FUTURE DEVELOPMENTS IN MODERN ENVIRONMENTS FOR CADCS

James H. Taylor, Magnus Rimvall and Hunt A. Sutherland
Control Systems Laboratory, GE Corporate Research and Development,
PO Box 8, Schenectady, New York 12301 USA

Abstract. Recent and future efforts at GE to develop modern environments for CADCS are discussed. The basic elements of these systems are:

- a User Interface which combines a "point-and-click" menu- and forms-driven interface with other access modes for the more experienced user,
- a Data-Base Manager organized in terms of Projects, Models and corresponding Results and other related data elements and including version control,
- an Expert System Shell, which performs routine higher-level CACE tasks, and
- a data-driven Supervisor that integrates the above elements with existing CACE packages for linear and nonlinear simulation, analysis and design.

As is usually the case, it has been learned that much more can be done to provide a fully supportive environment for controls engineering, and it has also become clear that certain things might better be done differently. This presentation will focus on such areas, especially on generic issues that can be applied to other CADCS systems.

Keywords. Computer-aided system design; computer interfaces for CAD; data-base management systems; expert systems; control engineering applications of computers.

1. INTRODUCTION

1.1. Motivation

Control system performance requirements are continually becoming more stringent. This trend fuels the demand for the use of advanced controls technology, which in turn translates into the current growing need for advanced computer-aided control engineering (CACE) software. This has given impetus to rapid strides made world-wide in CACE software development and usage, and has strongly motivated the GE MEAD Project.

The development of CACE software started several decades ago with the production of routines to perform specific functions that had previously been done manually (e.g., root locus, Bode analysis). In the 1970s the emphasis shifted to the "packaging" of routines to integrate them, share common data structures, and broaden their scope. In this second phase attention was also given to creating numerically robust algorithms; libraries such as LINPACK and EISPACK began to supplant "home-brew" algorithms that had been used before. More recently, the focus has shifted to further broadening of functionality (e.g., block-diagram interfaces for model building, automatic code generation) and to improving the overall environment.

Broadening and improving the CACE environment is the concern of this presentation. The areas to be addressed include the user interface, a "supervisor" to serve as the integrator for broad CACE functionality, and support facilities for data-base management and expert aiding. Much of the description of needs and considerations that follows is based on the GE MEAD Controls Environment (MEAD = Multi-disciplinary Expert-aided Analysis and Design; Taylor and McKeehen, 1989), and the experience gained in the course of the GE MEAD Project.

1.2. GE MEAD Controls Environment Overview

The GE MEAD Controls Environment (Taylor and colleagues, 1989, 1990) has been designed to address the support and environmental issues outlined above while taking maximum advantage of existing software modules. This software is the successor to a mature "production" environment prepared for the US Air Force, also called MEAD (USAF MEAD; Taylor and McKeehen, 1989; Hummel and Taylor, 1989). The basic elements of MEAD systems are:

- a point-and-click menu- and forms-driven User Interface (Rimvall and colleagues, 1989) that supports all basic CACE activity plus other access modes for the more experienced user (command-driven modes and a macro facility),
- a Data-Base Manager (Taylor, Nieh and Mroz, 1988) which organizes the user's work into Projects which are populated with models, results, and other related data elements,
- an Expert System Shell, which is programmed to perform routine higher-level CACE tasks that are beyond the capabilities of standard packages and require a level of heuristic decision-making or iteration (Taylor, 1988; this is only working in USAF MEAD), and
- a data-driven Supervisor (Rimvall and Taylor, 1991) that provides a shell for existing CACE packages for linear and nonlinear simulation, analysis and design, and interfaces with the Data-Base Manager and Expert System.

The resulting software architecture is depicted in Fig. 1. The CACE tools ("core packages") now include the PRO-MATLAB™ package for linear analysis and design, and the SIMNON™ package for nonlinear simulation, equilibrium determination, and linearization. Other modules are also based on existing software: the user interface was built using the GE Computer / Human Interface Development Environment (CHIDE; Lohr, 1989) which rests on the ROSE™ data-base manager; the MEAD data-base manager (DBM) uses ROSE and the DEC™ Code Management System (CMS - DEC, 1989) for version control; and the expert system uses the GE Delphi™ shell which rests on VAX™ Lisp. The supervisor and the front-end of the DBM are coded in the Ada™ language.

In the course of developing, testing, and evaluating USAF and GE MEAD, we believe that substantial progress has been made in supporting the controls engineer in a number of areas previously given little or no attention. In particular, the higher-level user interface, data-base management scheme, and expert-aiding are noteworthy new contributions. The MEAD User Interface is much more "user-friendly" compared with those of the underlying packages which have rigid command-driven interfaces. The data-base management capabilities solve important problems associated with version control of the user's models and tracking related files such as results - in essence, every element in the user's MEAD data base is fully documented as to how it was obtained. The MEAD expert system adds yet another area of support, which at this point has not been used to full advantage.

As is usually the case, we have also learned that much more can be done, and that certain things might better be done differently. The focus of this presentation will be on those areas where the MEAD concept and implementations can be modified, extended and improved. Particular emphasis is placed on generic issues that may have application to other modern CADCS environments. First, however, a short survey of the most prominent state-of-the-art CACE packages is provided to set the context for the discussions that follow.

1.3. ECSTASY Overview

The Environment for Control System Theory, Analysis and Synthesis (ECSTASY) infrastructure has been developed under the sponsorship of the UK's SERC Control and Instrumentation Sub-Committee (Munro, 1990). The conceptual design architecture of this software, depicted in Fig. 2, is quite similar to that of MEAD, being broader in some areas (e.g., report generation) and less ambitious in others (e.g., expert-aiding, data-base management). This environment is based on the same strategy of providing a "shell" for a variety of standard/commercial packages, as indicated in that figure. The design philosophy is also much the same, i.e., to focus on providing high-level support suitable for both academic and industrial controls practitioners and leave functional/numerical/algorithmic concerns to the developers of the underlying core packages. Being more recently conceived, ECSTASY takes advantage of a more up-to-date workstation platform compared with MEAD; nonetheless, the look and feel of the user interface has a number of similarities. The decision to launch the ECSTASY project was made in 1986; the system is presently in the prototype or "embryo software" stage (Munro, 1990). In many respects, ECSTASY and MEAD are the most similar of the modern controls engineering software environments currently reaching maturity.

The core packages in ECSTASY cover about the same functionality as MEAD in terms of nonlinear modeling and simulation through linear analysis and design. PRO-MATLAB and ACSL are the most prominent commercial packages used for this purpose; these are augmented by CLADP (Edmunds, 1979) and CSS (Edmunds, 1988), two British packages with great strength in the "UK School" of frequency-domain design; CSS is reviewed below.

1.4. CSS Overview

The CSS package (Edmunds, 1988) has a complete coverage of the standard repertoire of nonlinear modeling and simulation and linear analysis and design. The main emphasis is on generalized frequency-domain methods for linear multivariable systems, parameter estimation for single-input/single-output systems, and block-diagram-oriented nonlinear modeling. The linear multivariable systems capabilities of CSS are a quite direct reimplementation and extension of those in CLADP. Parameter estimation is embedded in a unique framework that directly translates parameter uncertainty into the frequency domain, to serve as the basis for robust control system design. The nonlinear modeling interface provides a front-end to ACSL, which then carries out the nonlinear simulation and analysis functions (e.g., linearization).

