AN INTEGRATED ENVIRONMENT FOR COMPUTER-AIDED CONTROL ENGINEERING

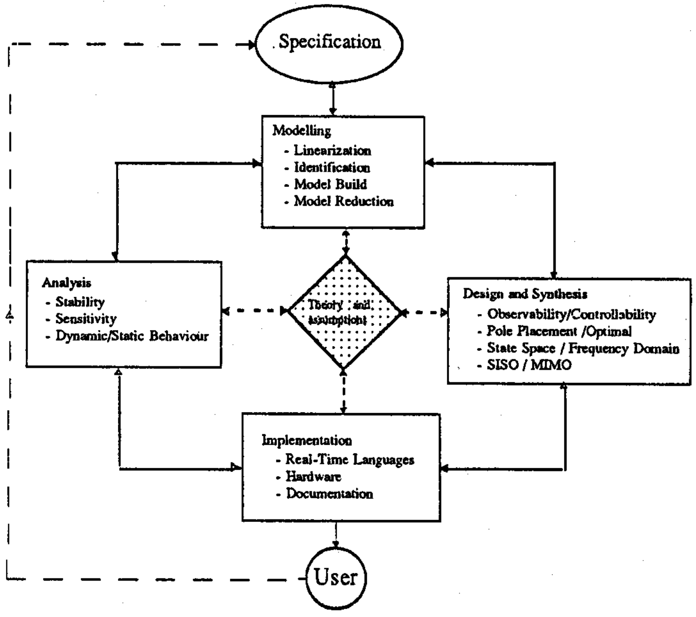
John Hickey, Digital Equipment Corp., Clonmel, Co. Tipperary, Ireland.
John Ringwood, Dublin City University, Glasnevin, Dublin, Ireland.

This paper describes the development of a Computer-Aided Control Engineering package to support the complete design cycle. It briefly summarises the ideal Control-Engineering process model used as the basis of defining the requirements of the package and summaries the shortcomings of current state-of-the-art packages against it. Then it describes the architecture designed and a prototype implementation called MSDI to support the design cycle for control systems encompassing *modelling* of the plant to be controlled, *specification* of the final objectives or performance, *design* of the required controllers and their *implementation* in hardware and software. The advantages of using an *object-oriented* design approach are discussed together with some of the software engineering aspects. Some aspects of the package are illustrated through test cases.

1 INTRODUCTION

Many software packages are currently available which assist in the design of control systems. In general, the bulk of commercial packages address only the design/analysis component of the total design cycle and can be classified as Computer-Aided Control System (CACSD) packages. A guide to many of the currently popular packages can be found in [1]. Packages that support the complete engineering of a controller can be classified as computer-aided control engineering (CACE) packages. In an effort to develop a complete CACE package, first a control engineering process model was formulated by reviewing general design theory and applying the control domain constraints. The model refined from this process is shown in Figure 1 [3].

Figure 1: Control System Design Process Model



The design specification sets the criteria or objective of the control system design. It needs to be accurate and complete to allow the ultimate design meet the desired requirements. This specification is generally non-static and changes during the design process as the designer understands the constraints and costs better.

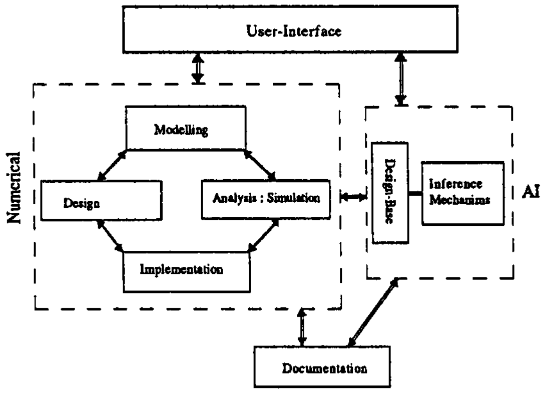
The model is the language or tool we use to discuss and evaluate various design options. This model takes many different forms from the rigorous mathematical framework of state space theory and frequency domain transfer functions to semi-formal statements about operation. It can be developed using identification techniques or from the underlying laws of nature. In practice, we usually only use one, i.e. the mathematical framework, in formal discussions and decisions despite the fact that an informal one also exists that is just as important to the success of the design [2]. This informal model is usually handled conceptually by the designer as he performs his task and is usually closely related to the formal one but cannot be directly expressed in that framework e.g accuracy and limitations of formal model.

