SIMULATION AND COMPUTER AIDED CONTROL SYSTEM DESIGN IN ENGINEERING EDUCATION

A. Fischlin*, M. Mansour, M. Rimvall and W. Schaufelberger

Department of Automatic Control & Industrial Electronics, Swiss Federal Institute of Technology, ETH-Zürich, ETH-Zentrum, CH-8092 Zürich, Switzerland
**also: Department of Phytomedicine, Swiss Federal Institute of Technology*

Abstract. A wide range of possibilities exists for the use of computers in the teaching of control system design and simulation. Several approaches such as the use of program packages, single purpose programs or complete environments are described in the paper. Experience gained from teaching with these methods is summarized, and recommendations for forther improvements are given.

Keywords. Computer Aided Control System Design. Education. MATLAB environment. Simulation. Teachware for control education.

INTRODUCTION

Non scholae, sed vitae discimus.
[Not for the school, for life do we learn.]
Epistolae morales

In the last few years, microcomputers have become economically so attractive that many educational institutions started to invest into their employment. This is especially true for simulation which has a long tradition in using computers.

There exist many approaches. They range from the programmed learning, dating back more than one decade before the era of personal computing, to courseware supported learning by completely free exploration of unknown structures. Although, thanks to personal computing, programmed learning has gone through some sort of a reincarnation recently, we do not believe that this approach is fruitful in the field of simulation. On the contrary! Simulation offers almost for any discipline the possibility to replace real-world, complex, experimental set-ups by simple and cheap models running on a microcomputer. Students may explore these models in place of the real systems in any way they wish; no risks are associated with such experiments, even if the beginning student should know almost nothing about the real system. Hence, not only is simulation a learning subject, but it is also the means to learn a subject. Because of its similarity to a real-life situation, in contrast to programmed learning, the student profits from opportunities to acquire skills useful even beyond just the learning situation. Moreover, the motivation of students may be increased by demanding less of cognitive in place of intuitively appealing, unforgettable personal experience.

In Computer-Aided Control Systems Design (CACSD), the use of CACSD-packages as well as simulation tools let us develop and test a new design for consistency up to any level of detail before having to implement it within the costly, real-life situation. However, two learning goals have to be clearly distinguished: The design of control systems for which appropriate software packages are just some means which may be employed without a detailed understanding of the underlying techniques and the detailed study of the software tools themselves and their employment from the perspective of the design of control systems. Both goals deserve appropriate attention and the development of specific software in order to support them optimally. In the first situation, the utilized techniques are typically hidden from the student so that he or she can concentrate on the main subject, the design of a control system. In the second learning situation, the techniques and algorithms must not be hidden, on the contrary, the courseware would become meaningless without their emphasis. The only purpose the simulated system serves is to provide a model, which is appealing to the student.

The distinction between the two learning goals described above is much more fundamental than it might appear at the first glance. Students of electrical engineering remain at our university (Swiss Federal Institute of Technology Zurich, ETHZ) for a minimum of 4 years. Spending much time and effort on training particular techniques, rather than basic principles, offers students the advantage of being able to start working in a professional position right after having finished their studies. On the other hand, it has the danger that the acquired knowledge quickly becomes obsolete. Especially when considering the current speed of technological development in the era of personal computing, these two goals become Skylla and Charybdis.

The students have to know the widely used software packages commercially available. On the other hand, they also have to gain a thorough understanding of the basic principles in order to be prepared when the next software generation becomes available on the market.

Emphasis should be set on the few well-established classics, such as MATLAB. They are worth learning the idiosyncratic peculiarities.

Another criteria to select a software package is its power to represent a whole class of tools. Given that the functions of such a tool serve a long-lasting purpose, even if the specific product does not survive, at least the corresponding family of similar

programs might. Having once mastered the principles, learning to use a new product of the same program family will be a minor task. Consequently, particular software used in education may be favoured for its didactic merits, demonstrating much clearer what its underlying principles are, over other pieces of software despite their current popularity among professional practitioners.

Attempts to minimize efforts to learn software idiosyncraciecs lead to design problems of user interfaces: Syntax, usage, and programming of operating systems; command-driven versus menu-driven software; graphics, i.e. graphical output and graphical input via a pointing device, e.g. a mouse; and the overall behavior of the application software, the so-called user model.

Most conventional operating systems have first to be learned before any computations can be performed. Typically, they are full of illogical idiosyncracies, forcing the user to devote intolerable large portions of his intellectual capacity to the debugging of meaningless, syntactical errors.

Command-driven software is much more difficult to master than menu-driven software featuring pop-up or pull-down menus. Although command-driven software can be used more efficiently if the user is proficient, menu-driven software can even be of much value to the versatile user when he is accessing many different computer systems.

Any program adopts a particular "user model". It determines the general behavior of the program, not dealing with aspects determined by the application-specific algorithms. Using more than one computer system and more than one program, the case with most of our students, means that the user model also changes. Again this represents an unnecessary and time-consuming burden for the student. Hence, many computer-based control system design techniques and tools can not be taught to the students, simply because the time necessary to learn the overhead cannot be afforded. This holds in particular for students of system theory for whom the emphasis lies not on computer techniques but on mathematical solutions to control system design problems.

Finally, graphical output should be featured whenever and wherever possible, since what holds in general is true even more so for computers: one graph can replace a thousand words. In learning, this is particularly crucial.

In our paper, we present some of the approaches with which we have experimented. Some of these allowed us to gain experience over a period of several years. They address many of the problems mentioned: the user shell MIDGET, which runs on top of the VAX/VMS operating system, addresses the operating system problem. The "Dialog Machine" is an approach to offer a general user model; some courseware programs featuring simulation just as a vehicle to purport an independent message, such as a method to design a controller, address the issue of software tailored to didactical needs. Finally we discuss our experience in the use of command-driven, general purpose research tools, CTRL-C and MATLAB, for educational purposes, addressing the issue of difficult to learn, but real-life software. Advantages and disadvantages of the various approaches are discussed, experience presented, and an outlook for future developments is offered.