The UI of CSS has been substantially modernized compared with that of CLADP. A block-diagram modeling environment is provided, and user interaction modes have been extended to include forms input, panel input, and question & answer styles in addition to a command-driven interface. As in CLADP, the present context of the user's activity determines what functionality is available; the context-dependent help system is designed to guide the user in this respect. In essence, the help system is an on-line Reference Manual for CSS.

1.5. Matrix Environments

The so-called "matrix environments" all trace their origins to MATLAB, a freeware "MATrix LABoratory" (Moler, 1980). This interactive linear-algebra workbench was not designed with controls engineers in mind, but the powerful expressiveness of the MATLAB command interface and the reliability of the underlying algorithms (LINPACK and EISPACK) made it well suited for control purposes. Several commercial vendors exploited these

---

™ MATLAB is a registered trademark of The MathWorks, South Natick, Massachusetts; SIMNON is a trademark of Lund University, Lund, Sweden; VAX, DEC is a registered trademarks of Digital Equipment Corp., Maynard, Massachusetts; ROSE is a trademark of Martin Hardwick, RPI, Troy, New York; Delphi is a trademark of GE; and Ada is a registered trademark of the U. S. Government, Ada Joint Program Office.

features, and during the first half of the 1980's the three packages MATRIX$_X$™, Ctrl-C™ and PRO-MATLAB set a *de facto* standard for linear CACE tools. These packages all extended MATLAB with numerous controls analysis and design algorithms, an extendable command language interface, and powerful interactive line graphics.

Several research projects during the mid 1980's advanced the concepts of MATLAB within a CACE framework, resulting in packages such as IMPACT (Rimvall and Cellier, 1985), EAGLES (1986), and BLAISE (Delebeque and Steer, 1986). A major thrust of these projects was to extend the matrix-based systems with controls-oriented data-structures for state-space and frequency-domain system and signal representations. Much of this work influenced the design of the Xmath™ program to be released by Integrated Systems in 1991.

The three dominant matrix environments can today be acquired with companion graphical block-diagram editors for hierarchical nonlinear modeling and simulation. MATRIX$_X$ is paired with the SystemBuild™ package, Ctrl-C is accompanied by the Model-C™ simulator, and PRO-MATLAB can be combined with the SIMULAB™ environment. These are all coupled through linearization algorithms and through the sharing of common data (such as the A,B,C,D matrices of a linear subsystem). SystemBuild, Model-C and Protoblock™ were compared and evaluated by Spang and coworkers (1990) for their modeling capabilities and user friendliness (SIMULAB was not available at that time); they concluded that SystemBuild and Model-C are basically the same, although SystemBuild is more mature and currently has more block types. Both of these tools are comparable to other modern graphical drawing tools found for example on the Macintosh™ and Sun™ computers. Protoblock takes a different approach to user interaction, with less use of mouse and cursor control, and thus does not fully exploit the interactive graphics capabilities of modern workstations. Some unique characteristics of these three product pairs will be discussed in the next subsections.

1.6. PRO-MATLAB / SIMULAB Overview

PRO-MATLAB implements numerical algorithms similar to those found in the original MATLAB program of Moler. However, it is completely rewritten in C, making it substantially more maintainable and efficient. PRO-MATLAB introduced the concept of "toolboxes", which can be bought as extensions to the base product or developed by the end-user. A toolbox is a collection of algorithms programmed in the proprietary PRO-MATLAB command language; these algorithms may be built on the base numerical algorithms of the PRO-MATLAB kernel and/or algorithms of other toolboxes. There are now 8 vendor-supported toolboxes ranging from classical controls, system identification, and robust control to signal processing and chemometrics. In addition, many individuals and organizations have implemented their own, domain-specific toolboxes - the work of Minto, Chow and Beseler (1989) is a noteworthy example.

SIMULAB is the newest block-diagram editor companion. It is based on the Motif™ standard, and thus shares the look-and-feel of other Motif applications. Nonlinear simulation, equilibriation, and linearization operations are available. SIMULAB runs as part of PRO-MATLAB, simplifying the exchange of data between the two programs.

1.7. Ctrl-C / Model-C Overview

Ctrl-C is a true extension of Moler's original MATLAB code. Better parameter handling and modern graphics have been introduced together with a large set of control-oriented algorithms.

Model-C is a graphical block-diagram editor and nonlinear simulator which also supports equilibriation and linearization. It has a look and feel similar to that of Macintosh applications, with powerful graphical editing capabilities. Model-C is strongly tied to Ctrl-C, thus all state-space systems and other non-scalar numerical data must be defined in Ctrl-C before they can be used in Model-C.

1.8. MATRIX$_X$ / SystemBuild / Xmath Overview

MATRIX$_X$ is very similar to Ctrl-C in its appearance and implementation. Its nonlinear companion package SystemBuild features a hierarchical modeling environment, a large set of building blocks, and a state-transition module for modeling discrete events. SystemBuild can linearize models, but cannot find equilibria. Another companion product, Autocode™, can automatically translate control block diagrams into FORTRAN, C or Ada.

Xmath is the newest matrix environment (Floyd et al, 1991), and the first commercial package to break away from some of the inherent constraints of the original MATLAB program:

- Xmath features control-related data structures in an object-oriented fashion. In addition to the complex matrix, which is the only structure explicitly supported in the classical matrix environments, Xmath provides data structures for state-space models, transfer-function representations (rational functions in root or coefficient form), parameter-dependent descriptions (e.g. tables) and special matrices (e.g. triangular and diagonal matrices).
- Data in Xmath may be grouped into partitions in a way similar to files being grouped in directories under an operating system.
- Operations are overloaded in a versatile and yet unambiguous manner. For example, the operator '*' is overloaded as follows:

matr1*matr2	matrix multiplication
poly1*poly2	convolution of two polynomials
syst1*syst2	state-space systems in series
syst*timehist	time-domain simulation

™ MATRIX$_X$, SystemBuild, Xmath and AutoCode are registered trademarks of Integrated Systems, Inc.; Ctrl-C and Model-C are registered trademarks of Systems Control Technology, Inc.; SIMULAB is a trademark of The MathWorks, Inc.; and Protoblock is a trademark of Grumman Aircraft. Macintosh is a trademark of Apple Computer, Inc., Sun Workstation is a trademark of Sun Microsystems, Inc., and Motif is a trademark of The Open Software Foundation.

In addition to these fundamental extensions to the matrix paradigm, Xmath introduces a graphical user interface based on Motif, powerful plotting capabilities, and programmatic user extendability.

2. USER INTERFACE IMPROVEMENTS AND EXTENSIONS

The present generation of CACE user interfaces, as typified by MEAD, combines the simplicity of modern graphical interfaces, using drop-down menus, forms, and "point-and-click" techniques, with the flexibility of command- and macro-interfaces, as found in the MATLAB family. This gives both the novice and expert user a powerful and yet fully manageable access to the packages. This is a particularly critical issue for CACE applications, since many users are quite computer-literate and demanding. Thus a user interface for CACE must provide a *carefully-designed* graphical interface, as well as a richer set of interaction modes, so that an experienced user can perform operations effectively and efficiently. The MEAD User Interface (UI), for example, is designed to facilitate access to the CACE capabilities of MEAD for a heterogeneous industrial user group.

