Future Directions in Computer-Aided Control System Design Software Development

J. Douglas Birdwell
Department of Electrical Engineering, The University of Tennessee, Knoxville, TN

ABSTRACT: Issues in the development of a new package of computer-aided control system design software are presented. The scope of the project is limited to the organization of existing results into a unified body of software, and the development of a standardized set of tools and services which allow long-term maintenance and growth of the package and support the user community. Issues which influence language selection, graphics requirements, core algorithm selection, model description requirements and software tools, and future hardware expectations are addressed.

Introduction

Computer-aided control system design (CACSD) software is designed to increase the productivity of engineers and theoreticians and should be implemented in an interactive environment for maximum effectiveness. User-friendliness is necessary for the new or occasional user, and flexibility must be maintained for the sophisticates. Two environments are necessary to provide assistance to both the end user and the implementer. A high-level command language environment is needed for engineers who use the package as a design tool in their applications. This environment should minimize the programming overhead required to use the package. A second, parallel environment is needed in the implementation language which can provide many of the high-level services to theoreticians involved in the design and testing of new algorithms and methodologies. A design objective of this environment would be the reduction of the programming overhead involved in accessing existing algorithms, data structures, and services. Both environments need advanced graphics and data handling capabilities, and need to maintain hardware independence.

Software system description tools aid both the end user and the implementer. Input/output support data base management are included here. Common definitions for data structures, algebraic manipulations on systems, and a core of basic operations are necessary for efficient implementation of algorithms. A "macro" capability allows extension of these tools in particular applications. Fortran, though widely accepted, does not provide the flexibility needed in CACSD. The tools need to be available in the implementation language and the user interface.

The package design should allow algorithm independence; the algorithms are expected to change over the package lifetime and should not affect the remainder of the implementation. All implemented algorithms should be reliable; a well-defined error/exception interface should be used to alert the user or other software of problems.

The remainder of this paper is a survey of some of the issues which should be considered in order to meet the CACSD software design goals stated above. It is intended that this paper provide a basis for discussion of the proper goals which a CACSD package should address and the issues which would influence its development.

Robert P. Broadwater received BSEE, MSEE, PhDEE degrees from Virginia Polytechnic Institute and State University in 1971, 1974, and 1976, respectively. In graduate school, he specialized in electrical power systems, modern control theory, and nuclear power. He has work experience with the General Electric Industrial Controls Department and the United States Army Material Command, and Babcock and Wilcox. While with Babcock and Wilcox, Dr. Broadwater performed applied research in nuclear and fossil plant controls. He was involved in the development of the Babcock and Wilcox Advanced Control Research Facility in which control studies are performed using real time, digital simulations of power plants. In nuclear plants, he has been involved in developing improved reactor coolant pressure and feedwater system control strategies. In fossil plants, he has worked on the application of identification to improve steam temperature control. Dr. Broadwater is currently with Tennessee Technological University where he is teaching power systems and is in the process of developing courses dealing with power plant dynamics and control. He is also currently performing work for the Tennessee Valley Public Power Authority, Tennessee Valley Authority, and Oak Ridge National Laboratory.

February 1983
**Language Selection**

The principal languages which can be considered for a CACSD package are Fortran 66 and 77, Ada, Algol 68, and PL/1. In order to make an intelligent choice, the impact of the selected language on several issues must be assessed. The final selection of issues and their relative importance are open questions; for this discussion, the following list is considered:

1. Package maintenance
2. Availability
3. Verifiability
4. Portability in an interactive environment
5. Documentation standards
6. User-friendliness
7. Efficiency
8. Available development and maintenance tools
9. Error/exception interface
10. Feasibility of an "algorithm-independent" package
11. Upward mobility
12. Compatibility with other languages

A table could be developed which ranks each language in every issue, assigns a weight representing the relative importance of each issue, and delivers a figure of merit for each language. I will mention only those aspects which are either very good or quite bad in discussing a given language.