OVERVIEW OF APPROACHES AND SOLUTIONS

As no universal simulation/CACSD software solving all simulation problems exists (and probably never will exist), a considerable number of different software packages for simulation and controlsystem design are available at the ETH. Table 1 lists the most commonly used packages at the Dept. of Automatic Control (it is an updated version of a list from Mansour and co-workers, 1984). Note that a distinction is made between packages used as tools only, and packages developed and/or complemented at the ETH.

1) ACSL (Advanced Continuous Simulation Language). An easy-to-learn, well-structured interactive simulation language for continuous and sampled data systems. ACSL includes features like macros, procedurals, and non-linear functions. In the interactive mode, the user can change model parameters, perform simulations using terminal color graphics, as well as analyze a simulated system through its Jacobian matrix and eigenvalues.
Available from Mitchell & Gauthier Associates, Inc., ACSL is presently the most used simulation language at the department.

2) COSY/SYSMOD (COmbined SYstems, SYStem MODelling). Simulation language for combined continuous and discrete systems, offers a high-level input language.
The syntactical language definition of COSY has been defined using a general-purpose parser. At present, the COSY definition describes a state-of-the-art simulation language with a structurability and versatility not available in any other languages. Preprocessor and runtime system are presently not available.
Developed at our institute; a subset of COSY (SYSMOD) has been implemented by Systems Designer Ltd. on command by the British Ministry of Defence (Baker and Smart, 1983).

3) DARE-INTERACTIVE (Differential Analyzer REplacement). Extension to DARE-P, makes the language interactive in a fashion similar to that of DARE-ELEVEN. New features include color graphics, a run-time display, split-screen graphics, modules for sensitivity analysis, replication and batch, and trend analysis. DARE-INTERACTIVE will in the future also provide access to a general data-base.
Developed at the institute from DARE-P; runs under VAX/VMS.

4) GASP-V/INTERACTIVE. Interactive version of the GASP-V package. Through the use of MIDGET and an interactive postprocessor from the DARE family, the user can enjoy a very flexible version of the powerful GASP-V package.
GASP-V/I was developed at our institute; it runs under VAX/VMS.

5) SDL (Simulation Data Language). Portable relational data base especially adapted to storage and retrieval of data from simulations. SDL can be easily connected to any simulation program providing for a FORTRAN interface.
Developed by Pritsker and Associates, Lafayette, IN., SDL is used in the second part of the simulation lecture (discrete simulation) in connection with SLAM-II.

6) SIMNON. Interactive simulation program for continuous and sampled data systems. Features very natural model descriptions as well as an easy-to-use dialog form. One of the first packages to include terminal graphics to support interactive analysis/synthesis/optimization.
Developed at Lund Institute of Technology (LTH), Lund, Sweden. Runs under VAX/VMS.

7) SLAM-II. Batch-oriented simulation language using a PERT network description of discrete models to be simulated. As SLAM-II is a superset of GASP-IV, it can also be used for combined con-

tinuous/discrete simulation with only part of the system described by a network.
SLAM-II was developed by Pritsker and Associates. It is presently used in the second part of the simulation lecture (discrete simulation).

8) STELLA. Ikon-oriented simulation package incorporated into the Macintosh environment. Although the present version of STELLA is much too rudimentary to be used for large simulations, its extreme simplicity makes it interesting for small student exercises.

TABLE 1: CACSD and simulation tools available at the ETH Zürich

	Use at different levels					
SIMULATION	T 5+6	T 7+8	Lab.	Stud. Proj.	Grad. Prog.	Use in Res.
ACSL	0	2	0	2	1	2
COSY/ *						
SYSMOD	-	-	-	2	0	2
DARE-INT*	0	0	2	0	0	1
DSL/VS	0	1	0	0	0	1
FORSIM	0	0	0	1	0	1
GASP-V/I*	0	0	0	0	0	1
SDL	0	1	0	1	0	1
SIMNON	1	1	0	0	1	0
SLAM-II	0	2	0	1	0	1
STELLA	1	1	0	0	0	1
CACSD						
CTRL-C	2	2	0	2	2	2
IMPACT*	-	-	-	2	0	2
MATLAB	0	0	1	0	1	2
PC-MATLAB	0	0	1	0	1	1
TOOLS						
DIALOG-MACHINE*	2	0	0	1	0	2
MIDGET*	2	2	2	2	2	2
SPECIAL-PURPOSE*	2	1	1	1	0	0

2: much used; 1: used; 0: hardly ever used; -: not applicable/not available; *: developed at ETH; T: term.

Although there is a large common area between the fields of simulation software and CACSD software, the two are treated separately in this paper. Some of the following CACSD systems also contain simulation features.

1) CTRL-C. Commercial package extending the original MATLAB program to a full control-environment. Flexible graphics, control algorithms and function/procedure capabilities have been added. Available from Systems Control Technology, Inc., California.

2) IMPACT. The newest CACSD project at our institute. IMPACT is conceptually based upon MATLAB, but implemented in Ada®. IMPACT supports a multitude of data-structures needed in control theory, (e.g. polynomial structures and system descriptions), and provides a dual-mode command-driven/question-and-answer-driven interactive interface for easy use.

3) MATLAB. Developed at the Univ. of New Mexico, MATLAB provides an easy-to-use, interactive interface to matrix manipulation algorithms of LINPACK and EISPACK.

4) PC-MATLAB. Commercial PC-version of MATLAB implemented in C. It extends MATLAB in the same areas as CTRL-C, and fully utilizes the 8087 co-processor to make PC-MATLAB almost as fast as the original MATLAB on e.g. a VAX-750. Available from The Mathworks, Inc., MA.