The MEAD UI's multifaceted design:

- supports users with widely different levels of familiarity with the environment,
- provides a single tool both for the engineer confronted with an occasional control problem and the expert control engineer using the most sophisticated control methods on a daily basis,
- provides access to different control packages in a uniform fashion,
- provides uniform interactions for similar but disjoint tasks, and
- helps to manage the data for very large projects in a reliable manner.

To do this, there are four modes of interaction:

- The primary UI mode, which is a menu- and forms-based point-and-click style interface, is the most suitable for beginning or once-in-awhile users. Moreover, the expert user is also well served by its efficient operation, as many individual point-and-click action buttons launch more powerful commands than the ones available in the underlying packages. For example, a 'connect' action in the MEAD model-building environment may translate into more than a dozen commands which are sent to the underlying linear analysis package. However, the functionality is definitely prescribed in this mode, so the expert user may not be able to achieve all desired results.
- The MEAD command mode allows the user to directly enter supervisor-level commands (see Section 5). This mode is primarily intended for the expert user wishing to use the full power of the Supervisor's command language (which includes conditional statements, loops, etc.). Although most commands on this level are available via the more friendly menu/forms mode as well, the ability to combine and structure commands freely can expedite tasks for the expert.
- "Package Mode" gives the user the option of entering arbitrary commands directly to the underlying packages, using their native command syntax. This requires the user to know how to operate the underlying package in stand-alone fashion; it is intended to be used when the exact desired functionality is not available through the use of MEAD commands. Commands are entered via the supervisor, so full data-base management capabilities are available on this level (Taylor et al, 1990).
- Macro Mode facilitates defining sequences of commands for repeated execution. These commands may be captured in script form during normal operation of MEAD, or they may be entered using a regular text editor. Macros may contain both MEAD and package commands, they may be edited for customizing, and they are automatically loaded into a selection form for easy access in menu mode.

2.1. Improved UI Look and Feel

In designing the MEAD UI, two basic principles have been *consistency* and *agility*. The MEAD graphical operating environment allows the user to perform all controls-related operations in a very consistent manner over mouse-operated menus and forms. A menu hierarchy is used to group related operations together into domains familiar to control engineers (Taylor et al, 1990). The menu tree hierarchy is limited to two or three levels for quick access to all domains. At the bottom of the menu tree, selection and action forms are used to give a highly interactive execution of most operations. A menu-tree path and action form are portrayed in Fig. 3, to illustrate these features.

A UI with these attributes is very complex and must manage a diversity of inter-related functions. As is true in the MEAD UI, the selections available and the context of the UI is highly dependent upon the past activity of the user. For this reason, the UI must retain past history and provide a safety net around actions which may be dangerous or time-wasting. For example, before any CACE operation can be done one must select and load a model - therefore, most of the MEAD menu tree is inaccessible until this is done. At a lower level, it would generally be foolish to perform a simulation without either defining input signals or initial conditions, so the MEAD UI logic also guards against this. Many of the refinements identified after initial evaluation of the MEAD UI involved improving the logic in the UI in this way; the underlying principle is that the UI should always make these issues transparent to the user.

Another consideration is the consistency among *all* applications found on the user's workstation, including non-CACE tools. First, basic software standards such as the X Windows System™ should be adopted for this reason. Further measures within the X Window framework include:

™ X Windows System is a trademark of MIT; Open Look is a trademark of AT&T Company.

- Choose a style for forms and menus which is accepted industry wide, so that the look and feel of the CACE application follows accepted style guidelines, as offered for example with Motif or Open Look™. This will ensure that less time is required for the user to become familiar with standard operations.
- Present options in menus or forms so that their applicability is understood before selection. An option should be "grayed-out" or invisible if not valid, and options should appear consistently in the same location. Also, options which proceed on to other menus or forms should be distinguished from those which immediately cause an action.
- Provide easy access to auxiliary information, regardless of which menu or form is active, without disturbing the context of the UI. In MEAD, for example, access is provided to on-line helps, status displays, notes, and, most importantly, to data-base browsing.
- Provide constant feedback to the user as operations are performed. This is particularly needed during tasks which may take tens of seconds or longer, such as loading a model from the data-base or performing a simulation. A protocol recently introduced into the MEAD UI now enables the Supervisor or Expert System to report intermediate status through the UI during lengthy operations.
- Manage the setup and initialization of auxiliary programs, such as plot utilities, text editors, or block diagram editors. Invoking these utilities can be done manually, but this is generally not required, since the object of interest in the data-base and the operations appropriate to that element are usually known in advance, so this step can thus be performed automatically.
- Provide context-directed helps which key upon the currently active form or menu.

Most of the above recommendations were implemented in MEAD, although without the advantages of using X Windows and Motif or Open Look.

2.2. *Improved UI Functionality*

The improvements previously outlined relate to user effectiveness within the existing environment. Extensions can also be made to increase the *functional capabilities* of the UI. These include:

- a *smart editor* for entry and execution of commands and macros,
- *user customization* of the menus and forms,
- tools for *adding new rule bases*, and
- *interactive manipulation and viewing* of data-base objects.

Smart editor: A smart editor would provide a facility for the interactive entry of MEAD and package commands, and assist with the construction and execution of macros. This would combine a language-sensitive editor with an interactive mode for immediate command execution. The commands could be built from scratch, or taken from a buffer which contains the sequence of commands previously executed. (The present version of the MEAD UI already provides access to the previously executed commands, via a standard text editor.) The smart editor would contain knowledge about the command syntax, providing either menus of available commands to select from, or command completion with prompting for parameters associated with a command. It would be loaded from the same command-definition files used by the data-driven supervisor, thus ensuring consistency.

User Customization: A toolbox of standard "dialog-boxes" and forms would enable the user to customize the functionality provided in the menus and forms, by installing custom macros. An action form, similar to that shown in Fig. 3, could be automatically instantiated from the interface definition specified in the macro, thus achieving the seamless integration of user-developed macros as additional functions appearing in the menu tree. (The present MEAD UI automatically provides a *selection* form for user-developed macros, but this form is not integrated with the main menus.)

Adding new rule bases: To support this, the UI must provide a more general framework for queries and responses, to facilitate the interaction of the user with the expert system. (USAF MEAD supports some simple communication protocols; however more generality is needed.) The required UI extensions should prove to be minor in addition to those outlined in the item above.

Dynamic access to data-base objects: An example of the concept is shown in Fig. 4 and has the notion of attaching various windows with configurable views of objects in the data-base. The windows would then provide either "direct manipulation" or a continuously updated view of the object. As shown in this example, the user has developed a system model containing a compensator with some design parameters to be tuned, and chooses to view both a root-locus diagram and the step response of the system as the design parameters are varied. Starting from the block diagram of the system, three additional windows are opened. One window contains a list of design parameters to be varied, perhaps taken from the compensator block contained in the model data-base. The other two windows display time-history and root-locus results. Then, as the user types in new values in the design parameter window, the plot windows would automatically update.