I will first consider Fortran. It is very difficult to verify code written in Fortran; this is because it is not a strongly typed language; i.e., the user is not forced to define variable types. Structured programming techniques are quite artificial in Fortran 66; this has improved in Fortran 77. Unfortunately, a separate verifier must be used to ensure good programming practices. Fortran 66 is not a portable language in an interactive environment, mainly because each machine uses a different representation and/or storage method for character data. Both Fortrans have two other strikes against them. No way of defining a user error/exception interface exists; by this, I mean the execution environment does nothing more than check for the usual machine arithmetic violations. Secondly, it is difficult to write a package which is independent of the algorithms; if one wishes to change an algorithm, it is quite likely that many other routines which use that algorithm's implementation must also be changed. This is because no provisions exist for dynamic storage allocation, and all workspace used by a subroutine which has variable dimensions must be passed in the parameter list. Fortran 66 has one feature in its favor; very efficient compilers and execution environments exist on most machines.

Ada was designed as an implementation language for large software packages. The Ada compiler, along with the Ada language system [1,2], were designed to satisfy the problems of software maintenance, verifiability, portability, and documentation. Unfortunately, an Ada compiler is not currently available on any machine, although this situation should change within the year with the introduction of the VAX ALS. Questions on Ada's efficiency and the availability of development and maintenance tools must await an implementation. Two areas in which Ada should do well are error/exception handling, where user-defined exceptions can be processed, and algorithm independence, since dynamic storage is available.

Algol 68 has many of the features of Ada and PL/1, but will probably be overshadowed by the capabilities of Ada. Algol 68 has two major problems. Many inconsistencies in the various implementations exist, especially in the input/output routines, making portable interactive implementation of a package difficult. Secondly, very few development or maintenance tools exist for Algol programs.

PL/1, like Fortran, has very efficient execution environments, although compilation can be costly. Unfortunately, complete implementations of PL/1 are available on only a few machines, and PL/1 programs are usually not portable.

In conclusion, the choice of language appears to be either Fortran or Ada. The disadvantages of Fortran, even in the 1977 standard, are many, but it is widely available. Ada solves many of the problems of large package development and maintenance; it remains to be seen how universal Ada will become.

**Graphics**

Two questions must be addressed in the design of the graphics subsystem for a CACSD package. First, what are the low-level graphics requirements for CACSD? Second, are currently available packages and standards adequate?

Given a set of low-level graphics routines which have a well-defined, standardized interface, portable software can be written to perform the advanced operations required by CACSD. It is critical that a graphics interface standard which is hardware independent yet flexible enough to accommodate the capabilities of a wide variety of hardware be established. Arguments exist in the control community that standards cannot be developed; however, such standards have been developed in operating systems for device-independent input/output. There is no reason to expect graphics hardware to present intractable problems.

The ACM Graphics Special Interest Group (SIGGRAPH) has been working on the standardization problem, and has proposed the ACM/SIGGRAPH Core Graphics System [3]. Several implementations of this proposed standard exist, including DIGRAF from the University of Colorado [4], VGM developed by Bell-Northern Research and DI-3000 from Precision Visuals, Inc. Similar device independent graphics packages are DISSPLA and DATAPLOT [5]. In addition, the American National Standards Institute (ANSI) has formed a Technical Committee on Graphics which will eventually result in an ANSI graphics standard. The International Standards Organization (ISO) also has a Graphics Subcommittee [6]. In view of the efforts at standardization, the controls community should assess the adequacy of the proposed standards in terms of CACSD requirements and provide input into the appropriate standardization committees.

**Core Algorithms**

A core set of algorithms must be implemented in any package which standardizes basic operations that are common to many higher-level problem solutions. The boundary between the core and the high-level routines is left fuzzy; basic matrix operations should be in the core, while linear system simulation routines would probably not be included. The large grey area is deliberate, since the boundary location depends on how structured an implementation is desired. The concepts described are applicable at any level of specialization.