Some universal algorithmic/operational software is used on many different simulation/CACSD packages. All the here listed tools have been developed at the institute, and are treated in detail in other parts of this paper.

1) DIALOG MACHINE. General Purpose graphical and dialog software to be used during the implementation of interactive software on a modern, single-user work station.

2) MIDGET. General purpose environment facilitating the use of larger (simulation) packages.

3) SPECIAL PURPOSE PROGRAMS. Stand-alone programs developed to be used in laboratory or classroom exercises to solve particular problems.

STUDENT ENVIRONMENTS UNDER MIDGET

As we have seen in the first part of this paper, a multitude of software packages are available for simulation and CACSD during exercises and/or laboratory work. However, their employment is unfortunately not as simple as "go to computer A, start package B, and solve problem C". Although the average student hopefully knows from the lecture how to solve problem C, he probably knows neither

- how to use computer A, nor
- how to start and work with package B.

Consequently, before any exercise using a (for the student) new computer and/or software package, much time must be spent on computer introductions. Thereby, the introduction to the used package itself may be well employed when it also gives the student insight into the techniques utilized by the package. Differently, the introduction to the operating environment of the package is often totally irrelevant to the exercise and subject matter. This can lead to an extensive loss of time. Generally speaking, there are three approaches to this problem:

- Introduce the principles and basic commands of the available operating system to the students, so that they can work independently on that particular computer.

- Provide the students with a step-by-step list of operating system commands to be entered with no or few explanatory comments.

- Incorporate the needed operating system commands into a simple-to-use user environment ("user-shell") to hide the operating system.

The first, conventional approach is the most inefficient one, yet probably the most wide-spread one. As every consistent subset of a conventional operating-system is likely to be quite substantial, much time would have to be spent before the student can work independently in a fairly efficient manner.

In some cases, it is better to let the students blindly copy a prescribed, but unexplained, list of commands. However, this method is only employable where a simple application package is to be used and few actions involving the operating system must be taken. Otherwise, the slightest error by the students will leave him totally confused.

In our view, the best approach is to provide the student with an adapted environment, and thereby to shield him from the intricacies of the operating system (except, of course, in lectures on system software, etc.). Such an environment must on one hand be simple enough to learn and use to warrant its implementation time, on the other hand, it must let the student perform all necessary actions (or we are back at square one again). These two requirements are at a first glance contradictory, yet the existence of a general environment or a tool for creating new environments (an "environment environment") would put us on the right track.

At the ETH, we have experienced with all three ways of presenting "conventional" computers to the students (typically VAX/VMS, CDC or IBM-mainframe environments). We employ the first two methods primarily during larger projects where there is ample time, or when the utilization of the employed package requires few operating system commands to be entered (e.g. invoking "matrix-environments"). For the access to more complex packages during shorter exercises, we try to take the third approach.

In a first generation of simulation-package environments implemented at the institute some four years ago, the student still worked with the normal operating system. To assist him, we supplied a few powerful commands hiding most of the actual system actions taken. However, this approach was unsatisfactory, as the student still "could take too many wrong paths" without realizing it.

In a second generation of packages, the student never got in contact with the underlying operating system. He only had access to a few commands presented to him over a menu, limiting the number of mistakes he could make and streamlining the access to of different software packages. Although this second generation worked very satisfactorily for the students, the implementation time of these environments became tremendous. Many times the students asked a relevant "why doesn't the environment allow me to do this?", which lead to extensions to each of the packages. Moreover, our students became "spoiled" and asked us to supply these nice environments for more packages. It soon became clear that what we needed was an environment for developing student environments. Thus the idea of MIDGET was born.

MIDGET (Menu-driven Interative Development-system for Generic Engineering Tasks) is a set of Pascal and operating system programs (VAX/VMS) for the development and employment of software environments. MIDGET standardizes the user interface of the different software environments without restricting the kind of actions supported by these environments. Hence, environments have been developed for very diverse software packages (from simulation languages to word-processing systems), yet the implementation time for individual environments is short (over 80% of the code are common to all environments and another 10-15% are reusable between similar kinds of environments). Normally, a "MIDGET-manager" familiar with the development system for environment as well as the software package to be given an environment, needs 2-3 working days to construct a new environment. This as contrast to 2-3 weeks for our initial, second-generation environments.

MIDGET is simple enough to be used after only 15-20 minutes of introduction. To prove this, let us spend the next 15-20 lines of text explaining the principles of a typical MIDGET environment.

In Fig. 1 you see the main menu presented to the user by the ACSL environment (ACSL is a powerful simulation language for continuous models). In the top third of the screen, the state of the environment is given. In particular, the condition of "flags" (take the value ON or OFF) are displayed and the name(s) of the problem presently treated is given. Each of these fields can be changed by entering corresponding commands - the most important of these commands are displayed in the middle of the screen. At the bottom of the screen, names of further sub-menus are displayed. These sub-menus generally contain commands not needed during the first days a student works under the environment; when needed, they get displayed simply by giving their name. Some of the commands (like the SELECT command choosing the name of the model to be treated) takes parameter(s). If the user does not supply these parameters with the command (in the form SELECT MYPROBLEM), he will be asked for the parameter value(s). Whenever the user is at loss, the hierarchical HELP-command gives further assistance. Contrastingly, when the user gets familiar with the environment, he can switch the menu OFF for faster operation.