The synthesis/analysis/evaluation feedback loop in control system design is where the various design strategies are evaluated to see their effect on the model. This cycle requires the ability to reason about and manipulate the model, to decompose, recombine, extend, reduce, analyse and synthesize changes. Evaluation in control design is usually done through simulation. This can be considered an analysis of performance. For this reason, the evaluation block can be combined into the analysis block as analysis of dynamic/static behaviour.

The controller implementation consists of hardware and software combinations. Both have a significant impact on the design viability in such terms as speed and wordsize [4]. These constraints need to to be factored into the design process to ensure its ultimate success. Production of design documentation is considered part of the implementation.

To be effective, the model for the CACE process needs to encompass model formulation, design specification, analysis and design methodologies and implementation. These components need to be accessed through a flexible and consistent user-interface. The ideal model developed for CACE is shown in Figure 2.

Figure 2: CACE Process Model



This model is broken into two main sections, numerical algorithms and AI algorithms. This is to reduce the overall complexity as many of the AI techniques that will be required in the various design phases of modelling, specification, etc. will be essentially the same. This structure will allow the incorporation of numerical software already available in an easy manner. The advances in computing hardware and compiler technology make optimising algorithms for max. speed less important than overall functionality and flexibility.

The *design-base* represents a combination of both the usual database facilities plus AI features such as rule-bases, theorem-provers, etc. . All the information known on a design is incorporated here e.g., the models, design decisions, why they were taken, rules of practice, etc. . This allows a design's history to be reviewed and analysed plus gives direct support to the documentation facility that allows customised reports to be generated.

Table 1: Requirements for a Modern CACE Package

System Component	Key Functions	Major Attributes Needed
User Interface	Schematic Capture	Block-Diagrams, Signal-Flowgraphs, Performance graphs
	Dialogue Facility	Expression-based Syntax, Matlab-like
	High-Level Graphics	Windowing, Plotting functions
	Macros	Command Macros, procedures, model macros
	Consistent	Structured language
	Help Features	Online, expandable
	Failure Response	Clear error messages
Modelling	Construction	Schematic, Control-type models e.g. State Space, Transfer Function; Identification
	Transformation	Discretization, State-Space <-> Transfer Function, Linearize
Specification	Capture	Specification Language
	Validation	Multi-Level - Behavioural, Implementation Level
Design	SISO	Pole Placement, PID, Optimal, Observer
	MIMO	Pole Placement, Optimal, INA, Observer
	Simulation	Time, Frequency Response
	Non-Linear Techniques	Linearisation, Simulation, Describing Function
	Analysis	Pole/ Zeros, Stability, Sensitivity
Implementation	Simulation	Hardware Limitations - computation time, reduced precision, Software
	Code-Generation	Controllers, Interface Routines, Jacketing Routines
Documentation	Generation	Design Decisions, Final Structure
Database Facilities	Control Data-Structures	Real, Complex, Matrices, Transfer Functions, State Space systems
	Management	Journalling, Modifying
	Interface to Foreign Code	Data, Procedures
Misc. Facilities	Design Rule Checker	
	Expert Assistant	

The major problems with existing packages are that they tend to be methodology focused (on a limited sub-set of the design cycle), provide inadequate data structures, the front-ends tend to be alphanumeric where the Engineer enters his already developed model so the system can use it. Little support is usually given for developing the model, particularly in the area of handling the "informal model". The controller is usually designed without reference to the the final implementation technology or consistency checks to validate design assumptions and actions. Generally the documentation facilities are very poor with little recording of design decisions. Most of the currently used CACSD packages were designed around MATLAB and concepts that prevailed at the early 1980s. Current efforts, such as the ECSTASY and CES projects [5,6] are attempting to overcome some of these disadvantages. An evaluation of some current packages against the CACE model are given in [6]. These shortcoming were felt to be significant enough to warrant the development of a new package which we have called *MSDI*. The prototype implementation has all the key functionality defined in Table 1.

