The Computer Aided Aircraft-design Package (CAAP)

By Guy U. Yalif
(617) 973-1015

Abstract

The preliminary design of an aircraft is a complex, labor-intensive, and creative process. Since the 1970's, many computer programs have been written to help automate preliminary airplane design. Time and resource analyses have identified, "a substantial decrease in project duration with the introduction of an automated design capability" (Ref. 1). Proof-of-concept studies have been completed which establish "a foundation for a computer-based airframe design capability" (Ref. 1). Unfortunately, today's design codes exist in many different languages on many, often expensive, hardware platforms. Through the use of a module-based system architecture, the Computer Aided Aircraft-design Package (CAAP) will eventually bring together many of the most useful features of existing programs. Through the use of an expert system, it will add an additional feature that could be described as indispensable to entry level engineers and students: the incorporation of "expert" knowledge into the automated design process.

Introduction

It is widely recognized that good engineers need not only the textbook knowledge learned in school, but also a good "feel" for the designs with which they are working. This "feel" can only be gained with both time and experience. An expert system is an ideal way to codify and capture this "feel". This idea is the key feature of CAAP. With this package, engineers will be able to use the knowledge of their predecessors, as well as learn from it. The potential value of such a program in aiding the engineering professionals as well as the student is great.

The ultimate goal of CAAP is to design a plane in an intelligent way based on user specifications. A rough-sizing configuration is created from user inputs and then analyzed using rule based programming. Throughout the design process, the user is given total access to the CAAP database, which is implemented using object oriented programming. The user can see how variables affect each other, view their present values, and see, create, and arrange rules in a customizable fashion using "Toolbox" files. CAAP exists as a core program with Toolbox files that add functionality to that core, similarly to the popular program "MATLAB". CAAP's core program has been written while its Toolbox files are still in development.

System Overview

Preliminary aircraft design, as described in above, is a multi-faceted problem whose features have driven the choice of software platform used to implement CAAP. This section will detail the features that led to a CLIPS based implementation for CAAP. One aspect of the usefulness of an expert system to the CAAP package has already been discussed.

The design process is a *potentially iterative* procedure. This is best explained with an example. During a hypothetical airplane design, one might re-size the wing five times. On the other hand, it is possible that the engineer will not alter the original fuselage. The possibility of iterative re-design for some components and not for others defines a potentially iterative process. Airplane design is such a process, and it is therefore well modeled by the rule based programming syntax of an expert system.

As other designers have noted, "tremendous amounts of data and information are created and manipulated [during the aircraft design process] to produce numerous parts which are eventually

assembled to a sophisticated system. ... It is becoming clear that a critical issue to effective design is the efficient management of design data" (Ref. 2). The data produced during the design process is voluminous at the very least, but it is not haphazardly arranged. The information needed to design a plane falls into organized patterns. Specifically, a hierarchical structure exists for most components of an airplane. One such component is engine classification, as illustrated in Figure 1. This figure diagrams the hierarchy that is used to describe engines in CAAP. This type of logical hierarchy exists throughout the airplane design process.

The data used during airplane design can be very complicated. Each part of the plane, such as the engine, has its own specifications. Each part also contains subparts, just as the engine has a turbine, compressor, and fuel pumping system. Each of these subparts has its own specifications in addition to sub-subparts. Therefore, design data needs to be arranged in an ordered manner that is readily accessible and understandable to the user. Object Oriented Programming is useful for storing the complex, voluminous, and hierarchically arranged data produced during airplane design. The usefulness of OOP has been recognized elsewhere in the aerospace industry. In a study entitled "Managing Engineering Design Information" (Ref. 2), ten different data storage methods were examined. The conclusion: "The object-oriented data model was found to be a better data modeling method for modeling aerospace vehicle design process" than any of the others studied (Ref. 2).