Not only does the "reusability" of the MIDGET environments decrease their implementation time, this also assists the user switching from one environment to another (for example, the user needing ACSL for continuous simulations and SLAM for discrete-event simulations). Even between environments of very diverse software packages (such as ACSL and SLAM), a large section of the commands remain the same. For example, almost all simulation environments support the commands SELECT, PROGRAM, EXECUTE and OUTPUT; yet the action taken as the user enters the commands is quite different (sometimes an EXECUTE compiles a model, another time an interpreter is called, and a third time the action depends on the value of a "flag"). Typically, the introduction time for new environments is around 5-10 minutes.

```
FLAGS   : MENU - ON ACSL listing - ON
FLAGS   : FORTRAN listing - OFF Loader map - OFF
FLAGS   : Own main program - OFF
FLAGS   : Analysis feature - OFF
DEFAULTS: All defaults       - [CACSD.RIM.SIM1]
SYSPICS : Selected problem   - MYPROBLEM

COMMANDS:
   DATA      ⇐ Edit the ACSL data file (run time
               commands)
   DIRECTORY ⇐ List all defined ACSL problems
   EXECUTE   ⇐ Compile and run ACSL problems
   HELP      ⇐ Extensive help information
   OUTPUT    ⇐ Edit ACSL output file
   PROGRAM   ⇐ Edit the ACSL program (system
               description)
   SELECT    ⇐ Select problem to be treated

SUB-MENUS containing further commands:
   ADVANCED  ⇐ Advanced commands to run ACSL
   DEFAULTS  ⇐ Commands to change default values
   DEVICES   ⇐ Commands for external devices
   EDIT      ⇐ Additional edit commands
   FILES     ⇐ File manipulation commands
   SYSTEM    ⇐ Commands leave current development
               system

SELECT COMMAND:
```

Fig. 1. Main menu of the MIDGET environment to ACSL

Somewhat surprisingly, the use of MIDGET has not limited itself to students. Many of our colleagues use MIDGET whenever they work with any simulation language (and are then at a total loss when they have to work on a "strange" computer where they suddenly have to "know what to do"). Except for the cases where the user wants to modify the simulation package itself, the use of MIDGET increases productivity without operational limitations. Moreover, the installation of MIDGET at other academic institutions worldwide, and the development of over a dozen different environments indicate that our approach to student environments for larger software packages on machines with conventional operating systems is sound.

THE DIALOG MACHINE

The "Dialog Machine" is a software package which has been produced as part of an authoring system under development at the Swiss Federal Institute of Technology Zurich (ETHZ) by the project team CELTIA (Computer-aided Explorative Learning and Teaching with Interactive Animated Simulation). It has been implemented on the Apple® Macintosh computer (512K RAM or more) and in its current version it consists of eleven modules supporting pull-down menus, windows, window-related input and output, modal dialogs, modeless dialogs, alerts, and files. The "Dialog Machine" is an abstract machine filtering so- called user events, reating to them whenever possible automatically and passing them whenever necessary to handlers provided by the application program.

The "Dialog Machine", once started, attempts to keep control over all run-time activities of an application program. It intercepts all events due to a user action, the so-called user events, such as pressing the mouse button, choosing a menu item, activating a window, clicking an object, or dragging an object on the screen, and reacts to them in a predefined, standard way. Only events, which cannot be treated automatically by the "Dialog Machine" are channeled through the system to the application-specific program sections. All user events are rigorously defined and the programmer can interface his application-specific code to the user events in a structured way. For instance, the clicking in the front window, or the closing of a window, which does not correspond to the choosing of a menu item, are such events. Program control is only temporarily passed to application-specific program sections. Consequently, the application-specific software consists of a set of procedures which can be called in an arbitrary sequence, rather than of a conventional block of program statements (straightline code) to be executed one after the other (Fig. 2).

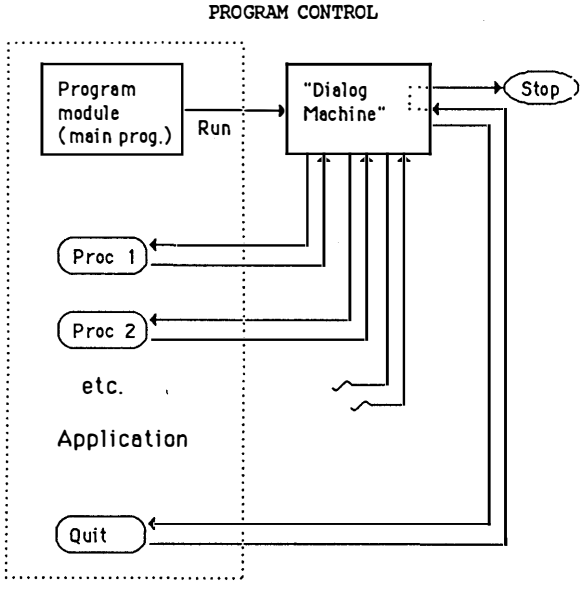

Fig. 2. "Dialog Machine" and Program Control.

The "Dialog Machine" separates those program parts, which are common to any interactive application program from those program sections, which are application-specific. It thus simplifies the programming task of a typical application program substantially. Furthermore, the "Dialog Machine" standardizes not only the appearance, but also the behavior of an application program as far as the user dialog is concerned, helping students to orient themselves within complex courseware and reducing the time needed for learning to use new software.

The "Dialog Machine" can be considered as a versatile software development environment featuring consistent usage and simplified programming of modern working stations (Fischlin, 1986).

The "Dialog Machine" consists of eleven Modula-2 library modules (DM stands for "Dialog Machine"): DMMaster, DMMenus, DMWindows, DMWindowIO, DMModalDlgs, DMModelessDlgs, DMAlerts, DMFiles, and two auxiliary modules DMLanguage, DMConversions, plus DMSystem (Fig. 3).

The modules DMMaster, DMMenus, DMWindows, DMWindowIO, and DMLanguage depend mutually on each other. They build the core of the "Dialog Machine" and must always be resident. The modules DMModalDlgs, DMModelessDlgs, DMConversions and DMFiles can be used in addition to the core modules. Two may even be used completely independently of the "Dialog Machine" (DMConversions, DMFiles). The system-specific characteristics, e.g. the screen dimensions are provided by the auxiliary module DMSystem.