2 Software Engineering Issues

The implementation language for the MSDI package needed to support intensive I/O activity (alphanumeric plus graphical), intensive computational activity, database manipulation, AI paradigms formulation and large system development. Using the characteristics required of a CACE package and the principles of software engineering Fortran, Pascal, C, C++, Ada, Lisp and Prologue were evaluated for suitability for this project. Ada was selected as the language best suited to implement a modern CACE package. It nearest rival, C++, failed mainly in the area of readability and the availability of good compilers. Details of this evaluation are contained in [3].

The main advantage of Ada over other languages was its powerful implementation of modern software concepts like strong typing, concurrence, modularity, maintainability, readability, operator overloading (the ability of of an operator such as +,- or * to have several alternative meanings at a given point in the program). Also as Ada was originally designed for embedded applications it was ideal to use to implement the designed controllers.

The term *package* used latter describes an Ada programme unit that forms a collection of logically related entities providing resources to application programmes that call it. In general a package implements a new data type and the operations that can be applied to that data type. An example would be a complex matrix data type which can have operations such as addition, subtraction and inversion. The package encapsulates (puts a wall around) these resources so that the details of how an addition or inversion is performed can be separated from the use of the function.

The controlled interface of Ada to other languages was very important as most of the current numerical software is written in Fortran (Eispack, Linpack, etc.). It would have been foolish to have to re-write these types of packages just because of the adoption of a new language. Thus Fortran was used as a secondary language throughout the numerical parts of the system where already implemented solutions to numeric problems were available.

For graphical routine development, VAX GKS, a device independent graphic package, was used to ensure portability between computer systems and easy maintenance. VAX GKS was considered the lowest level of a graphical entity. Thus terminal dependency was removed.

For AI routines, initially it was planned to use Prologue. But early in the project, two Ada packages were obtained, ALISP and EXPERT which forefilled the needs of AI routine development. ALISP was a generic package that provided the necessary facilities to emulate the capabilities commonly used in AI but not directly supported in Ada. EXPERT provided the facilities to do backward inferencing on a rulebase.

The design method used for software development was based on the *object oriented method* developed in [7]. The method basically consists of five steps :

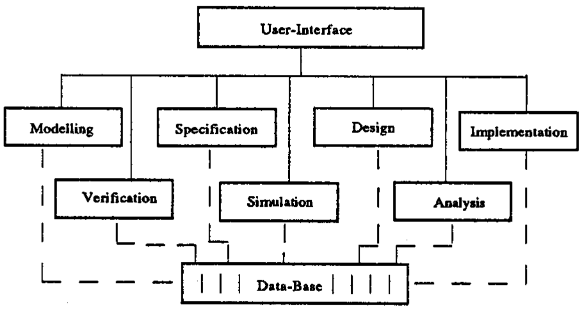
Identify the Objects and their Attributes	This requires the identification of the major components of a problem space plus their role in our model. This in our case included user-interface, database, etc.
Identify the Operations that affect each Object and the operations each Object must initiate	This requires the characterisation of the behaviour of the system or subcomponent and its objects. The semantics of the object is established by determining the operations that may be meaningfully performed by the object or on the object. Also this requires the identification of any constraints on time and space that must be observed.
Establish the Visibility of each object in relation to each other	The purpose of this step is to capture the topology of objects from our model of the system. This requires defining what objects "see" and are "seen" by a given object.
Establish the Interface of each Object	This defines what can be viewed inside a module and outside. This is where a module specification is produced in formal notation. For this formal notation the Ada language itself was used, as its specification/body concepts lends itself to this.
Implement each Object	This is the detailed coding of the body of an object.

By using Ada to develop the CACE package much of the project management facilities were already provided (in APSE). The other tools used to develop the package were the VAXset tools (Language Sensitive Editor, Debugger, Programme Code Analyser) and XD-ADA from System Designers to cross-compile code for target processors.

3 MSDI ARCHITECTURE

Most current CACSD/CACE packages have been developed using a functional decomposition method with a typical architecture like the one in Figure 3. Functional architectural design divides the CACE process into stages or functions seeking to modularize the system. A major problem seen with this method is that it forces a concentration on the operations to be performed, such as definition of a model or perform a numerical calculation.

