discussed, the answer involves the integration of dependency management and automated design sensitivities (as well as cascading uncertainties). These are opportunities for research, the raison d'etre for AVEC.

AVEC admittedly lacks much of the convenient interactive features that are characteristic of the scripted languages identified in [17]. However, it turns out, those features did not significantly contribute to the operation of the computational design process described in [16]. The scripted code used in [16] was excellent for prototyping the

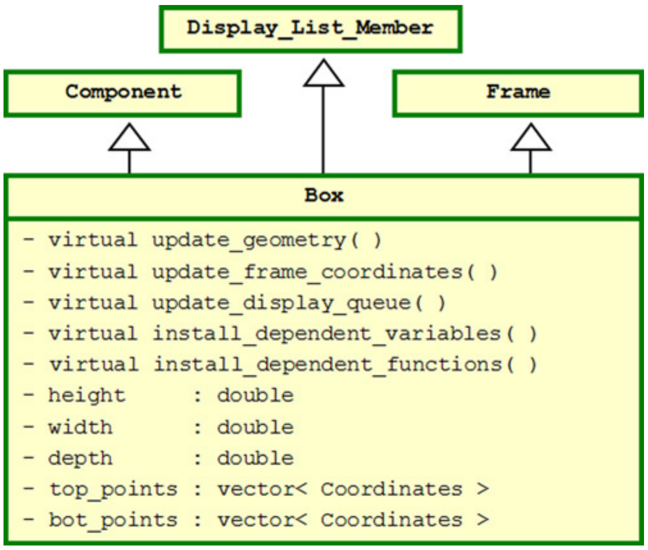
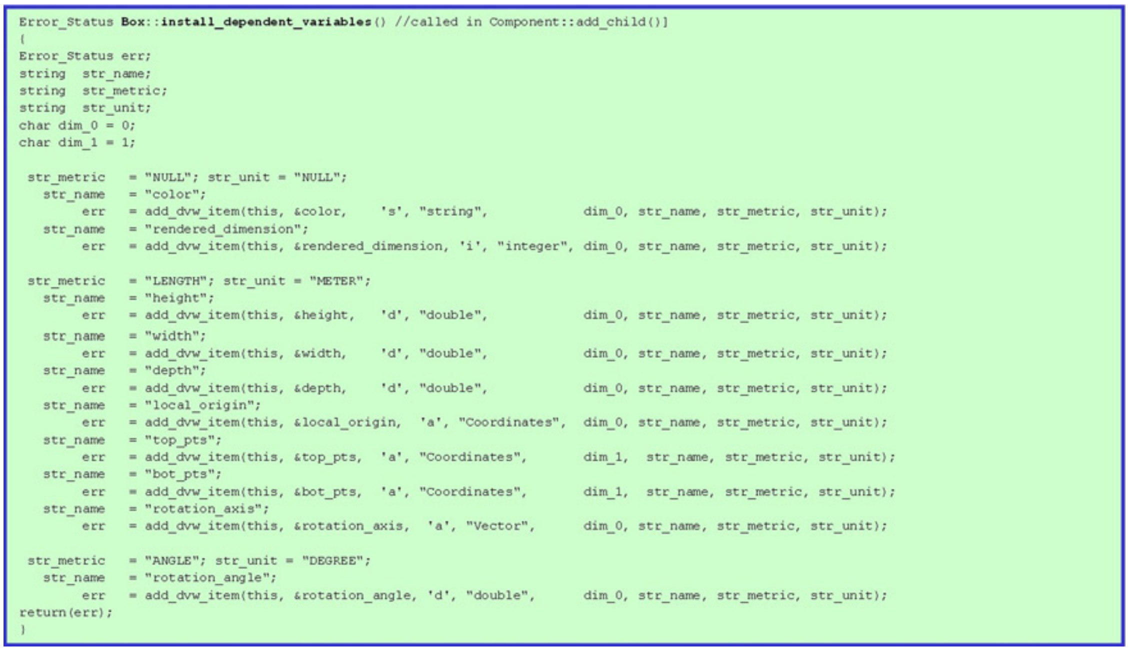

Display_List_Member
Component
Frame
Box
- virtual update_geometry( )
- virtual update_frame_coordinates( )
- virtual update_display_queue( )
- virtual install_dependent_variables( )
- virtual install_dependent_functions( )
- height : double
- width : double
- depth : double
- top_points : vector< Coordinates >
- bot_points : vector< Coordinates >