OOP also facilities the organization of the large number of routines available to aid in aircraft design. Effective routine organization is a desirable quality of any airplane design program. CAAP seeks to accomplish routine organization in two ways. First, routines are grouped into the Toolbox files introduced above. Second, within each Toolbox, different equations are applied to different parts of the airplane as is appropriate. Having the ability to separate the equations according to airplane component aids in the logical organization of the program. Such separation also increases the efficiency of CAAP. For example, it would be a waste of computational time to have the aspect ratio rule searching instances of the FUSELAGE class for possible span and area values. OOP and CLIPS run-time modules allows the programmer to implement such class-specific routine separation.

As discussed above, the order of execution of the routines that analyze an airplane cannot be determined before run-time because of the potentially iterative nature of design. The routines themselves, however, are composed of equations that *do* need to be executed in a predetermined order. For example, the routine that determines the range-payload curve needs to add up the range covered during climb, cruise, and descent over and over again until the desired limiting payloads are reached. This is an ordered process that is best modeled by a procedural programming paradigm.

The desirability of using multiple programming paradigms has been discussed above. Because of these needs, CLIPS was chosen to implement CAAP. CLIPS provides useful and effective rule based, object oriented, and procedural programming paradigms as well as a high level of cross-paradigm interactions. CLIPS is also offered on a wide variety of hardware platforms, ensuring its ability to the student and professional alike.

A Macintosh platform was used to implement these program design goals. The Macintosh was chosen because of the widespread access to Macs, as opposed to the limited access available to more powerful UNIX workstations such as IRIS's or Sun's. Nonetheless, if a user had access to these workstations, CAAP would be fully portable to their environments. CAAP's text based user interface has been written completely in CLIPS. A second, graphically based user interface, however, has been designed strictly for Macintosh use. CAAP has been successfully run on many different Macintosh models, although no porting tests have been performed for other platforms.

It has been recognized that a modular layout "will preserve the ability for independent expansion at a later date" (Ref. 3). This key concept is reflected in CAAP's design. As Kroo and Takai have succinctly stated, "If the founding fathers did not envision that future aircraft might employ maneuver load alleviation, winglets, or three lifting surfaces, it may be more expedient to rewrite the code than to fool the program into thinking of winglets, for example, as oddly shaped stores" (Ref. 4). With a sectioned program, such problems can be quickly alleviated with the addition of another Toolbox or an upgrade to an existing one.

As mentioned above, CAAP is organized into a central program and a variety of Toolboxes which add functionality to the base program. The core program represents all of the code that is necessary to run an expert system which utilizes CAAP's data structures. As a result, this code has the possibility of being recycled in the future. The Toolboxes will add functionality to the core program. If someone wishes to run CAAP at all, they need to possess the core program. If someone also wishes to perform weight sizing of their aircraft, they also need to possess the Weight Toolbox. If a certain Toolbox is not present while a user is designing an airplane with CAAP, the part of the analysis that the Toolbox is responsible for will not be completed. Continuing with the previous example, if the Weight Toolbox is missing, no weights will be assigned to the various components of the plane. Missing Toolboxes could prevent an airplane from being completely designed. Nonetheless, arranging a program in this fashion allows users to customize their personal version of CAAP as they desire. The user will, as a hypothetical example, have the ability to choose a Roskam Weight Toolbox instead of a Raymer Weight Toolbox, if they so desired. In addition, if the user does not want CAAP to perform a certain kind of analysis, the Toolbox based program allows them to disable a segment of code analysis easily. It is worth noting that no time consuming re-compilation is presently necessary to remove a Toolbox from CAAP. A simple menu option allows users to choose which Toolboxes will be loaded at start-up.

Mimicking the real engineer, CAAP's core program will prompt the creation of a rough, initial sizing for an airplane. The code will then analyze this initial configuration of the plane. If the given configuration does not meet one of the user's performance specifications, or if the plane does not pass the appropriate FAR regulations, CAAP will modify a particular part of the plane.