Four software design goals should be addressed during core package development. I) The error/exception handling characteristics of all the procedures should be consistent. II) The visible portion of the implementation should not depend on the algorithm used to solve a specific problem. III) Unreliable software, where an error occurs without notification of the user or other software, must be avoided, and IV) the visible
interface for a given problem should not depend on the data type. To meet the above goals, the core package should be specified by first identifying the set of problems which it is to solve. Examples of problems which would probably be contained in the core are algebraic Riccati equation and linear equation solutions, numerical integration, eigenvalue/eigenvector analysis, and basic matrix operations. For each problem input and output parameters and error/exception information can be identified.

Algorithms can then be selected to solve the core problems. For some problems, several appropriate algorithms may exist; there is no reason the package cannot take advantage of these alternatives. An excellent survey of available algorithms has been published by Laub [7]. One question is in order concerning well-conditioned problems and numerically stable algorithms. Should we limit a package to the solution of only well-conditioned problems, and can we insist on numerically stable algorithms? I suspect not, because engineers need to be able to solve problems which can be ill-conditioned or for which no algorithm exists which is guaranteed numerically stable; however, users can expect a warning to "proceed at their own risk" and notification of errors, condition bounds and other exceptions.

At that point, an error/exception interface can be standardized. Sufficient information must be made available to the user or his software through this interface to guarantee the core software's reliability. Examples of the types of information are matrix condition estimates, failure to compute an eigenvalue, and invalid data. From the design of the error/exception interface, the user can expect reliable software; furthermore, if the interface is well-designed, when improved algorithms are developed, they can be incorporated in the package without requiring changes in the applications programs.

Allowable data types can then be identified for each problem, standardizing the data formats for the package. The visible portions of all the procedures used to solve the core problems can be specified at this point, independently of the choice of algorithm. In the future, core algorithms can be changed without modifying the visible attributes of the core package; thus, no modifications will be necessary to software using the core package.

In conclusion, the two most important issues in the core package design are the standardization of the error/exception interface, and the choice of problems rather than algorithms for inclusion in the package. The design of the visible portion of the package independent of the algorithms, and the insistence on software reliability are necessary goals. Lastly, it should be noted that these goals are in conflict with a Fortran implementation; it is difficult to ensure algorithm independence in portable Fortran, and the implementation of a standardized, complete error/exception interface in Fortran is almost unattainable.

Tools

A quality package needs not only state-of-the-art algorithms, but also a variety of software tools. These tools are the difference between a subroutine library and an easily used package. The tools can be grouped in four areas: I) Software support in the implementation language, II) data base management, III) end user support, and IV) package development, maintenance, and verification aids.

Software support in the implementation language includes model description aids, common definitions for data structures, algebraic manipulations on systems, and basic matrix operations. Model generation requires I) the formulation of differential and difference equations, 2) the manipulation and reduction of block diagrams to obtain DE's, 3) the ability to define new subsystems, using a combination of the implementation language and previously defined components, 4) the interconnection of subsystems, which are either user-defined or are from a library, 5) the mixing of continuous- and discrete-time subsystems, 6) the ability to use both explicit and implicit models, 7) automatic modification of an existing model, and 8) input signal definition. Examples of packages with model generation capability include EASY5 (available through Boeing Computer Services), CSMP, and SPICE. In addition, extensive research has been done under the direction of K. Astrom at the Lund Institute of Technology on structured modeling languages [8]. EASY5 contains advanced modeling aids, but lacks the ability to define new subsystems using pre-defined subsystems. CSMP is limited to SISO components. SPICE, a circuit analysis program, allows user-defined components.

Model description aids rely on the standardization of data representation, algebraic manipulations on systems, such as cascading two systems, and basic matrix operations. Examples of important data structures in linear systems design are matrices, linear systems, linear feedback, and filters. If these structures are standardized, inter-procedure communications become much easier. In a language such as Ada, new data types can be defined which users can refer to in their programs. This reduces the user's programming overhead substantially over what is required in Fortran or PL/1. Once the data structures are defined, operations can be defined on those structures, making the work involved in system interconnection more natural and less prone to errors.