The module DMMaster is the master module maintaining overall control of all actions, in particular user events. User events that can be handled automatically by the "Dialog Machine" are either passed to the other modules or module DMMaster responds to them directly. These mechanisms are transparent to the programmer or user. Does the user event require some particular, application-specific response, information on the event is found within the appropriate module. For example, the current mouse position is needed during dragging. Since the "Dialog Machine" supports only dragging within a window, this information is available from module DMWindowIO. However, pressing a key on the keyboard is a user event, which is not related to any particular user interface object, such as e.g. a window. Hence, information on user events of this type, e.g. which key has been pressed, have to be taken from module DMMaster.

The module DMMenus supports the installation and management of menus. Typically, application-specific procedures are installed within the "Dialog Machine" and provide the desired action when the corresponding menu item is chosen. All procedures for the activation or deactivation of menus, or changing menu texts, or checking a menu item, plus all other similar menu management tasks are supported.

The module DMWindows provides the window management. Several window types are supported: It is possible to create windows with or without scroll bars (scrolling and updating is performed automatically), with a fixed or adjustable size, with or without a close box, at a fixed screen location or movable, etc. Typically, a window is created by simply calling the procedure CreateWindow, and all other, subsequent window related tasks, such as resizing, activating, or closing of a window, are left to the "Dialog Machine" via mouse user events.

Module DMWindowIO provides graphical input via the pointing device and textual or graphical output based on a small set of simple coordinate systems.

Via module DMWindowIO it is possible to detect one type of user events, i.e. the clicking within the content of a window, and to relate them to graphical objects of a round or rectangular shape. Furthermore, routines to drag graphical objects once their sizing has been detected, are offered. Procedures to determine scrolling amounts and scroll slider positions.

The first output coordinate system of module DMWindowIO supports textual output by addressing character cells in rows and columns. A second, graphical coordinate system is based on a two-dimensional cartesian coordinate system in pixel units with its

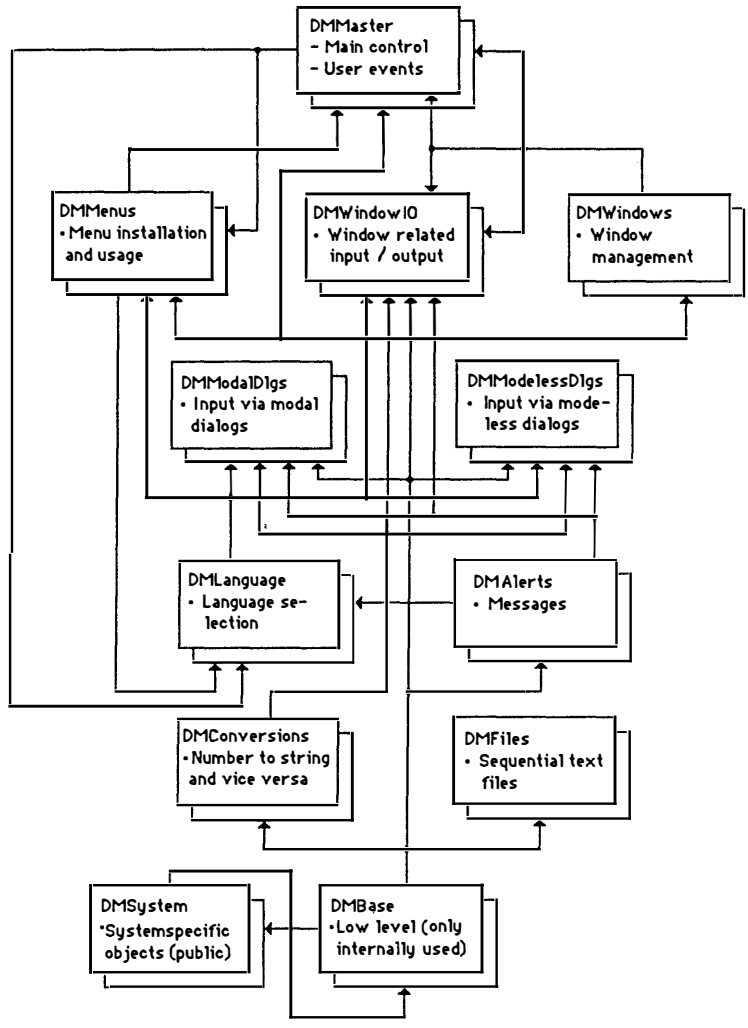

Fig. 3. The structure of the "Dialog Machine"

origin typically to the lower left corner of the output window. Thirdly, turtle graphic routines are offered. The fourth coordinate system is the so-called user coordinate system, which addresses points as pairs of real numbers. It maps any two-dimensional cartesian coordinate system defined in real numbers to a rectangular portion of the current output window. It is possible to freely switch from one coordinate system to the other. All output is actually done with one common pen only. Hence, the various coordinate systems are just convenient means to access the basic underlying output mechanism. Moreover, conversion routines are available and it is even possible to display a predefined picture stored as a bitmap.

The modules DMModalDlgs and DMModelessDlgs provide means to enter data. Any elementary data type, character, integer, cardinal, and real (boolean is supported through check boxes), may be entered. Data will automatically be checked for syntactic correctness and for whether its value lies within a range defined by the application. Of course, strings can be entered; however, without any checking. Other supported items are push buttons, sets of radio buttons, check boxes, and scroll bars.

Module DMModalDlgs features modal dialogs. They are characterized by the fact that once a modal dialog has started, the user is forced into the correct termination of this dialog. Only syntactically correct data within a predefined range can be entered, otherwise they are rejected till correct.

Module DMModelessDlg allows to define editing fields for any of the elementary data types and other dialog items within an ordinary window as managed by module DMWindows. Consequently, the user may interrupt or resume the modeless dialog at any moment, the same way as he or she may switch his attention from one window to another.

The module DmAlerts may be used to display warning or error messages.