A diagram of the system architecture that will accomplish these tasks is presented in Figure 2. The diagram depicts the interactions between CAAP's run-time modules. Into each of these run-time modules will be loaded routines that perform certain functions in the airplane design process. For example, the Performance Module will contain routines that determine performance estimates. Run-time modules are not to be confused with Toolbox files. Toolbox files are files that contain routines organized in an arbitrary manner chosen by the user. Run-time modules group routines by functionality only.

The CAAP core program is presented around all of the run-time modules in order to emphasize that it is the core code that drives all of the routines within the run-time modules and allows them to perform their allotted analytical tasks. The Initial Sizing Module produces the initial parameters for the first configuration. The rest of the modules then analyze, alter, and re-analyze the subsequent configurations. The final plane will be presented to the user through the routines in the Geometry Graphics Module. The user will then be able to change the proposed solution/configuration, and the process will start over again. This time, however, the "old" solution with the user's modifications will become the "initial configuration".

The Rule Writing Utility

The rules that make up CAAP need not only to perform their prescribed functions, but also to provide variable dependency information to the routine that performs variable assignments for CAAP. As described later in the *Consistency Maintenance* section, if a variable in the system is

calculated, all of the variables that depend on its value must be recalculated. It is useful to maintain a data storage system which can provide CAAP with these variable dependencies. Rules are no longer written in their traditional format. In previous versions of CAAP, when a program developer wanted to add a rule to the expert system, they have had to learn the syntax of rule writing in CLIPS and then how to hard code the rule into the system. This required coding some standard constructs that perform some of the repeated type checking that goes on within CAAP. These constructs were usually very long and "messy", and therefore very time consuming to write.

In CAAP Version 2.0, programmers can add rules to the system by using the Rule Writing Utility (RWU). In order to add a rule, the programmer creates an instance of the class RULE_INFORMATION. They can do this manually or with option 8 of CAAP's main menu. Both methods create objects containing several slots: one set of slots is created listing the input variables and another set is created listing the output variables of the rule that is represented by the instance. The input variables can be restricted to be equal, not equal, less than, greater than, less than or equal to, or greater than or equal to some other value. The text of the calculation that will be executed by the actual rule is also stored in a slot of the instance of RULE_INFORMATION. Therefore, the programmer needs only enter two sets of variables (input and output) and a string representing a calculation (and some housekeeping information). The rest is handled by the RWU.

Before system operation beings, the RWU code creates the class RULE_INFORMATION and the rule "make_rules." "Make_rules" creates expert system rules and places them into the Rule Base based on the existing instances of RULE_INFORMATION. It also adds all of the constraint checking that is necessary for proper CAAP operation. Such a utility could be useful in other Expert Systems that involve the same type of input and output for each rule. As previously mentioned, such code recycling opportunities are an important aspect of CAAP.

Once "make_rules" has fired, each rule is represented in two places: one sense of the rule exists in the Rule Base as an actual expert system rule. Another sense of the rule exists in the Object Oriented Database as an instance of the class RULE_INFORMATION. This second representation of the rule is used by the assignment routine to satisfy its need to know how variables depend on each other. Assignment routine operation and the way the routine uses variable dependency information are described in the *Consistency Maintenance* section.

The present method of double representation is more efficient than what was possible with CLIPS 5.1. Previously, if a programmer wanted to add a rule with five inputs and five outputs, they would have to check that 5 X 5 or twenty-five separate variable dependencies were included in the dependency functions (in addition to the constraint checking information). In Version 2.0, the programmer simply needs to list the five inputs and the five outputs of the rule at the time of RULE_INFORMATION instance creation. This brings CAAP a large step closer towards decreasing the future programmer's work load. Additions to the Rule Base can now be generated more easily and more quickly than before.

The Core Program

CAAP's Core Program has been given the basic abilities needed to design an airplane. At the time of this writing, the Core Program still requires the addition of large amounts of data in the form of Toolbox files in order to be able to design an airplane. Nonetheless, the full functionality of the Core Program has been implemented. The main menu of the package appears in Figure 3. Descriptive English equivalents are used at every point throughout the user interface. A lot of attention was focused on making CAAP user friendly in order not to loose any potential users due to the newness of the program.