Such windows must have dynamic access to data-base objects rather than the static access now provided by the MEAD UI. As shown in this example, data-base objects must be simultaneously changed and viewed; the present MEAD UI does not permit this. This concept has great potential to improve the productivity of the controls designer and is currently an active research interest, such as the demonstration in Ravn, 1989.

3. DATA-BASE MANAGER IMPROVEMENTS AND EXTENSIONS

It is fair to say that little was done in terms of providing engineering data-base-management support for CACE prior to MEAD. A user's models and results simply accumulated in the workspace (e.g., in a subdirectory under VAX VMS), and it was up

to the engineer to perform version control, to relate results to specific model instances and conditions, to relate linearized models to the "parent" nonlinear model and operating point, and so on. To rectify this situation, data-base-management requirements from the user's perspective were developed under the USAF MEAD project (Taylor, Nieh and Mroz, 1988).

The CACE user's data base is traditionally but informally organized in the hierarchy *Projects, Models, Components,* and *Results*. The user often sets up a workspace for each project (e.g., Project = *GE_654*), develops models (e.g., Model = *Turbine*) which are comprised of components (e.g., Component 1 = *Stator*, Component 2 = *Rotor*, Component 3 = *Fuel_injector*, ...), and which are used to generate various results (e.g., simulation time-histories, linearizations). This has been accommodated directly in the MEAD data-base tree, as illustrated in Fig. 5.

Within this framework, the MEAD Data-Base Manager (DBM) was designed to address the problems of maintaining the integrity and documentability of the user's models and analysis and design results. It achieves this as follows:

- Rigorous version control exists at the Component level, and classes are used at the Model level to define specific instances. For example, **class=1** of *Turbine* might have included *Stator;1, Rotor;1, Fuel_injector;1, ...* where the notation *";k"* refers to version *k* of a component; **class=7** of *Turbine* includes *Stator;2, Rotor;7, Fuel_injector;3, ...* (in other words, Stator, Rotor and Fuel_injector have been modified once, six times, and twice, respectively). Any class of *Turbine* that has not been purged can be fetched from the data base and used; any results generated with a given class will be stored with other results obtained with the same class, so there is never any question about how a result was created.
- Traceability between derivative models (e.g., linearizations and reduced-order linear models) and their parents (the original nonlinear or high-order model) is maintained. For example, *Lin_Turbine* is a linearization of *Turbine class=7* at the operating point *Power = 10 000 HP;* this information is stored in the data base as a *Reference* and *Condition_Spec*.
- Single-point storage of components is provided for those that may be used in building any number of models. For example, *Turbine* is the "home" of the component *Rotor;* model *TurbCtrl* uses this same component by *Linking* to the component stored in *Turbine*.
- An on-line Note Facility permits the user to store information/on-line documentation for any given project, model, component, or result in the data base. Headers are automatically generated to uniquely identify the element to which a note refers, and time-stamps are included whenever a note is added or modified.

Elements of the user's MEAD data base are accessed by a Browsing Facility that allows the user to display, annotate, purge and delete them via a point-and-click "selection form", as portrayed in Fig. 6.

The main deficiencies of the present version of this DBM are that it is somewhat limited in terms of data manipulation, it is not easy to search for data elements, and it is restricted to access by a single user. In addition, there are some aspects of support that are only partially addressed. These shortcomings can be alleviated by:

- making the DBM more open and flexible - e.g., allow the user to rename, move, compare, and search for specific data elements, and permit the interactive display of notes;
- complementing the point-and-click interface by adding other access modes;
- extending the Notes Facility so the on-line documentation of the user's design activity can be better supported, including automatic document generation; and
- adding functionality to permit *safe* and *flexible* multi-user access.

3.1. Flexible Data Element Manipulation

Renaming and moving data elements are elementary functionalities that are easy to implement. (This is necessitated by the fact that users typically become dissatisfied with the original name they gave an element or with where it was placed, and are then very frustrated if such a change cannot be done.) Any element can be renamed (as long as name conflicts are avoided), and moves can be permitted with the following limits: Results cannot be moved from one Model to another, Components cannot be moved to a Model where they are not used (in other words, a Component could only be moved from its "home" to another model that uses it via the Link mechanism mentioned above). Models (and associated components and results) can be moved arbitrarily among the user's Projects.

Comparing data elements can be done in several senses. At the simplest level, one would like to compare various time-histories obtained with a model or several models by cross-plotting the results; this is trivial. For higher-level comparisons, it would be helpful to have an object-oriented system, so each element has a *method* associated with the operation of *comparison;* for example:

- Component *Compt1* can be compared with *Compt2* to see how they differ (these elements could be different versions of the same component or merely similar components); this could be done on the data level (A_{23} might have different values) or attribute level (e.g., by comparing their Bode plots),
- *Result1* can be compared with *Result2* to see how they differ (again, either at the data level by using an ASCII differs utility or at the attribute level by cross-plotting the results or determining mean square error), and
- *Result1* can be compared with *Result2* to see how they differ in their *definition* (e.g., Result1 might differ from Result2 because Result1 was obtained with gain $K_{23} = 1.5$ and Result2 corresponds to gain $K_{23} = 2.33$).

3.2. *Improved Data Base Access*

Adding other modes of access to the point-and-click interface would do much to open up the MEAD user's data base. At present, the user has a limited "window" into the DB, e.g., a Browsing Facility Screen may display all the models in a given project (Fig. 6), or all the results for a given model class, and that is all. One way to facilitate finding data elements by name would be to incorporate a way to portray the entire user's data-base tree in a graphical form that conforms to Fig. 5. Such a display would allow one to determine which project contains *Turbine* much faster than by searching the project screens in the DB Browser one after another until it is located.

There are many cases where a command-mode interface would be still more effective. For example, a simple query language could be used to find all simulation result(s) for all classes of model *Turbine* with a step input of amplitude *WF = 2.33* much more expeditiously than browsing. Perhaps a limited subset of SQL (Standard Query Language) would be a good choice for this use.

3.3. *Improved On-Line Documentation*

The Note Facility can be made much more accessible if the notes can be displayed or modified *from the current screen* rather than from the Browser (Fig. 6). For example, if a 'Note' button is always available, then the user can:

- click 'Note' immediately after configuring a model to annotate it,
- click 'Note' immediately after saving a result to document it, and
- click 'Note' immediately after "modelizing" a result (installing a result as a model in the data base) to annotate the new model.

Further extensions could be made to create an auto-documenting environment. For example, MEAD presently does not prompt for notes as the user works and produces new data elements. In addition, the Notes Facility makes no attempt to relate individual note files to an overall document for a project or model. If an auto-documenting environment were implemented and AUTODOC were turned on, then a document framework could be created from templates and every user action that results in saving a data element could be recorded in that report and the user could be prompted for comments/text blocks to narrate the course of the effort. Organizations that require standard report formats and design approaches could thereby capture much of the required documentation material on-line.

3.4. *Multi-User Data Base Access*

Multi-user access to a single MEAD data base is the most important and substantial extension of the MEAD DBM. This would allow several engineers to work on the same project without the duplication of data (models etc.) and the corollary problems of maintenance and coordination. The main issues involved in developing multi-user data bases relate to safety: How can users share models and still be confident that they know precisely what they are using (version and class control provide some support here), and how can users update models safely (e.g., modify and create new classes without using stale versions of components); software engineering tools exist to solve this problem.