Figure 3: MSDI Functional Architecture



This type of architecture tends towards making data global - instead of localising it where it is used, thus making the system more resilient to change. Instead an object-orient approach was used to develop the MSDI architecture.

3.1 Object-Oriented Architectural Design

The first pass of *object-oriented design* (OOD) revealed two main objects, a *complete system* entity and a *specification* entity. A *system* entity defines the current representation of a design. It includes the plant, the designed controllers and their implementations. This includes both the formal model and aspects of the informal model. The *system* entities will change over time as we proceed through the design process, adding controllers and defining their implementation. The operations that affect a *system* entity and those that it must initiate are :

a. Formulation - definition and building of a *system* model from its components (i.e. mainly the definition of the plant).
b. Modification - addition, deletion and changing of components of a *system*.
c. Design - addition of controller components.
d. Implementation - addition of implementations of controller components.
e. Simulation - simulating various stages of the *system*, behavioural and final implementation.
f. Analysis - Stability of a plant model, sensitivity, pole/zero locations, etc.
g. Verification - verifying that a *system* meets a specification.
h. Save/Restoration - saving and recalling of a *system* to/from a file.
i. Initialisation - initialising a *system* to empty.

The specification entities operations attributes are :

a. Formulation - definition and building of specification entity.
b. Modification - Addition, deletion and changing parts of the specification.
c. Profiling - generation of a specification profile, i.e. the time response and the frequency response envelops it specifies.
d. Save/Restoration - saving/recalling a specification to/from a file.
e. Initialisation - Initialising specification to empty.

As can be seen the *system* entity depends on the specification entity. The specification entity could have been included as part of the system entity definition, but as much more research than undertaken to date is needed on the specification side, it was felt better to encapsulate it on its own for future modification.

Table 2: Control Engineering Data Types

Data Type	Typical Uses
Usually High Level Language Data Type : Integer, Real, Boolean, String	Low level functions
Matrices : Real, Complex	Analysis and design algorithms, model representations
Polynomials	Dynamical equations
Transfer Functions	SISO Model representations
Transfer Function Matrix	MIMO representations
State-Space Representation	4-tuple representation of a dynamical model.
Domains	One-dimensional structure used for range descriptions
Trajectories	For time histories, signal and plot descriptions.
Table	Parsing action, Frequency response
Non-Linear Descriptions	Non-linear model descriptions

As part of the object-oriented decomposition, data-types needed to support control system engineering were identified. The lack of adequate data structures, as already outlined, is one the most serious drawbacks of many control packages. Often the *Complex Matrix* is the only data-structure supported. The MSDI architecture supports the control oriented structures of Table 2.

3.2 System Diagram

The *system* entity is defined as a type called *system diagram*. The term *system diagram* is used to differentiate it from a block diagram which is usually used to refer to the definition of a model representation composed of transfer functions. The *system diagram* can be composed of many different components such as state space models, transfer functions, signal sources, and non-linear components.

The *system diagram* is used to build up a user's model of a system from *atomic* components, *composite* components, structural interconnections and graphical attributes. An *atomic* component is an instance of a functional representation (or model template) that can be given a set of input values and can compute a set of output values via an *evaluation function* (EvFn). This EvFn is an algorithmic definition of the behaviour of a component. An atomic component is devoid of any structural information and cannot be divided into any smaller element. Each atomic component can be thought of as a generic template for a model that can have its parameters filled in to define its particular behaviour. An example of an atomic component is a discrete state space model block which has its output computed algorithmically from:

$$x_{k+1} = \Phi.x_k + \mathrm{T}.u_k \tag{1}$$

$$y_k = C.x_k + D.u_k \tag{2}$$

with u_k being passed to the EvFn for a state space object. A particular instance of this atomic component would define the values of Φ, T, C, D, initial state and sample interval.

To achieve flexibility the MSDI architecture clearly delineates between the behavioural and structural aspects of models. It provides a uniform structural modelling framework which contains placeholders or templates for behavioural descriptions. The atomic components defined for MSDI and their algorithmic form of EvFn are shown in Table 3.