The user is presently allowed to create airplanes, modify the specifications the airplane is designed to, and change the airplane's describing variable values. The user may save and load airplanes from disk. The user is also given complete access to the CAAP database. Designers are allowed to look at the variable values that represent an airplane, either individually or in a summary sheet format. They may also look at the dependencies that exist within the CAAP database. This valuable tool would tell the user, for example, that the aspect ratio depends on the wing area and the wing span for its values (reverse dependencies), and that when the wing span is changed, the aspect ratio will have to be recalculated (forward dependencies).

The user is allowed to see a list of all variables which have been defined to CAAP and which are not used in any of the presently loaded rules. This can help designers load the appropriate Toolboxes or add rules where necessary. Users can write rules during run-time and add them to Toolbox files. This feature allows for simple expansion of CAAP by individual users in the future, and, combined with the Toolbox manipulation functions, has allowed CAAP to become a self-modifiable program. With this ability, CAAP can evolve to meet individuals needs as they create and change Toolbox files and the rules that populate them.

The user is given some aesthetic controls over CAAP output. The user can look at CAAP's internal variables representations. Toolboxes can be created, deleted, and loaded during run-time. Users can view which Toolboxes have been loaded into the CAAP database as well as choose which Toolboxes to load each time CAAP starts up. The dynamic use of Toolbox files presents some interesting situations. The files can act as a medium for knowledge exchange in the future. For example, Joe can design a plane with the Toolboxes he has built over time. He can then add Mary's rules to his personal Toolbox, and redesign his plane in order to discover how Mary's know-how can improve his model. Such an interactive exchange of information could be very useful, especially in an teaching environment.

Consistency Maintenance and The Availability of Several Routines to Calculate One Variable

There are several different methods available to estimate almost any of the parameters used in airplane design. Different sources will quote different methods, each with its own result. A consistent method for routine execution is needed. When there is more than one equation or routine available to calculate a given parameter, CAAP will select the most advanced one for which all of the input variables have been determined. For example, the airplane drag coefficient can be calculated using equation (1) or with a drag build-up.

$$C_{L_\alpha} = C_{l_\alpha} \frac{A}{A + [2(A+4)/(A+2)]} \qquad (1)$$

C_{L_α} = lift curve slope for a finite lifting surface

C_{l_α} = section lift curve slope

A = aspect ratio

If the components of the plane have been defined, the latter, more advanced drag estimation method will be used. If the components have not yet been defined, the former, simpler method will be used. Importantly, once the components have been defined, the LHS of the drag build-up rule will be satisfied and CAAP will *recalculate* the drag based on the more advanced drag build-up. All calculations are based on the most advanced routine available, due to the rule based programming implementation chosen for CAAP.

Rule "advancedness" will be represented by a priority associated with each rule. This priority is stored in a "rulepriority" function in CAAP's core program. It is presently used to ensure that the

Inference Engine sees more "advanced" rules more quickly than it sees more primitive rules. Rulepriority is used by the RWU to set rule saliences during system initialization. This procedure improves program efficiency by decreasing the likelihood that a particular variable will be calculated many times by successively more advanced routines when a very advanced one could have done the job originally. Rule prioritization also allows the user to be confident that the crude initial estimates used in the Initial Sizing Toolbox will not be used in the final configuration. As soon as the airplane begins to take shape, the Initial Sizing Toolbox's estimates will be replaced with more advanced values[1].

If two methods are similarly "advanced", one method will be chosen over the other arbitrarily, but not randomly. If the designer has a preference as to which method is used by CAAP, he or she can specify this to the package. The "rulepriority" function alleviates the need for addressing the situation when two expert recommendations agree. Either they will have different priorities, or their location on the agenda will determine which is fired.