Fig. 29 UML class diagram for AVEC::Box.



```
Error_Status Box::install_dependent_variables() //called in Component::add_child()]
{
Error_Status err;
string  str_name;
string  str_metric;
string  str_unit;
char dim_0 = 0;
char dim_1 = 1;

 str_metric   = "NULL"; str_unit = "NULL";
   str_name   = "color";
        err   = add_dvw_item(this, &color,    's', "string",            dim_0, str_name, str_metric, str_unit);
   str_name   = "rendered_dimension";
        err   = add_dvw_item(this, &rendered_dimension, 'i', "integer", dim_0, str_name, str_metric, str_unit);

 str_metric   = "LENGTH"; str_unit = "METER";
   str_name   = "height";
        err   = add_dvw_item(this, &height,   'd', "double",            dim_0, str_name, str_metric, str_unit);
   str_name   = "width";
        err   = add_dvw_item(this, &width,    'd', "double",            dim_0, str_name, str_metric, str_unit);
   str_name   = "depth";
        err   = add_dvw_item(this, &depth,    'd', "double",            dim_0, str_name, str_metric, str_unit);
   str_name   = "local_origin";
        err   = add_dvw_item(this, &local_origin,  'a', "Coordinates",  dim_0, str_name, str_metric, str_unit);
   str_name   = "top_pts";
        err   = add_dvw_item(this, &top_pts,  'a', "Coordinates",       dim_1,  str_name, str_metric, str_unit);
   str_name   = "bot_pts";
        err   = add_dvw_item(this, &bot_pts,  'a', "Coordinates",       dim_1,  str_name, str_metric, str_unit);
   str_name   = "rotation_axis";
        err   = add_dvw_item(this, &rotation_axis,  'a', "Vector",      dim_0, str_name, str_metric, str_unit);

 str_metric   = "ANGLE"; str_unit = "DEGREE";
   str_name   = "rotation_angle";
        err   = add_dvw_item(this, &rotation_angle, 'd', "double",      dim_0, str_name, str_metric, str_unit);
return(err);
}
```

Fig. 30 Virtual function Box::install_dependent_variables.

Table 4 Dependency rules

	DVW	DFW
Master	DVW can have one DVW or one DFW master	DFW can have many DVW but no DFW master
Slave	DVW can have many DVW or many DFW slaves	DFW can have many DVW but no DFW slave

process. However, the need for development of a design optimization process for a family of design variants is cause for pause and rethinking. AVEC is being designed with this experience in mind as well as the need for researchers to have access to far-reaching design optimization models of advanced systems.

Variable values in AVEC can be transfered in terms of its equivalent string value or shared in terms of its pointer address in memory. The ability to share pointer memory is a major convenience where software integration is concerned. For instance, shared pointer memory saves the pain of translating each data item when an entire class is passed. The ability to exchange an equivalent string value enables future expansion of AVEC to include geographically distributed modeling (multiple computers) with the aid of [20]. Reference [20] is best suited for geographical distributions of design modeling between processors and clusters of processors where data security is a concern.

VI. AVEC Pilot Status Update

As indicated, AVEC is a pilot code in development. The purpose of a pilot code is to create solid requirements for production code. This purpose is being achieved. To date, most attention has gone into development of geometric entities. In Fig. 13, this relates to block C Parametric Geometry. The status of recently added or enhanced features is described here:

A. Units of Measurement

The management of units of measure is a feature of AVEC. In AVEC, units of measure are user specified. The conversion data for the global table is archived in an ASCII file that is easily modified by the end user (Level I). Class AVEC_Units_Catalog is declared in global space as avec_units_catalog. Conversion data for this class is read from a file from within the main( ) calling program. Data are automatically loaded when AVEC is instantiated. Base units of measurement are maintained in any instantiated class of type model. Units are converted to and from base units only when the end user interacts with class variables for data viewing and modification (normally through a graphical user interface). Thus, the model is guaranteed to use a consistent set of base units (e.g. meters for length, kilograms for mass, seconds for time, etc.). Units of measurement at variable level are user specified and arbitrary to the extent unit conversion is declared in the global table.

B. Geometric Entities

New geometric entities will naturally arise with AVEC. Figure 31 is a depiction of an instantiated Surface class as rendered in AVEC. This is a Hermite surface based on a rectangular matrix of coordinates and vectors. A special AVEC_Type was developed to support Surface development. The class Point_on_Surface contains both coordinates and three vectors. As indicated in Fig. 31, a Hermite surface formulation is very adaptable. However, future developments with a sophisticated geometry library will create an inheritable NURBS class that can be reduced to Hermite form.