The module DMFiles provides simple means to store or retrieve data sequentially on a disk file. Files can be searched or created by means of dialog boxes. All elementary data types, characters, strings, integers, cardinals, or reals, may be written or read. Several files may be accessed simultaneously.

Module DMLanguage lets define a particular language as the current language. All modules of the "Dialog Machine" will adjust automatically to the language chosen, and the programmer may also implement another language by writing his own implementation for this module.

Module DMConversions offers conversion routines to convert numbers of type CARDINAL, INTEGER, or REAL to a string or vice-versa.

Modules DMSystem and DMBase export system-specific objects, such as hardware dependencies. For example, the screen resolution is exported by module DMSystem. Module DMBase is not to be used by the

"Dialog Machine" client, it only serves the "Dialog Machine" itself and its portability to other machines (e.g. Sun).

The "Dialog Machine" is used by importing any of the objects from any of the library modules into a program module. The importing compilation unit, a program module, forms the base of the application to be developed. At minimum, it must contain the statement activating the "Dialog Machine". Hence, the simplest, still executable program consists of five lines only. Typically, the application module contains a set of procedures which are called, when the associated menu items are chosen (Fig. 2). In addition, it contains statements installing these procedures into the "Dialog Machine". These statements are usually executed before the last statement of the program, the statement which starts the "Dialog Machine". Such a program module requires compilation (plus linking, depending on the Modula-2 implementation) and is then ready for execution.

The latest version 0.3 of the "Dialog Machine" has been implemented using the newest Modula-2 implementation MacMETH [Macintosh Modula ETH]. MacMETH was developed at the Dept. of Computer Science at the Swiss Federal Institute of Technology Zurich (ETHZ) under N. Wirth, the creator of Pascal and Modula-2 (MacMETH consists of a 1-pass compiler, symbolic debugger, editor (displaying compilation errors), linker respectively application maker, loader, plus a shell to quickly switch between compiler, editor, and debugger or execute a program).

THE USE OF MATRIX-ENVIRONMENTS IN ENGINEERING EDUCATION

Many of the algorithms used in control theory, and especially those used in the time-domain, are composed of simple linear-algebra primitives such as matrix operations ("+", "-", "*", and "/"), eigenvalue operations and other matrix manipulations/decompositions. Another characteristic of control theory is the habitual use of graphs and graphical methods, particularly during frequency domain design. Traditionally, both of these approaches required the use of pencil and paper. With the slide-ruler you could only draw straight lines (!), and it gave you little assistance during matrix operations. Not even the advent of cheap pocket calculators helped much (although you could use the programmable ones to calculate the points to plot by hand). Similarly, few computer CACSD programs existed, and the few existing ones were normally not very sophisticated. At best, a CACSD program of the 1970's gave access to the most common linear-algebra routines over an interactive question-and-answer interface of some kind. The dialog was totally controlled by the computer and any deviation from the pre-programmed path was impossible (particularly distressing, when after a false numerical input the user had to continue a meaningless conversation until the computer produced some mendacious results).

In 1980 MATLAB (MATrix LABoratory), a new interactive program giving access to the linear-algebra routines of LINPACK and EISPACK, appeared (Moler, 1980). Although intended for numerical analysists, the control community soon realized that MATLAB (or extensions thereof) ideally suited for their needs. MATLAB is in the public domain and soon the phrase "the best piece of software available for 75 bucks" (distribution cost) was coined.

However, some people wanted more and had the money to pay for it. Therefore, soon packages derived from MATLAB but incorporating further control-related algorithms and the so needed graphical output sold for $20.000 to $60.000 (commercial price). As they got more wide-spread and cheaper, the term matrix-environments was coined and refers to all (CACSD) packages having a userinterface similar to that of MATLAB. Interestingly enough, almost all developments of new CACSD-packages commenced after 1981-82 based upon MATLAB in some way or another (Little and co-workers, 1984; Rimvall & Cellier, 1985; Walker and co-workers, 1982).

After this uninhibited praise of MATLAB, those who do not (yet) know any matrix environment package deserve a short introduction:

A basic matrix-environment can be thought of as a pocket calculator working on complex matrices rather than on scalars. Matrices are entered in a straight-forward fashion using a transparent command language. For example, to interactively enter a 3*3-matrix and calculate its eigenvalues and eigenvectors using the freeform input format of original MATLAB (user input bold):

```
<>
a=[1 3 5
   7 6 5
   0 0 5];

<>
[vec,val] = eig(a)

VAL =

  -1.7202   0.0000   0.0000
   0.0000   8.7202   0.0000
   0.0000   0.0000   5.0000

VEC =

  -0.7408   0.3879  -0.4000
   0.6717   0.9982  -2.2000
   0.0000   0.0000   1.0000
```

The matrix environments are normally implemented as stack-machines with the dimension of the different matrices limited only by the total amount of available computer memory (and the patience of the user entering the matrices).

Basic operations in these matrix-environments include many linear-algebra functions such as inversions, transformations, and decompositions. Moreover, the user can define more complex operations on his own by clustering primitive operations using an interactive "programming language". Thereby, structures such as IF-THEN-ELSE, FOR-LOOPs and WHILE-LOOPs are supported by all matrix environments, some also let the user define functions with parameters and local variables. The following example shows how a function for calculating a controllability matrix can be defined in CTRL-C:

```
// [qs] = CONTROL(a,b)
[ma,na] = SIZE(a);
[mb,nb] = SIZE(b);
IF ma <> na,...
      DISPLAY('Non-square A-matrix'), ...
ELSE IF ma <> mb,...
      DISPLAY('Unequal number of rows in A,B'),...
ELSE qs = b; l = b;...
     FOR i=2:ma, l = a*l; qs = [qs,l];
```

While the matrix environments retain all attributes normally associated with command-driven interfaces (flexibility, speed, etc.), they through these structured language elements also provide the user with an algorithmic interface. It makes the system extendable -, this in particular has made the matrix environment so popular and, in our view, so superior to all other CACSD approaches.