Preliminary thinking regarding opening the DBM to multi-user access was presented in Taylor, Nieh and Mroz (1988). The layer Sub-project was proposed in addition to those shown in Fig. 5, to accommodate a project leader (working at the project level) and other controls engineers working in individual workspaces corresponding to each sub-project. With this extended hierarchy, standard software development tools could be used to allow the leader to maintain the integrity of the overall data base and to control access to the various data elements. For example, DEC CMS (which the USAF MEAD DBM uses for model version control) supports the following (DEC, 1989):

> storing elements in a library, fetching elements for modification in the user's workspace, controlling concurrent changes to the same element, merging concurrent changes to an element, creating successive versions of an element, comparing two versions of a library element, relating library elements into groups, defining classes corresponding to versions of a set of elements, tracking which users are working on various elements from a library, and maintaining a historical record of element and library transactions.

Other higher-level functions can be performed on CACE data elements to support multi-user access rigorously. For example, the DEC Module Management System (MMS - DEC, 1990) automates and simplifies building software systems based on source code, object libraries, include files, compilers, and compilation and link options. This would further discipline and rigorize the building of complicated models. The above examples are based on VAX VMS tools; suitable support software is also available under UNIX™, e.g., the Configuration Management Facility (CMF™) provides similar functionality to CMS + MMS.

3.5. *DBM Conclusions*

In summary, much can be done to extend and improve the MEAD DBM. The features that might be considered by other CACE tool developers include flexible and convenient DBM access modes; full version and class control and tracking of results, references and links in the context of multi-user data bases; and provision for auto-documentation. These can be implemented using existing support software tools (e.g., CMS and MMS, as mentioned above) or by duplicating the capabilities of such tools, as required.

4. EXPERT SYSTEM IMPROVEMENTS AND EXTENSIONS

The coupling of an expert system to a CACE environment was proposed in the early 1980s (e.g., Taylor, MacFarlane and Frederick, 1983), and

---

™ UNIX is a trademark of AT&T Company; CMF is a trademark of EXPERTWARE, Inc.

The MEAD Supervisor was completely reimplemented between USAF MEAD and GE MEAD. Its design thus already reflects many of the lessons learned during the first 2 years of MEAD experience. For example, the USAF MEAD Supervisor contained explicit code for translating "MEAD commands" into package commands. Thus, knowledge of the syntax used by the different core packages was hard-wired and spread throughout the Supervisor. In GE MEAD, the data-driven Supervisor (Rimvall and Taylor, 1991) allows all MEAD commands to be defined in external definition files, which include all information needed to call the different core packages. Actions to be performed are input in the MEAD command language, which is then automatically translated into the correct syntax for the target package. These external definition files are in some sense similar to the ".m files" constituting a PRO-MATLAB toolbox, resulting in a similarly open and extendable architecture. The main difference between PRO-MATLAB and GE MEAD command definition formats is that the MEAD interpretive programming language is more powerful, to allow data-base and expert-system interactions as well as the mapping of MEAD commands into different core packages.

The open architecture of MEAD allows the system to be extended in several ways:

- allowing the creation of new commands (e.g., new aggregations of core package commands),
- accessing added functionality/commands in upgrades to already supported packages, and
- supporting new core packages.

5.1. Creation of New Commands

Individual users should be able to create new functionality by aggregating existing primitives into more powerful commands. In GE MEAD, this can be done by writing command-language macros. The Supervisor command language includes command flow statements such as conditional statements and loops, as well as some 75 control-theoretic and data-base related commands. These can be combined and structured freely to expedite tasks for the user. A detailed example of this is included in Rimvall and Taylor (1991).

In order to provide the individual user/user-group the ability to add functionality to MEAD through command-language macros without interfering with regular updates of the kernel MEAD system, macros may be declared on four different levels:

- system macros (common to all MEAD installations and maintained by the developers),
- site or implementation macros (to be distributed and maintained on a site-by-site basis),
- group macros (to be used by individual groups wishing to share added functionality), and
- user macros (to be utilized by one person).

This "layering" allows upgrades to be made to the overall system without interfering with any functionality introduced on another level.

5.2. Accessing New Core Package Functionality

As we have seen, one advantage of a general-purpose command language is that new commands can be added in the form of macros and procedures at any time. In MEAD, this capability may also be used to accommodate any upgrades (especially, new operations) made to the core packages.

5.3. Supporting New Core Packages

The GE MEAD Supervisor allows new commands to be added without coding. However, to add a *new package* one must:

- extend the existing MEAD macros to define how various MEAD commands are to be mapped into package commands (these macros are data-driven (no compilation necessary), and coded in the MEAD command language),
- create Ada-coded translation modules to accept MEAD-language commands and translate them into the new core package syntax, and
- create Ada-coded handshaking modules to interact with the package at the lowest level (recognizing prompts, error messages, etc).

The first step may be accomplished by editing non-compiled data-files; only the two last tasks presently require any programming in the traditional sense. It is possible to make the two latter modules data-driven as well, and thus allow the addition of new package drivers without further coding. This would have the advantages that access to the Supervisor source-code would not be needed, and users would be able to hook on new core packages without having any detailed knowledge about the Supervisor intrinsics.

To reach this state, it will be necessary to provide a means for formalizing the command language of the new package so that an automatic translation from the MEAD command language (or, more accurately, from its internal "threaded-code" representation) to the package command language can take place. This translation is in some sense the reverse of the "compiler-compiler" problem, and similar to the problem of generating a code-generating back end for a given compiler.

An even more challenging problem will be to formalize the hand-shaking mechanisms and error-detection/error-recovery schemes of a generic command-driven package so that the supervisor knows what kind of state the new package is left in. Pattern matching and dynamic state transition tools might provide the necessary machinery.

In summary, our experiences with the MEAD supervisor and with integrating kernel programs such as MATRIX$_X$ and PRO-MATLAB indicate that the openness of the architecture is the key factor in both ease-of-design and ease-of-use. This will become even more important in the future as different groups experience the need to connect or integrate different controls packages, or integrate controls packages with non-controls software.

6. OTHER EXTENSIONS

6.1. Automatic Controller Code Generation

Most CACE environments are limited to controller analysis and design, whereas controller

early work at GE and RPI (Taylor and Frederick, 1984; James, Frederick and Taylor, 1985/87), resulted in a specific plan for such a system and an implementation called CACE-III. Based on this experience, and activity in this area at other institutions, the question of selecting the best model or paradigm for expert-aided CACE was considered; a survey of AI applications and paradigms may be found in Taylor (1988) along with conclusions concerning their applicability and efficacy.

CACE-III was based on the idea that the expert system would serve as an interface and completely shield the user from the underlying CACE tools. The MEAD Expert System (ES), in contrast, has been founded on viewing the ES as a "control engineer's assistant". According to this paradigm (Taylor, 1988), the user invokes a rule base to carry out a task as follows: The button for an expert-aided function is clicked, the corresponding rule base is loaded into the ES, and it proceeds to elicit set-up information (e.g., specifications for a design process) and carry out the task (perhaps with intermediate interaction with the user). The ES then prepares a report, which the user can display and either initiate a new or modified task or move on. Since most controls engineers are quite computer-literate and aware of the use of conventional tools, this seems more appropriate than the approach used in CACE-III. Again, note that the ES is only operational in USAF MEAD (Taylor and McKeehen, 1989).