The number of geometric entities required to render conceptual and preliminary design features is small. For one-dimensional constructs, we require line segments and space curves. For two-dimensional constructs, we require a planar patch and a Hermite (cubic) parametric surface. Classes for higher-order geometric assemblies are in development. These include contour class to interpolate a surface between a series of curves. Classes to address Boolean intersection and trimming will become valuable. Where computational design is one's business, meshing and geometry are very closely linked. One should not be developed without the other in mind. Although AVEC will be developed with a small number of native geometry classes, AVEC is designed to facilitate the integration of any geometry kernel and meshing utilities.

C. The Geometry Viewer

Viewing orientation of geometry in the AVEC graphical pane is controlled through the main view menu with options for (a) translate (b) rotate (c) zoom (d) adjust viewing volume (e) set camera coordinates (f) set target coordinates. All action is based on standard OpenGL viewing constructs. Translations shift camera and target simultaneously. Rotations take place in local spherical coordinates. Zoom responds with exponential action (slow or fast depending on distance between camera and target).

D. Save and Restore Features

AVEC models can be saved and restored in XML format. The data structure is preserved, including all parent–child relationships between components and all master/slave dependencies between variables. Creating the save function

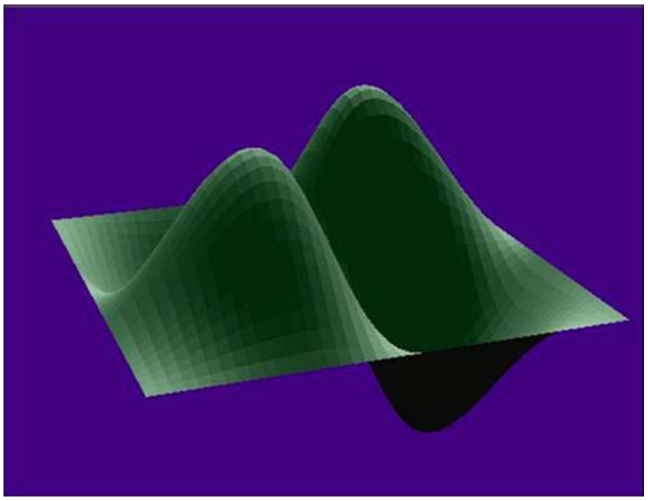


Fig. 31 AVEC surface class rendering.

is a simple process. Model restoration is much more challenging to program and requires two sweeps. The first sweep establishes parent–child hierarchy. The second sweep establishes master/slave dependencies.

AVEC provides the ability to save derived classes. A derived class is constructed from an instantiated AVEC class in terms of the parent object and all the children progeny and with variable dependencies declared. A derived class is saved in XML format but without enumerating the independent variables. The derived class can be instantiated as a part of any subsequent model. For instance, the Enhanced_Airfoil class will be interactively constructed from the basic Airfoil class with class Curve instantiated as child objects. This multilevel class will be saved in XML format for recall as part of other models that require the Enhanced_Airfoil class. Dependency paths are reassigned relative to the restored point in the model tree.

The functionality used to save and restore models and derived-classes in AVEC will also support a future copy-object( ) function. However, this will be addressed after some significant level of analysis management has been developed in AVEC. This will drive the need for database management. (for example, [29])

VII. Conclusions

A computational design environment has the potential to fill a gap between technology prioritization processes and revolutionary system capabilities. QTA has been contemplated, but has not been able to generate high-fidelity models with the speed required to keep pace with the quick response needs of the technology business.

AVEC is the pilot code for an envisioned integration tool that serves a design research team with a library of classes that can be inherited along with standard C++ code. AVEC addresses common basic needs such as dependency management, graphical rendering, etc. Collaborators benefit with a library of inheritable classes that automate and manage mundane computer processes. AVEC represents a possible prototype towards the establishment of open-source integration software that supports Computational Design directly and AFRL QTA indirectly.

AVEC is compiled in C++. The choice of compiled C++ serves the computational community best with highly reliable code. The functionality in the pilot AVEC code could be readily replicated using interpreted code. Personal experience indicates the environments based on interpreted code are best suited for prototyping a design process. However, the interpreted codes lack software discipline and tend to misbehave as design space increases with model size and numbers of design variables. References [16,22] provide a record of this experience.