Consistency Maintenance and Parameter Modifications

During the design process, configurations are created and analyzed. If the analysis shows a given configuration to be inadequate in some way, the rules within the Expert run-time module will modify one of the design parameters of the given configuration, in effect creating a new configuration. Until the effects of this single modified parameter have been propagated throughout the airplane, the configuration will be inconsistent. In another scenario, an advanced routine might recalculate a design parameter previously calculated by a more primitive routine. Again, until the effects of this change have been propagated throughout the system, an inconsistent configuration will exist. The solution to this problem follows.

Consistency maintenance will be accomplished in two ways. When a rule within the Expert run-time module modifies an airplane design parameter, it will have to do so in a "responsible" manner. For example, suppose the Expert rule, for an "expert" reason, decides that the aspect ratio of the wing needs to be changed. If it simply changes the aspect ratio, the span and/or wing area will be inconsistent. Therefore, the Expert rule will have to *also* change the span or the wing area. The rule could, for example, adjust the aspect ratio while keeping the wing area constant. In other words, the Expert rule will have to look at the input variables that determine the value of the design parameter and modify them so that they are consistent with the new value of the changed variable.

The second consistency maintenance procedure will be based on computational paths. Figure 4 presents a diagram of a hypothetical set of computational paths. Each box on the diagram represents a variable. The directed connections represent the dependency of a variable on the value of other variables.

Suppose that the variable in the shaded box has just been redefined, perhaps by a rule from the Expert run-time module or by an advanced estimation routine. The value of every box "downstream" of the shaded box is now inconsistent. The "downstream" variables are represented by the presence of an "X" in the variable box. The "downstream" variables need to be recalculated as if they had never been determined in the first place. Each rule will have access to the list of variables which depend on the variable in the shaded box (i.e. X_1 in this example). This list is stored in instances of the RULE_INFORMATION class, introduced in the *Rule Writing Utility*

---

[1]This does not necessarily have to occur, if one is not careful. It would be possible for the airplane to be presented as a final product without enough of it having been calculated to replace the Initial Sizing Toolbox's estimates. This would be an absurd situation, and it would result in problems. CAAP will not present a plane to the user unless a minimum set of parameters have been calculated to a sufficient level of "advancedness". This way, no Initial Sizing Toolbox estimates will make their way to the user.

section. The rule will erase, or undefine[2], all of the variables that depend on the changed variable (i.e. it will undefine X_1). The Toolbox will then undefine all of the variables that depended on those variables, and so on until the there are no more dependent variables to undefine (i.e. X_2, X_3, ... , X_6). This systematic undefining is called the "downstream" erasure procedure. It has been coded as part of the "assign" routine that is used for all variable assignments. Every rule must use the "assign" routine. After a "downstream" erasure, the other rules in CAAP will automatically recalculate the undefined "downstream" variables. This will occur since the LHS of CAAP rules is satisfied when the input variables for the rule are undefined.

A problem with the method of consistency maintenance presented in Figure 4 will arise if any loops exist within the computational paths. A discussion of this problem is beyond the scope of this paper, and the problem has only been partially solved. A full solution to the "Loop Problem" is one of the major remaining issues facing CAAP.

Practical Limitations

The future of CAAP will focus on three different areas: the core program, the Toolboxes, and the user interface. The essentials of the core program have been entirely written. Some extra functionality has also been added to the program. Nonetheless, there is always room for improvement and CAAP is by no means complete. Among the pieces of code not yet written is a numerical optimizer. Such code could provide CAAP with a way to make "expert" recommendations when no rules from the Expert run-time module apply to a given configuration. If no rules exist to help, CAAP could turn to numerical optimization methods in order to determine what changes to make to a configuration in order to make it meet all user and FAR requirements. A simultaneous equation solver could significantly facilitate solving the airplane design problem.

The Toolbox files need to receive a significant amount of data. Proof of study Toolbox files have been implemented and successfully tested, but there remains a lot of data to input in order to fully design an airplane. The graphical user interface ran into difficulties associated with system level Macintosh programming. Finding an alternative to friendly user interactions will be a priority for CAAP in the future.