In control-oriented matrix environments, one or several of the following additions are made:

- control-oriented data structures other than matrices (like transfer-function matrices, linear and nonlinear system descriptions)
- algorithms usable in control-theory
- interfaces to nonlinear simulation languages
- commands and software for generating general graphical output (and in particular for generating frequencyplots and time-histories)

Even if this short survey of matrix environments did not convince everybody to spend $75 (or more), we do not hesitate to state that the present matrix environments represent the state of the art in CACSD tools today. However, are they valuable and useful in control/engineering education?

One of the two main educational goals of introducing computer tools in (control) education, namely to the student familiarity with a wide range of software tools which he of she will encounter later as a professional, is certainly fulfilled by the matrix environments. These environments are here to stay and will be developed further. Therefore, a basic knowledge of their functionality will be just as important to a control scientist as a solid basis of structured programming is to a software engineer.

The second educational goal, namely to speed up or increase learning of the subject matter, can be attained by the use of matrix environments if they are employed properly:

- Matrix environments are inherently complex. Yet with a gradual introduction, the student can master their use within one course. A first exercise should probably not exceed the complexity of our initial eigenvalue example.

  Although the original matrix environments contained only one user interface, the interactive command language, some modern extensions support alternative input-modi. For instance, in IMPACT (Rimvall & Cellier, 1985) the user can instantiate a question-and-answer dialog whenever he does not know how to complete a command input. Such multi-mode dialogs could aid the introduction of matrix environments to students.

- An indiscriminant use of preprogrammed algorithms is dangerous, in particular, in the first problems of each new chapter. The student should be allowed to "program" his own algorithms, like the one in our controllability example, otherwise control theory algorithms are viewed as a set of black-boxes.

- Ideally, matrix environments should be used as a base tool throughout the control education. However, their use should be complemented with special-purpose software having simple-to-use interfaces for "canned experiments" and exercises involving little or no direct numerical (matrix) calculations.

THE USE OF SINGLE PURPOSE PROGRAMS

Single purpose programs are used in the teaching process for a very specific purpose. The goal of their use may be to illustrate difficult parts of control theory or to acquire specific knowledge in a certain domain. Four typical examples in use in the teaching at the ETH Zürich are the following:

1. Parameter tuning for three-term controllers

   Several rules of thumb are available for the tuning of three-term controllers. By designing such controllers and testing them by simulation, the student may get a feeling for heuristic tuning methods, and he may also to his surprise find out that different tuning algorithms (Ziegler-Nichols, Chien-Hrones-Reswick, "Betragsoptimum", etc.) yield quite different results.

2. Design of state variable feedback controller and observer

   Such an exercise may be designed with several goals in mind. One is certainly to get acquainted with problems with many parameters. Even for a second order example, there are at least 6 parameters (two feedback coefficients for the controller, and two for the observer, the sampling time and the static gain). It is therefore desirable to have a good procedure for the design available. Another goal is the comparison of designs by pole-placement or Riccati design techniques. The student may compare the two by working through a set of examples. He can do his exercise with very little knowledge of the computer that is used by providing him with an environment as the one given in Figure 4.

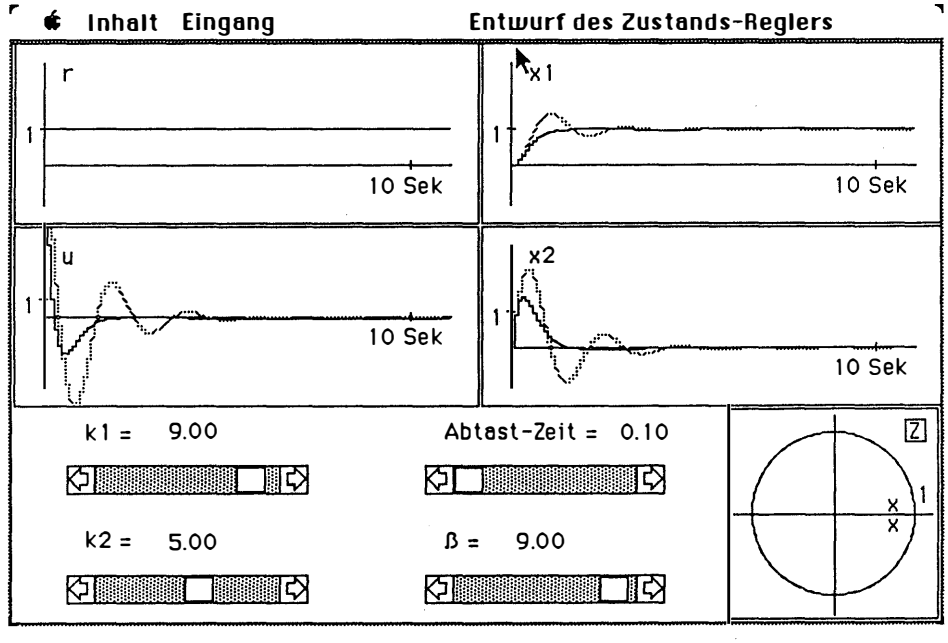

Fig. 4. State variable feedback control

3. Two-term controller on a plant with saturation

   Nonlinear phenomena are especially well suited for demonstration by computer simulation. The student may experiment with linear tuning methods and with anti-reset windup circuits.

4. Sequencing control

   Despite the fact that sequencing control is very important in practice, it is treated only marginally in most control curricula. The main reason is probably that sequencing control systems are inherently nonlinear and consist of parallel processes. Here again, simulation plays an important role. The student may learn basic concepts such as reachability of deadlocks by experimenting with small examples. He may also investigate his own design. Lisp and Prolog are suitable languages used for the implementation of simulations, SLAM is used for queuing systems.