The present MEAD ES is very limited in scope and behavior. While it takes advantage of its capability to apply heuristic decision-making in the course of executing a clear-cut task, many other high-level benefits of this technology are neglected. The following extensions are applications of ES functionalities so far unused or underused:

- *smart helps* - asking 'why' and 'how' can elicit useful answers based on the underlying rule-base. Examples (based on the rule base for lead/lag compensator design in James, Frederick and Taylor (1985/87)):
  - — The ES asks for a closed-loop bandwidth (CLBW) specification to serve as the basis for lead/lag compensator synthesis; instead of supplying a number, the user clicks 'how'. The response could be to use the corresponding ES "reasonableness" rule to say: "You might try a value between 2* and 5* the CLBW you can achieve using a gain adjustment to achieve the same desired gain margin you specified in the previous step." A higher-level response could be given by having the ES carry out the gain adjustment and find the resulting CLBW.
  - — The user might click 'why' instead of 'how' in the above instance; in this case the ES could use its "ready_to_go" rule to say: "The problem is not completely defined until you provide specifications for gain margin, CLBW, and steady-state error."
- *tutoring* - the ES could continuously display logical and numerical steps to the user. Example (again, based on James, Frederick and Taylor (1985/87); this represents backward chaining to achieve a goal):
  - — The design task is done if the specifications have been met . . .
  - — The design specifications have been met if:
    steady-state error is within . . .
    and the CLBW is within . . .
    and the gain margin is within . . .
  - — The steady-state error is within . . . if DC_Gain is within . . .
  - — DC_Gain can be adjusted if initial DC_Gain is known
  - — If initial DC_Gain is unknown then call Find_DC_Gain
  - — The initial DC_Gain is 12.83 dB
  - — . . . and so on.
- *progress reports* - an intermediate level of user information could be provided every time the ES reached a milestone - e.g., after each iteration of lead compensation the lead parameters (gain, center frequency and pole/zero ratio), achieved performance (bandwidth etc.), and open-loop characteristics (e.g., Nichols plot) could be displayed; and
- *user influence* - at each "progress report" the experienced user could be allowed to modify the lead parameters if it is believed faster convergence will be achieved.

(Acknowledgement: the last two ideas have been implemented elsewhere (personal communication of D. K. Frederick).)

5. SUPERVISOR IMPROVEMENTS AND EXTENSIONS

All CACE packages, whether command-driven or graphics-oriented, will have some central controlling kernel, or supervisor, governing the execution of commands, storage and retrieval of data, error handling, external file handling, and so on. In command-driven packages this kernel primarily consists of a command-language parser accepting and decoding user commands, a command interpreter mapping these commands onto action routines, and a data handler for storing and retrieving the results of the commands. More modern, graphics-oriented packages such as MEAD, EAGLES or Xmath relieve the user from having to enter detailed and often ideosyncratic commands, yet these packages also have a similar kernel performing the above operations (where the command parser may be replaced with a module interpreting the user's interactive menu or form operations).

The MEAD Supervisor plays such a central role in the MEAD architecture (Fig. 1). In this position, it serves as coordinator and package integrator. The various numerical core packages run as separate processes under the direct control of the Supervisor, which is responsible for combining and controlling these packages as well as reformatting or converting data, when necessary, to ensure compatibility. The Supervisor is also responsible for facilitating communication among the UI/user, the DBM, and the expert system.

implementation must presently be done outside the CADCS environment (this is now true for MEAD). This requires a manual transformation of the controller as represented in the CACE environment (for example as a state-space matrix representation or a controller block diagram) into computer code. Such a manual translation is not only very tedious and time-consuming, it is also quite error-prone and, depending on the available experimental setup, hard to verify. Various attempts to automate this process have been made. The SystemBuild/ MATRIX$_X$ companion package Autocode will translate block diagram representations of linear or nonlinear controllers into FORTRAN, Ada or C real-time code. Autocode actually uses the same block-diagram editor as SystemBuild, making it possible to perform simulations or linearizations and produce real-time code from the same diagrams. ALS CASE, a block-diagram to code translator developed by Draper Lab under NASA sponsorship (CSDL, 1989), is another recent code-generation package that produces C or Ada.

These first-generation code generators are not without their problems. They often operate on a block-by-block basis, producing unoptimized and ill-structured code. In conjunction with todays immature Ada compilers, this results in unacceptably slow code. Moreover, their ability to generate code customized to a particular environment or real-time operating system is often limited.

Despite these shortcomings, they constitute a first step in the right direction - an automatic code generator to translate block diagrams into real-time code is essential for gaining the engineering productivity sought today. In the case of MEAD, this code generator should preferably tie in directly with the data base so that the different components of a model can be automatically translated into a consistent real-time code unit.

6.2. *Disciplinary Extensions*

Many extensions can be made to support specific disciplines more completely. Nonlinear system modeling is often time-consuming and prohibitively expensive, so replacing or augmenting general-purpose simulators (see section 1) with domain-specific modeling environments would result in substantial improvements in CACE efficacy. For example, one can add a NASTRAN interface for structural modeling, or a special-purpose electromechanical simulator for analog device simulation, e.g., Saber™, to broaden the scope of the environment. A variety of extensions of this sort can be made to realize a broadly multidisciplinary CACE system - however, the UI and Supervisor extensions in previous discussions are essential for providing the infrastructure to make this feasible in a fully integrated fashion. The problems and factors involved in the cross-integration of such packages with the generic CACE functionality remains a topic for further research - it is not sufficient merely to "hang" a new core package below a CACE Supervisor, as the preceeding discussions make clear.

™ Saber is a trademark of Analogy, Inc.

7. SUMMARY AND CONCLUSIONS

We hope that the lessons learned in developing MEAD will be helpful in charting new courses in CADCS, and that the ideas and extensions described in this paper will be of broad applicability and benefit to many package developers. In addition, we trust that the overview of packages and functionality will be useful for controls practitioners in evaluating and selecting CACE software.

REFERENCES

CSDL (Charles Stark Draper Lab) (1989). Using the automated programming subsystem of ALS CASE. Cambridge, MA USA.

DEC (Digital Equipment Corp.) (1989). *Guide to VAX DEC/ Code Management System*. Maynard, MA.

Delebeque, F. and Steer, S. (1986). Some remarks about the design of an interactive CACSD package: the Blaise experence. *Proc. IEEE Third Symp. on CACSD*, Arlington, VA USA.

EAGLES/Controls Documentation. (1986). Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550 USA.

Edmunds, J. M. (1979). Cambridge linear analysis and design program. *IFAC Symp. on CADCS*, Zurich, Switzerland.

Floyd, M. A., Dawes, P. J. and Milletti, U. (1991). Xmath: a new generation of object-oriented CACSD tools. *Proc. European Controls Conf.*, Grenoble, France.

Hummel, T. C. and Taylor, J. H. (1989). Multi-disciplinary expert-aided analysis and design (MEAD). *Proc. Third Annual Conf. on Aerospace Computational Control*, Oxnard, CA USA.