The first category of plane that CAAP should be able to completely design will be the light, general aviation, non-acrobatic, single engine aircraft. The graphics for displaying the airplane are next on the implementation list. Eventually, trend studies and increased user involvement in the design process could be added. For example, if the user wished CAAP to produce several final designs instead of one, this could be done. If the user wished to watch CAAP fire one rule at a time, this could be done. A utility could be added to allow users to see which rules are firing at any given time. This would provide the user with a better "feel" for how the package is going about designing their airplane.

Conclusion

A firm theoretical foundation has been developed for CAAP. The problem of designing an airplane has been laid out and implemented using rule based, object oriented, and procedural programming paradigms. Rule based programming enables CAAP to capture expert knowledge and to mimic the *potentially iterative* nature of preliminary airplane design. Object oriented programming handles the voluminous, complex, and hierarchically arranged data produced during airplane design.

---

[2]CLIPS 6.0 no longer supports undefined slot values. It is necessary to have such reserved values for airplane variables that may take on a range of values. In order to satisfy the LHS's of any of the rules, the LHS's must contain tests for variables to see if they have not yet been calculated, that is that they are undefined. A typical undefined value is -1e-30 for a floating point variable.

Procedural programming is used to implement the actual analysis routines necessary for engineering design. CAAP has realized core program implementation and proof-of-concept Toolbox file creation and test. CAAP can begin designing airplanes and awaits the addition of more data in order to be able to complete the design process. CAAP is still in the developmental phase.