DISCUSSION AND RECOMMENDATIONS

Different possibilities for the use of simulation and CACSD in engineering education have been presented in the last sections. The question to be discussed here, concerns the suitability of the different approaches in a given environment.

No general advice can be given as the situation may differ very much from school to school and from program to program. In general terms, our experience can be summarized as follows:

Teach your own subject: Students are supposed to study a certain subject in a course, i.e. linear control, nonlinear control, identification, adaptive control, etc. They should spend only limited time learning how to use a computer (operating systems, languages). Either this knowledge is already available, or the computer-specific characteristics should be hidden from the student as well as possible.

Introduce tools carefully: Students should familiarize themselves with one or two commercial packages in the area of control. This should be accepted as a learning goal, and sufficient time and a sound introduction should be provided.

Provide an environment: If several groups team up, a teaching and learning environment of considerable potential can be made available. The teachers and students of ETH have for example the following environment available at no cost on the Macintosh:

MacWrite, MacPaint, MacProject
Pascal, Modula-2
Prolog, Xlisp
Poly (a simple introductory CAD-system)
DialogMachine

This allows for a concentrated effort to create courseware. The students can also be asked to write their own programs for certain simple tasks.

Set goals according to the time available: Student exercises and projects may be of very different durations. There is a considerable difference between 1 to 4 hours exercises and a 200 hour project. A good solution for a brief introduction to a subject may be very different from the one adopted for a large project.

Train the teachers: Good teaching should be based on personal experience. Lecturers and assistants must be thoroughly familiar with the subject they teach, including the hardware and software used during the exercises. From our experience, a group entering the field of computer based teaching needs a lead of six months to one year before any teaching should be done on a large scale.

Limit the number of machines and packages: The market in this field is very dynamic. A group could easily spend all its time and effort evaluating machines and software. It is very reasonable to limit the products supported and to take decisions for a certain amount of time. Well established products with good support are often more effective for teaching than the most recent hardware without software.

Avoid the NIH effect: 'Not invented here' is often a solid barrier to use a product, especially in the European universities, where even textbooks are used very seldomly but are replaced by handouts produced locally. Good software is produced in many places. The attitude to use only packages which have been produced locally is unreasonable.

Use a user model: A consistent user model and the corresponding man-machine interaction simplifies the task for the developer and user of program packages.

TRENDS

A reasonable assessment of the value of the teaching methods discussed in this paper will only be possible after gaining considerable classroom experience in different surroundings. From this, one can conclude that it is useful to continue the exploitation of the field in different directions, following the guidelines proposed in the last section. A considerable influence is expected from a widespread use of graphics for input and output and from the use of artificial intelligence techniques such as expert systems or object centered programming techniques for user guidance and for help in problem solving.

CONCLUSIONS

Different ways to use computers for teaching control systems by using simulation and computer aided control system design techniques have been shown. The teacher has a wide choice of options ranging from simple single-purpose programs to sophisticated design packages commercially available. The experience gained from experimenting with these tools at the Swiss Federal Institute of Technology has been summarized. Recommendations for the use of software are provided. Much time and effort can be wasted by careless use of computer facilities in the teaching process. The paper advocates a carefully planned approach to the problem of teachware design, implementation and use.

REFERENCES

Allenspach, H., and Schaufelberger, W. (1985). The Use of a Home Computer for Computer Control Experiments. IFAC/IFORS Conf. on Control Science and Technology for Development, Beijing, PR China

Baker, N.J.C., and Smart, P.J. (1983). The SYSMOD Simulation Language. In W. Ameling (Ed.), Proc. 1st Europ. Simulation Conference ESC'83. Informatik Fachberichte, Springer-Verlag.

Fischlin, A. (1986). Simplifying the usage and the programming of modern working stations with Modula-2: The "Dialog-Machine". In prep.

Little, J.N., Emani-Naeini, A., Bangert, S.B. (1984). CTRL-C and matrix environments for the computer aided design of control systems. Proc. 6th Int. Conf. on Analysis and Optimization (INRIA). Lecture Notes in Control and Information Sciences, 63, SpringerVerlag.

Little, J., Mooler, C. (1985). PC-MATLAB User's Guide. The MathWorks, Inc., 124 Foxwood Rd., Portola Valley, CA 94025, USA.

Mansour, M., Schaufelberger, W. (1981). Digital Computer Control Experiments in the Control Group of ETH Zürich. IFAC World Congress, Kyoto, Japan.

Mansour, M., Schaufelberger, W., Cellier, F.E., Maier, G.E., Rimvall, M. (1984). The Use of Computers in the Education of Control Engineers at ETH Zürich. Europ. J. of Educ., 9, pp. 135-151.

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

Nievergelt, J., Ventura, A., Hinterberger, H. (1986). Interactive Computer Programs for Education, Philosophy, Techniques and Education. Addison-Wesley.

Rimvall, C.M., Mansour, M., Schaufelberger, W. (1985). Computer Aided Design of Control Systems, an Integrated Approach. Proc. 3rd IFAC Symp. on Computer Aided Control System Design, Pergamon Press.

Rimvall, M., and Cellier, F.E. (1985). A structured approach to CACSD. In M. Jamshidi, and C. Herget (Eds.), Advances in Computer-Aided Control Systems Engineering. North-Holland.

Schaufelberger, W., Good, H., Itten, A. (1986). Education for Microprocessor Application in Control. IFAC Symp. on Microprocessor Application in Process Control. Istanbul, 22.-25. 7. 1986.

Schaufelberger, W., Sprecher, P., Wegmann, P. (1985). Echtzeitprogrammierung bei Automatisierungssystemen. Teubner Studienbücher Elektrotechnik.

Walker, R., Gregory, C., Shah, S. (1982). MATRIX, a data anlysis, system identification, control design, and simulation package. IEEE Control Systems Magazine, December 1982.