AVEC is in an early stage, and will significant development before practical design applications are published. Today, a C++ programmer today can readily inherit and modify AVEC classes related to dependency management and geometric rendering. End users will appreciate the parametric nature of design models that become established in AVEC. Ongoing developments with automatic differentiation will be integrated with AVEC automated dependency management.

AVEC will undergo a major rewrite in order to accommodate gradient based design optimization in order to realize the QTA functionality simply laid out in Fig. 9. With recent developments in automatic differentiation (AD), it is possible to create new class structures that extend current programming codes to simultaneously and automatically generate a trail of design sensitivities. As envisioned, AD and automated design variable management will be integrated to support QTA with a large number of design optimized system variants representative of various permutations of technologies. AD is discussed in [30] as part of a specific plan to generate geometric design sensitivities.

The SensorCraft concept will provide an application focus for establishing software development priorities. The design strategy begins with simple geometries and high-order physics which could ultimately converge on a detailed design study following a Pareto-optimization process. AVEC will support the creation of a large database of system design variants for post-processing, and thereby realize practical QTA for effective technology management in AFRL in collaboration with system users.

References

[1] Zeh, J. M., and Schumacher, C. J., "Simulation Based Research and Development in the Air Force Research Laboratory," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Montreal, Canada, 6–9 Aug. 2001, AIAA-2001-4371.

[2] Tejtel, D., Zeune, C. H., Revels, A. R., Held, T. W., and Braisted, W. R., "Breathing New Life into Old Processes: An Updated Approach to Vehicle Analysis and Technology Assessment," *AIAA 5th Aviation, Technology, Integration, and Operations Conference* (*ATIO*), Arlington, VA, 26–28 Sept. 2005, AIAA-2005-7304.

[3] Biltgen, P., and Mavris, D. N., “A Methodology for Capability-Focused Technology Evaluation of Systems-of-Systems,” *45th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, 8–11 Jan. 2007, AIAA-2007-1331.

[4] Cavanaugh, S., Chytka, T., Arcara, P., Jones, S., Stanley, D., and Wilhite, A., “NASA Langley Systems Analysis & Concepts Directorate Technology Assessment/Portfolio Analysis,” *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA, 6–8 Sept. 2006, AIAA-2006-7029.

[5] Keyes, J., Troutman, P. A., Saucillo, R., Cirillo, W. M., Cavanaugh, S., and Stromgren, C., “NASA Langley Research Center Systems Analysis & Concepts Directorate Participation in the Exploration Systems Architecture Study,” *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA, 6–8 Sept. 2006, AIAA-2006-7030.

[6] Robinson, J. S., Martin, J. G., Bowles, J. V., Mehta, U. B., and Snyder, C. A., “An Overview of the Role of Systems Analysis in NASA’s Hypersonics Project,” *14th AIAA/AHI Space Planes and Hypersonic Systems and Technolgies Conference*, 6–9 Nov. 2006, National Convention Centre, Canberra, Australia, AIAA-2006-8013.

[7] De Weck, O. L., Chang, D. D., Suzuki, R., and Morikawa, E., “Quantitative Assessment of Technology Infusion in Communications Satellite Constellations,” *21st International Communications Satellite Systems Conference and Exhibit*, 15–19 Apr. 2003, Yokohama, Japan, AIAA-2003-2355.

[8] Simpson, T. W., Carlsen, D. E., Congdon, C. D., Stump, G., and Yukish, M. A. “Trade Space Exploration of a Wing Design Problem Using Visual Steering and Multi-Dimensional Data Visualization,” *49th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Material Conference*, Schaumburg, IL, 7–10 April 2008, AIAA-2008-2139.

[9] Moffitt, B. A., Bradley, T. H., Parekh, D. E., and Mavris, D., “Validation of Vortex Propeller Theory for UAV Design with Uncertainty Analysis,” *46th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, 7–10 Jan. 2007.

[10] Binder, J. D., “Knowledge-Based Engineering—Automating the Process,” *Aerospace America*, Vol. 34, March 1996, pp. 14–16.

[11] Kroo, I., “An Interactive System for Aircraft Design and Optimization,” *AIAA Aerospace Design Conference*, Irvine, CA, 3–6 Feb. 1992, AIAA-92-1190.

[12] Malone, B., and Woyak, S., “An Object-Oriented Analysis and Optimization Control Environment for the Conceptual Design of Aircraft,” *1st AIAA Aircraft Engineering, Technology, and Operations Congress*, Los Angeles, CA, 19–21 Sept. 1995, AIAA-95-3862.