Figures

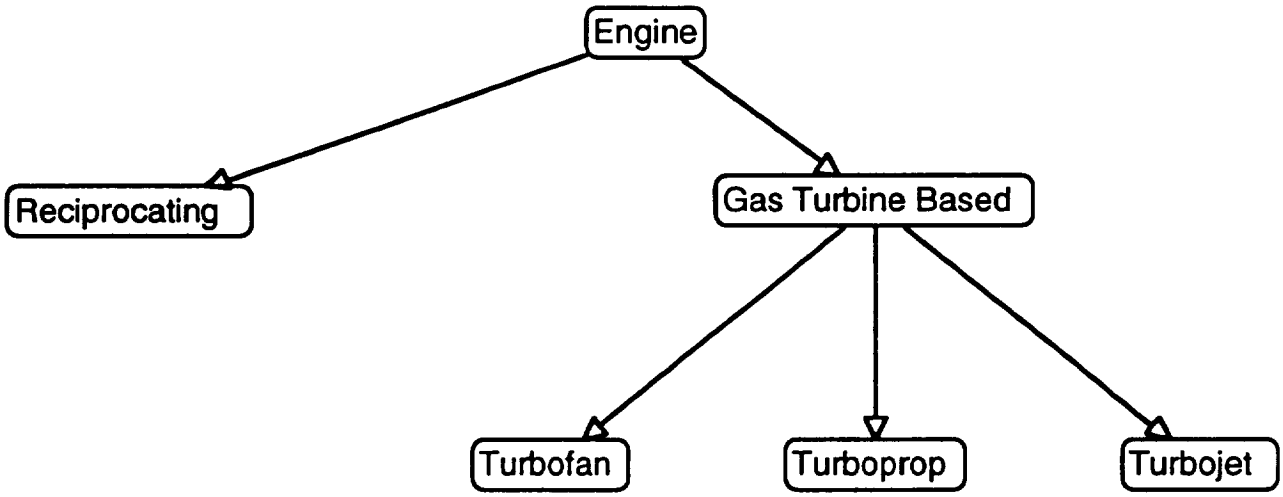

Figure 1 - Engine Classifications in CAAP

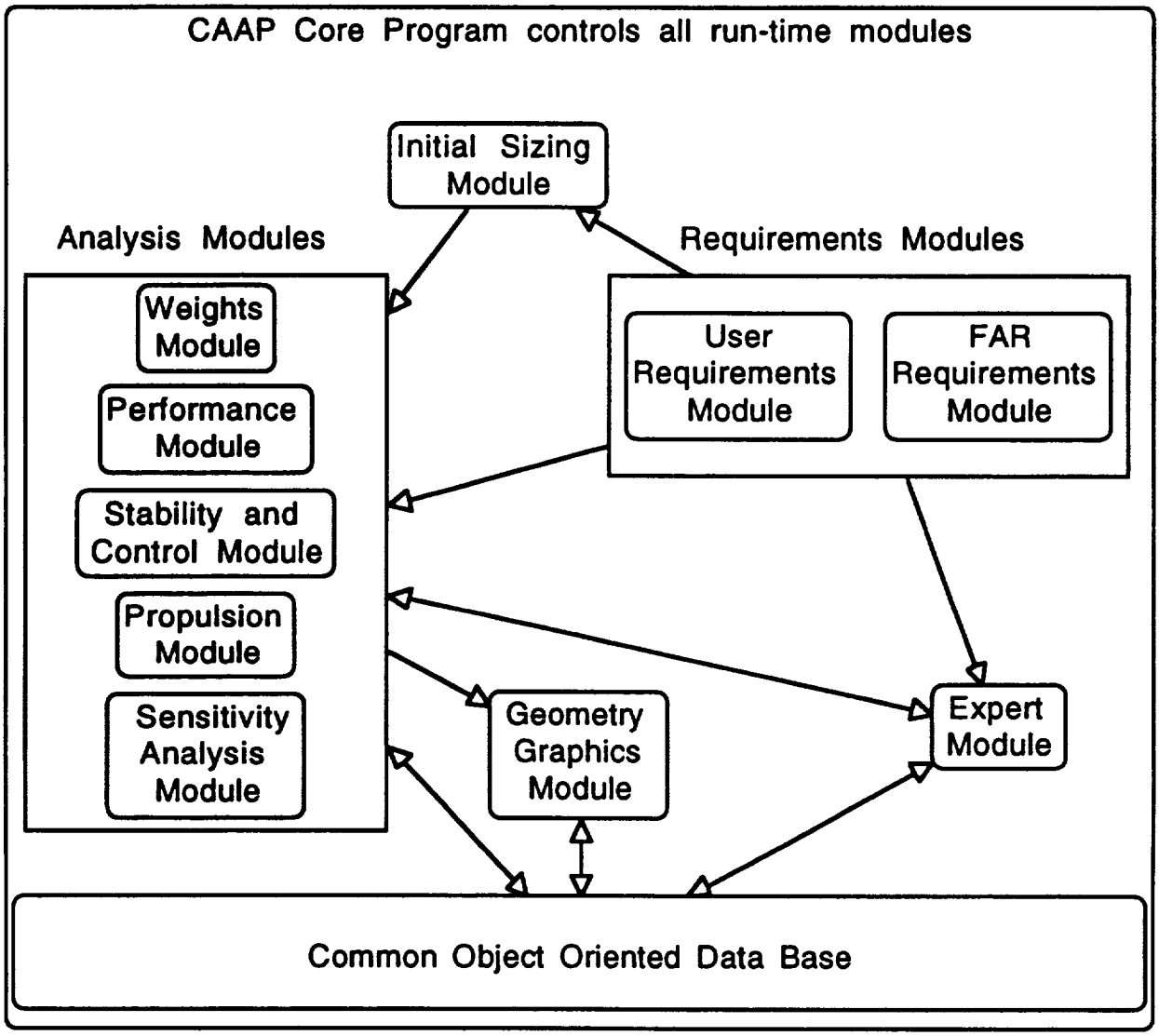

Figure 2 - System Architecture

Main Menu
---
1) Create an airplane
2) Modify requirements or plane
3) Analyze airplanes
4) Save an airplane to disk
5) Load an airplane from disk
6) Look at an airplane
7) Look at slot dependencies
8) Add a rule to the database
9) Look at internal variable representation
10) Manipulate Toolboxes
11) Miscellaneous
12) Quit