Data base management includes the standardization of input/output support routines and the storage format of data on on-line devices. These functions are useful in any application, but are necessary when a high-level command language is defined. Data base management then allows the storage of intermediate results, whose structure may be quite complex, and their subsequent retrieval by a new command process.

The command language should parallel the capabilities given a user writing in the implementation language. The purpose of the command language is to allow the one-time solution of problems without requiring the detail of the implementation language. Both menu-driven and formal languages have been used. A good command language with macro capabilities is needed to allow the user to define repetitive functions as a new command. Menu-driven languages are usually cumbersome in a macro environment, especially if language constructs such as IF-THEN-ELSE, LOOP, CASE, nesting, and recursion are defined. Formal languages, while less restrictive, are more difficult for the inexperienced user to learn; however, in many operating system environments, they can be made to parallel the operating system command language. In this context, a formal language can appear as an extension to the system-supplied commands, with which the user is already familiar. Good examples of command languages which are extendible to a CACSD environment are contained in the TOPS-10/20, VM/CMS, VAX/VMS, and MULTICS operating systems. The last class of tools contains those used to develop, maintain, and verify the package and application programs. Two tools are desirable during program development: advanced editors and standards
verifiers such as PFORT [9]. Advanced editors are tailored to a specific language and perform chores such as program formatting, structuring, and variable declaration. Although certainly not necessary, they can increase the probability of producing reliable and readable code. Standards verifiers are necessary to ensure that code generated by several programmers conforms to the package specifications. The PFORT verifier for Fortran is an excellent example, and ensures that portable Fortran is generated.

During package verification, dynamic and static program analyzers are invaluable. Dynamic analyzers trace the execution of a program and can pinpoint execution bottlenecks. Dramatic increases in package efficiency can result.

In addition, a dynamic analyzer can be used to design validation tests for the package by ensuring that all portions of the package are tested. Static analyzers give information on interdependencies between the program segments in a package, and can ensure that all segments use the same data definitions. From the output of a static analyzer, when a change must be made, all points within the package which are affected can be determined. In any large package, maintenance aids for the source, object, and link file databases and updates are required.

In most languages, these aids must be written by the developer. The Ada language system is supposed to include these aids to support large software development projects.

In conclusion, this section has reviewed the requirements for software tools in CACSD package development. Four groups of tools were established, covering support in the implementation language, data base management, end user support, and package development and maintenance support.

Future Hardware

This short section only asks questions, and provides no answers. The capabilities of computer hardware are advancing exponentially. Since the design of a new CACSD package will require four or five years, the advances in hardware technology over the time period must be considered. As one scenario, imagine that a computer equivalent to a VAX can sit on your desk for $10,000 in 1985. Special purpose hardware, particularly array and vector processors, and advanced graphics, may also be available. What changes can be made in CACSD packages to take advantage of this capability?

Conclusions

This paper was written to stimulate discussion and to provide a starting basis for the development of a standard for a new CACSD package. Rather than attempting to specify the content and form of a CACSD package, it presents a collection of issues and decisions which must be addressed in the design of any CACSD package.

Acknowledgement

Conversations with the following people provided a valuable input to this paper: M. Athans (MIT), D. Castanon (ALPHATECH), B. Dowdy (TVA), R. Ellis (BCS), A. Erisman (BCS), R. Hammond (BCS), C. Herget (LLNL), A. Laub (USC), T. Reddoch (ORNL), A. Spang (GE), T. Trygar (DOE), Lt. Col. V. Mall (Ada Joint Program Office), and D. Ward (Stanford). The responsibility for the contents of this paper rests solely with the author. This work was supported by the Oak Ridge National Laboratory under subcontract 7685 project authorization X40 with the University of Tennessee.

References

[2] H. Ledgard, Ada, an Introduction and