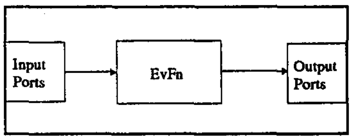
A *composite* component is defined from a structured group of atomic components. Its EvFn is defined from the way the atomic components are structured. An example of such a composite component is two transfer functions in series. Normally, a composite component will not be so simple, as various outputs of an atomic component will feed into different component blocks. Some of these blocks may be non-linear, such as a saturator, preventing computation of EvFn into a nice simple equation. Such composite components have their EvFn sectioned into an ordered list of atomic EvFns that are sequentially computed, with individual outputs computed before they are needed for an input.

Table 3: MSDI Atomic Components and their Evaluation Functions

Atomic Object Name	Evaluation Function	Comment
Continuous State Space	$\dot{x}(t) = A.x(t) + B.u(t)$ $y(t) = C.x(t) + D.u(t)$	Initial state and current state contained in model.
Discrete State Space	$x_{k+1} = \Phi.x_k + \mathrm{T}.u_k$ $y_k = C.x_k + D.u_k$	Initial state, current state and sample interval contained in model.
Continuous Transfer Function	G(s) = Y(s)/ U(s)	
Discrete Transfer Function	G(z) = Y(z)/ U(z)	Sample time included.
Gain	y(t) = K.u(t)	K is the gain constant.
Summer	y(t) = $\sum_{i=j}^{1} Sign_i.u_i(t)$	j = no. of inputs to summer. $Sign_i$ defines whether i^{th} input is added or subtracted
Step Source	y(t) = Magnitude	
Pulse Source	$y(t) = \begin{cases} Magnitude & if\ t \le Width\ ; \\ 0 & if\ t > Width\ ; \end{cases}$	
Ramp Source	$y(t) = Rate.t$	
Sine-wave Source	$y(t) = Amplitude.\sin(2\pi ft)$	f, frequency in Hz.
Square-wave Source	$y(t) = Amplitude.sign(sine(2\pi ft))$	f, frequency in Hz.
White Noise Source	y(t) = Magnitude.random(t) + Bias	random(t) is a random no. between -1 and 1.
Coloured Noise Source	y(t) = Magnitude.G(white noise)	G(.) represents a filter
General Wave Source	y(t) = Value(t);	Taken from a file.
Saturators	$y_k = \begin{cases} u_k & if\ Sat_{min} \le u_k \le Sat_{max}\ ; \\ Sat_{max} & if\ u_k > Sat_{max}\ ; \\ Sat_{min} & if\ u_k < Sat_{min}\ ; \end{cases}$	Max. and Min. Limits
Code Block	y(t) = Fn(u(t))	Fn defines a compiled subroutine
Input Block	y(t) = u(t)	Defines an input connection to *system diagram*
Output Block	y(t) = u(t)	Defines an output connection to *system diagram*

The structural framework is where component interconnections are defined. These interconnections are made up by connecting input/output (I/O) ports of a component. This connection is represented as an abstraction of a signal conductor. A port is defined as an input (or sink node) or an output (or Source node). Output ports are considered the source of a signal on a connection path and define the value of the signal on the path. Only one source port can drive any one connection path. Each path must terminate at a sink or input port. Therefore, such things as connecting an input-to-input or an output-to-output are not defined. From this, an atomic component is seen to be composed as show in Figure 4:

Figure 4: Atomic Component Structure



This template is used to define the data type of each of the atomic components. An example of this is shown in the definition of a state space model in Example 1.

Example 1: State Space Model Representation

```
type State_Space_Atomic ( Domain     : Time_Domain := Continuous;
                          No_States  : Natural     := 0;
                          No_Inputs  : Natural     := 0;
                          No_Outputs : Natural     := 0          ) is
record

     State_Space_Behaviour : State_Space_Rep ( Domain, No_States, No_Inputs, No_Outputs );
     Inp_Connection        : IO( 1 .. No_Inputs );
     Out_Connection        : IO( 1 .. No_Outputs );

end record.

where State_Space_Behaviour defines a State Space Representational type
defined in equations (1) and (2).
```

Based on these atomic components, looking at the design from a bottom-up view, several clear objects are seen to be required to support the MSDI system. Most of these data types map directly onto atomic components. Atomic components, as well as being used to compose a macro block (a composite component type), can be used to define other atomic components that depend on them. An example of this is a discrete state space model, derived from a continuous model, or an implementation of a controller derived from a controller model. The system needs to monitor and maintain these dependencies as illustrated in Figure 5.