Please choose an option (0 to reprint menu)>

Figure 3 - CAAP Main Menu

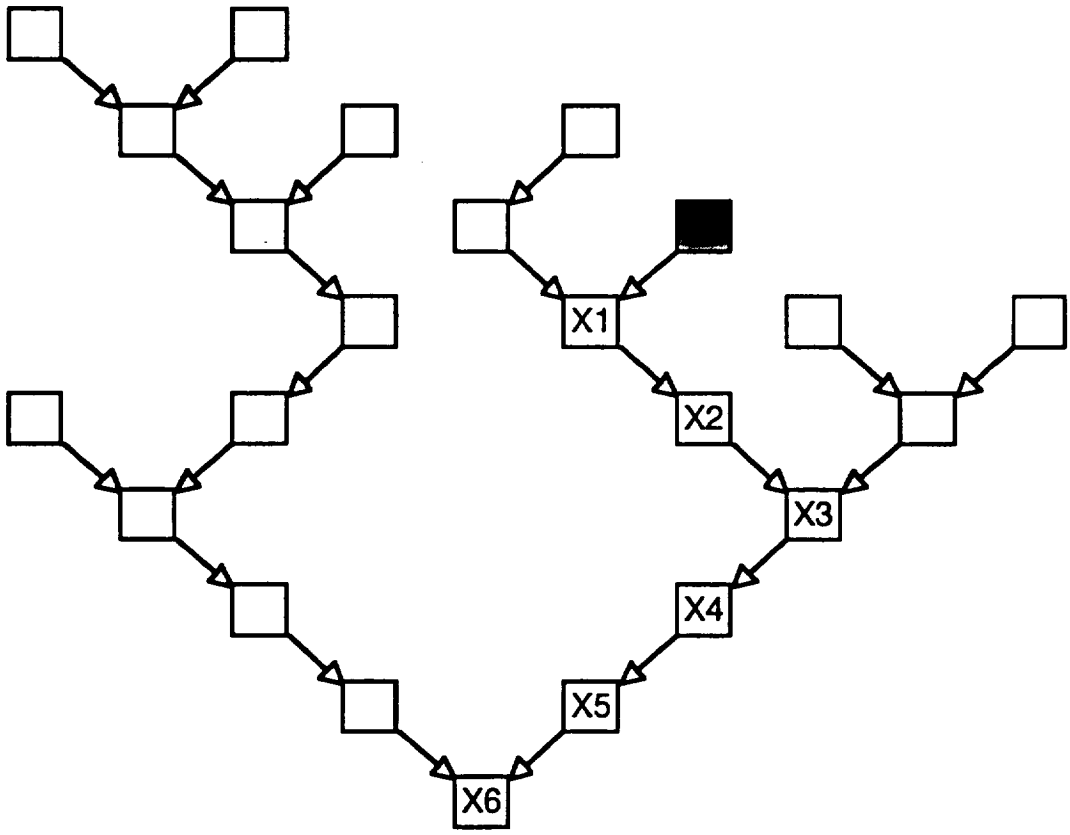

Figure 4 - Consistency Maintenance Example

References

1. Newman, D., and K. Stanzione. Aircraft Configuration Design Code Proof-Of-Concept: Design of the Crewstation Subsystem. Proc. of the AIAA Aircraft Design Systems and Operations Meeting. 23-25 Sept. 1991. Baltimore: AIAA paper No. 91-3097, 1991.

2. Fulton, R. E., and Yeh Chao-pin. Managing Engineering Design Information. Proc. of the AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference. 7-9 Sept. 1988. Atlanta: AIAA paper No. 88-4452, 1988.

3. Roskam, Jan, and Seyyed Malaek. "Automated Aircraft Configuration Design and Analysis." SAE Technical Paper Series No. 891072 (1989): General Aviation Aircraft Meeting & Exposition (Wichita, KS), 1989.

4. Kroo, I., and M. Takai. A Quasi-Procedural, Knowledge-Based System for Aircraft Design. Proc. of the AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting. 7-9 Sept. 1988. Atlanta: AIAA paper No. 88-4428, 1988.