ISI (Integrated Systems, Inc.) (1990). *MATRIX$_X$/ SystemBuild Tutorial*. 2500 Mission College Blvd., Santa Clara, CA 95054 USA.

James, J. R. (1988). Expert system shells for combining symbolic and numeric processing in CADCS. *Proc. 4th IFAC Symp. on CADCS '88*, Beijing, PR China.

James, J. R., D. K. Frederick, and J. H. Taylor. (1985/87). On the application of expert systems programming techniques to the design of lead/lag precompensators. *Proc. Control 85*, Cambridge, UK; also in *IEE Proceedings D: Ctrl Theory and Applications, 134*, 137-144.

Lohr, P. J. (1989). CHIDE: a usable UIMS for the engineering environment. Tech. Report, GE Corp. R & D, Schenectady, NY 12301 USA.

MathWorks, Inc. (1989). *MATLAB User's Guide*. 21 Eliot St., South Natick, MA 01760 USA.

MathWorks, Inc. (1991). *SIMULAB User's Guide*. 21 Eliot St., South Natick, MA 01760 USA.

Minto, K. D., Chow, J. H., and Beseler, J. (1989). Lateral axis autopilot design for large transport aircraft: an explicit model matching approach. *Proc. American Control Conf.*, Pittsburgh, PA USA.

Moler, C. (1980). *MATLAB User's Guide*. Dept. of Computer Science, University of New Mexico, Albuquerque, NM 87131 USA.

Munro, N. (1990). ECSTASY - A control system CAD environment. *Proc. 11th IFAC World Congress*, Tallinn, USSR.

Ravn, O. (1989). On user-friendly interface construction for CACSD packages. *Proc. IEEE Workshop on CACSD*, Tampa FL USA.

Rimvall, M. and Cellier, F. E. (1985). A Structured Approach to CACSD. In M. Jamshidi and C.J. Herget (eds) *Advances in Computer-Aided Control Systems Engineering*. pp. 149-158, North Holland, Elsevier Science Publisher, Amsterdam.

Rimvall, C. M., H. A. Sutherland, J. H. Taylor, and P. J. Lohr. (1989). GE's MEAD user interface - a flexible menu- and forms-driven interface for engineering applications. *Proc. IEEE Workshop on CACSD*, Tampa FL USA.

Rimvall, M., and Taylor, J. H. (1991). Data-driven supervisor design for CACE package integration. *Proc. 5th IFAC Symp. on CADCS*, Swansea, UK.

SCT (Systems Control Technology, Inc.) (1987). *Model-C User's Guide*. Palo Alto, CA 94304 USA.

SCT (Systems Control Technology, Inc.) (1988). *Ctrl-C User's Guide*. Palo Alto, CA 94304 USA.

Spang, H. A. III, Rimvall, C. M., Sutherland, H. A., and Dixon, W. (1990). An evaluation of block diagram CAE tools. *Proc. 11th IFAC World Congress*, Tallinn, USSR.

Taylor, J. H. (1988). Expert-aided environments for CAE of control systems. (plenary lecture). *Proc. 4th IFAC Symp. on CADCS '88*, Beijing, PR China.

Taylor, J. H., MacFarlane, A. G. J., and Frederick, D. K. (1983). Second-generation software plan for CACSD. IEEE Symp. on CACSD, MIT, Cambridge MA USA.

Taylor, J. H. and D. K. Frederick. (1984). An expert system architecture for CACE. *IEEE Proceedings, 72*, 1795-1805.

Taylor, J. H. and P. D. McKeehen. (1989). A CACE environment for multi-disciplinary expert-aided analysis and design (MEAD). *Proc. National Aerospace and Electronics Conf. (NAECON)*, Dayton, OH USA.

Taylor, J. H., K-H Nieh, and P. A. Mroz. (1988). A data-base management scheme for CACE. *Proc. American Control Conf.*, Atlanta, GA USA.

Taylor, J. H., Frederick, D. K., Rimvall C. M., and Sutherland, H. A. (1989). The GE MEAD CACE environment. *Proc. IEEE Workshop on CACSD*, Tampa FL USA.

Taylor, J. H., Frederick, D. K., Rimvall C. M., and Sutherland, H. A. (1990). CACE environments: architecture, user interface, data-base management, and expert aiding. (invited tutorial) *Proc. 11th IFAC World Congress*, Tallinn, USSR.