[13] Kim, H., Malone, B., and Sobieszczanski-Sobieski, J., “A Distributed, Parallel, and Collaborative Environment for Design of Complex Systems,” *45th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Palm Springs, CA, 19–22 April 2004.

[14] Rodriguez, D., and Sturdza, P., “A Rapid Geometry Engine for Preliminary Aircraft Design,” *44th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, 9–12 Jan. 2006, AIAA-2006-0929.

[15] Vandenbrande, J., Grandine, T. A., and Hogan, T., “The Search for the Perfect Body: Shape Control for MultiDisciplinary Design Optimization,” *44th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, 9–12 Jan. 2006, AIAA-2006-0928.

[16] Blair, M., Canfield, R. A., and Roberts, Jr., R. W., “Joined-Wing Aeroelastic Design with Geometric Non-linearity,” *Journal of Aircraft*, Vol. 42, No. 4, July–Aug. 2005, pp. 832–848.
doi: 10.2514/1.2199

[17] Blair, M., “Computational Design Challenges for Non-Linear Aeroelastic Systems,” *International Forum on Aeroelasticity and Structural Dynamics*, Munich, Germany, 28–30 June 2005, IF-145.

[18] Veley, D. E., Blair, M., and Zweber, J. V., “Aerospace Technology Assessment System,” *7th AIAA/ISSMO MultiDisciplinary Analysis and Optimization Conference*, St Louis, MO, 2–4 Sept. 1998, AIAA-98-4825.

[19] Stephenson, W. J., Veley, D. E., and Hill, S., “Composite Vehicle Design Environment,” *48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Material Conference*, Honolulu, HI, 23–26 April 2007, AIAA-2007-2371.

[20] Sobolewski, M., and Kolonay, R. M., “Federated Grid Computing with Interactive Service-Oriented Programming,” *Concurrent Engineering*, Vol. 14, No. 1, March 2006.
doi: 10.1177/1063293X06064148

[21] Kolonay, R. M., and Burton, S. A., “Object Models for Distributed Miltidisciplinary Analysis and Optimization (MAO) Environments that Promote CAE Interoperability,” *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY, 30 Aug.–1 Sept. 2004, AIAA-2004-4599.

[22] Hunten, K., McCulley, C., De La Garza, A., and Blair, M., “The Application of the MISTC Framework to Structural Design Optimization,” *46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Austin, TX, 18–21 April 2005, AIAA-2005-2127.

[23] Yu, W., and Blair, M., “GEBT: A General-Purpose Tool for Nonlinear Analysis of Composite Beams,” *51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Material Conference*, Orlando, FL, 12–15 April 2010, AIAA-2010-3019.

[24] Johnson, F. P., “SensorCraft,” *AFRL Technology Horizons*, Vol. 2, No. 1, March 1, pp. 10–11, http://www.afrlhorizons.com/Briefs/Dec04/VA0308.html.

[25] Kim, Y. I., Park, G. J., Kolonay, R. M., Blair, M., and Canfield, R. A., “Nonlinear Dynamic Response Structural Optimization of a Joined-Wing Using Equivalent Static Loads,” *49th AIAA/ASME,ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Schaumburg, IL, 7–10 April 2008, AIAA-2008-2159.

[26] Rasmussen, C. C., Canfield, R. A., and Blair, M., “Joined-Wing Sensor-Craft Configuration Design,” *Journal of Aircraft*, Vol. 43, No. 5, Sept.–Oct. 2006.
doi: 10.2514/1.21951

[27] Raymer, D. P., *Aircraft Design: A Conceptual Approach*, AIAA Education Series, AIAA, Reston, VA, 1999.

[28] Demasi, L., and Livne, E., “The Structural Order Reduction Challenge in the Case of Geometrically Nonlinear Joined-Wing Configurations,” 48*th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Material Conference*, Honolulu, HI, 23–26 April 2007, AIAA-2007-2052.

[29] Lin, R., and Afjeh, A. A., “An XML-Based Integrated Database Model for MultiDisciplinary Aircraft Design,” *Journal of Aerospace Computing, Information, and Communication*, Vol. 1, March 2004, pp. 154–172.
doi: 10.2514/1.2006

[30] Blair, M., “SCOOT: Sensitivity Class with Operator Overloaded Types,” *51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Material Conference*, Orlando, FL, 12–15 April 2010, AIAA-2010-2918.

Ellis Hitt
Associate Editor