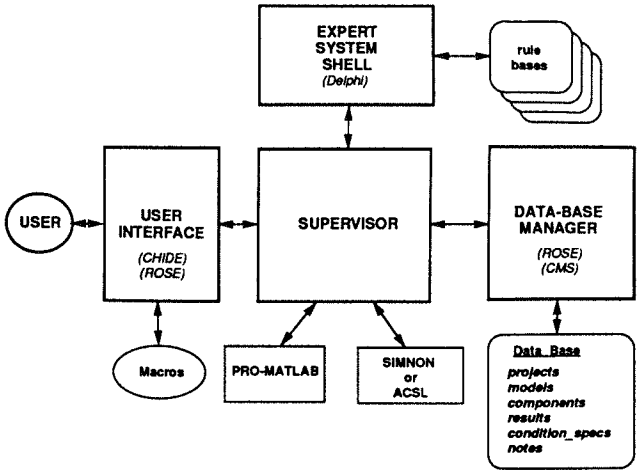

Figure 1. GE MEAD Architecture

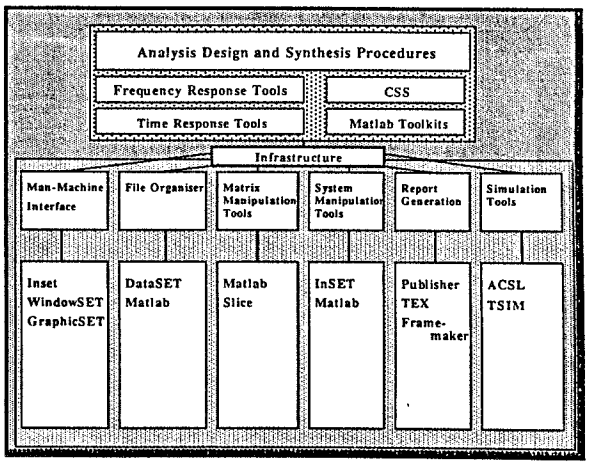

Figure 2. ECSTASY Architecture

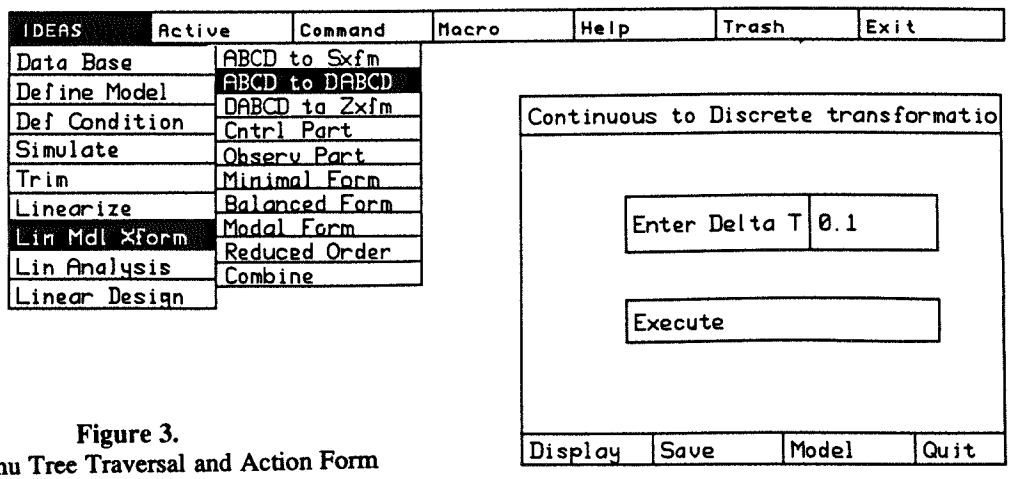

Figure 3.
MEAD Menu Tree Traversal and Action Form

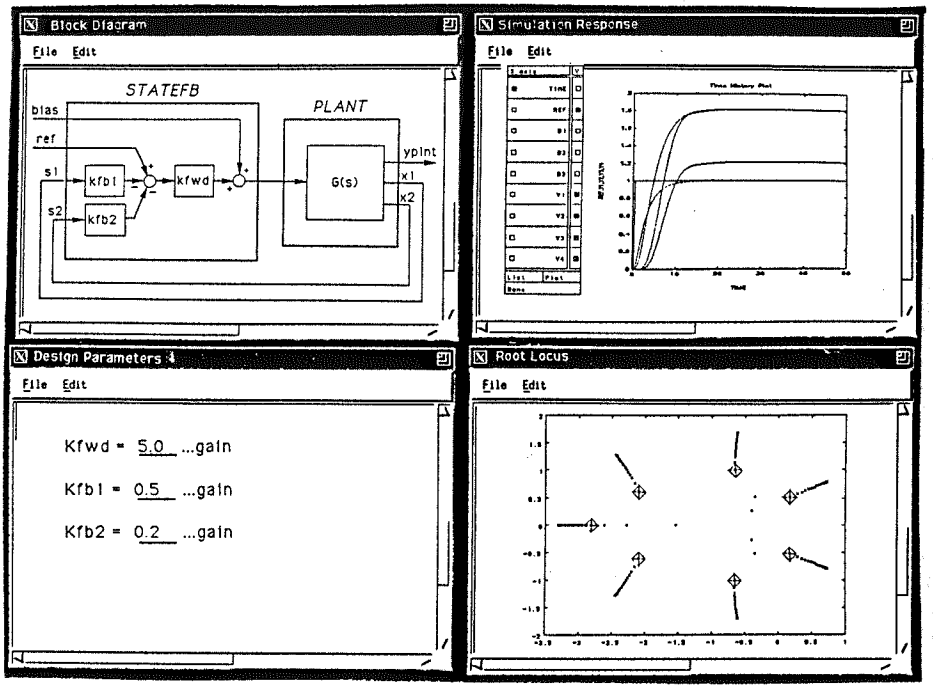

Figure 4.
Dynamic Data-Base Viewing

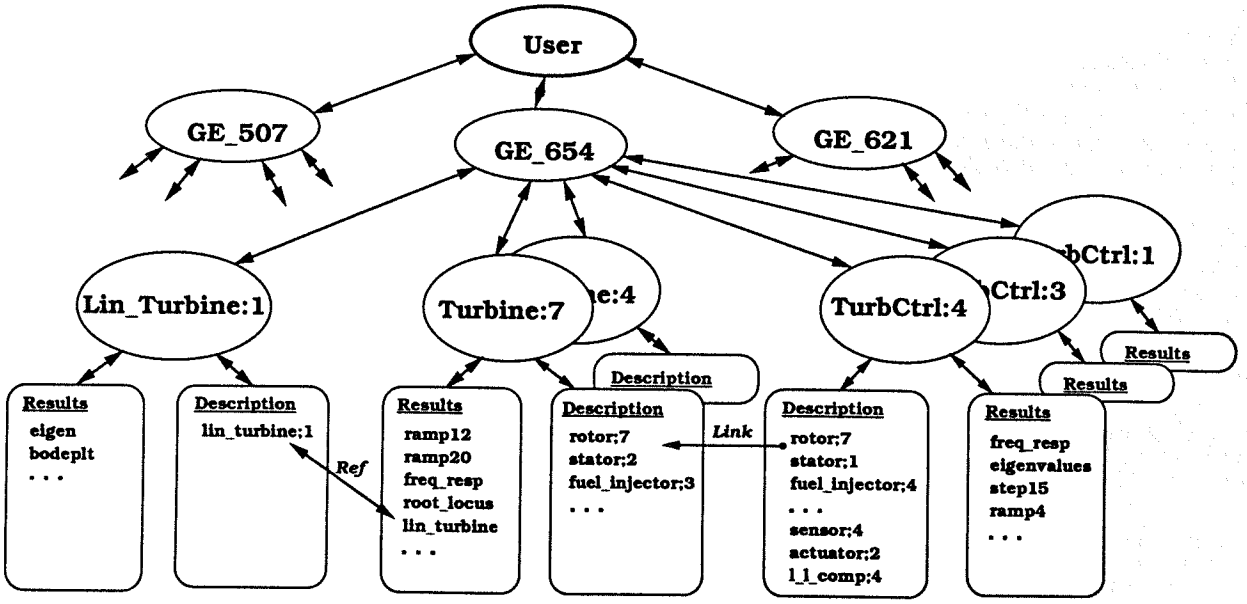

Figure 5.
MEAD Data-Base Hierarchy

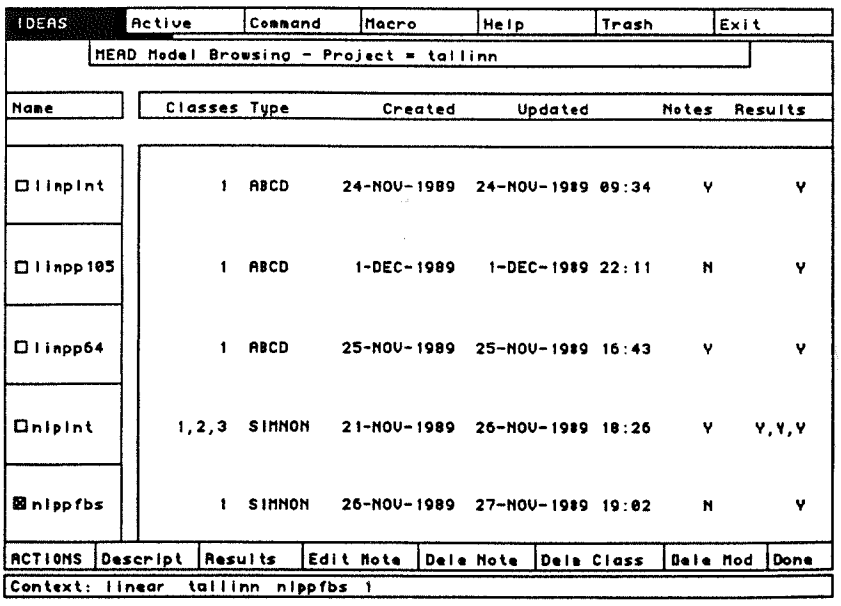

Figure 6.
MEAD Model Browsing Screen