Figure 5: Dependency Structure

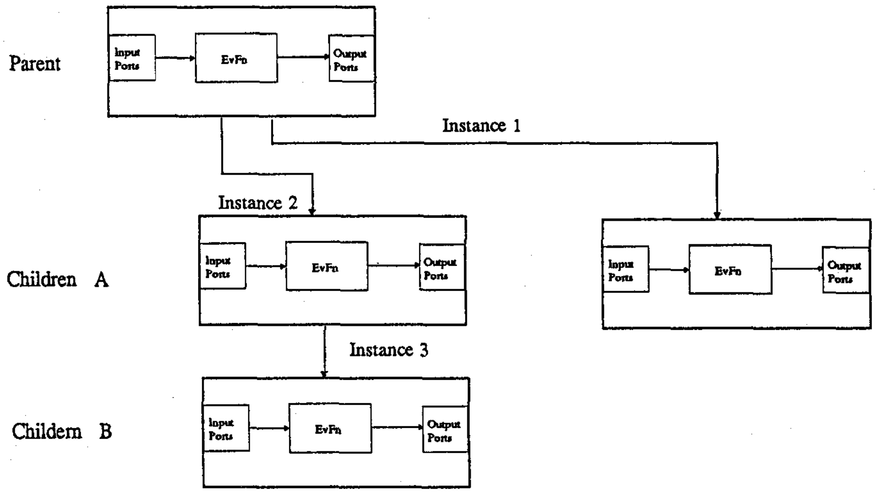


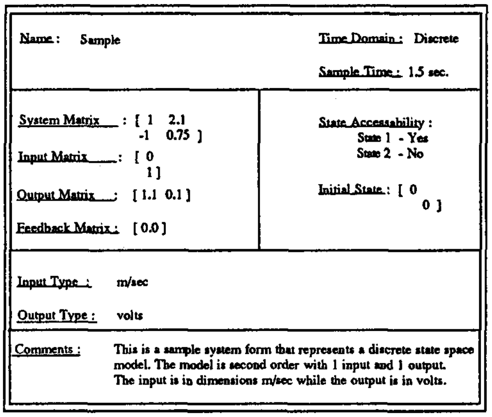
Table 4: Class of Dependent Components

Atomic Component	Parameter(s)
State Space	Sample time, transfer function model
Transfer Function	Sample time, state space model
Controller	Linear model (i.e. state space, TF or macro)
Implementation	Controller, hardware (i.e. wordlength, multiplication speed)

To achieve this, the architecture defines a class of atomic components, that can be instanced with a model parameter - similar to the idea of a generic package in Ada. This class of components is shown in Table 4. The difference between defining a state space model by (a) defining Φ, T, C, D and sample time and (b) discretizing a continuous state space model is that (a) sets up an independent state space type while (b) creates a dependent state space type which depends on the continuous model. To the user, (a) creates a new component and adds a new Icon to the terminal surface, while (b) creates a new component but does not add a new graphic representation or *Icon* to the terminal surface. If this continuous model is deleted or changed, then the dependent discrete model is updated accordingly (i.e. deleted or re-computed). These dependencies are maintained to free the designer from tracking changes from one model into another, and to prevent errors from entering into a design from the failure to update a dependent model. To ease manipulations within the package, these dependencies are maintained both ways - i.e. parent-children and child-parent.

From previous description, the *system diagram* (SD) is composed of components that are interconnected together to model a plant or system. It contains behavioural descriptions of the components, structural information on how they are interconnected and graphical data on how the model is pictorially presented to the user. Using the OOD approach, the SD was modelled as being represented by three entities : behaviour, connection and iconic. These are implemented through 3 separate packages called *system form*, *connections* and *icon* respectively. The *system form* package provides the capabilities to create, modify and delete behavioural representations. It solicits and presents this information as shown in Figure 6. This type of form allows concise and clear data entry and display. The *connections* package allows the user to select his I/O ports for connections and allows a connection to be drawn between them by the user. These connections, from a user view, can start at either an input or output port, the line drawn through various points, and finally terminated at another port. The *connections* package checks the validity of the interconnection i.e. one input port on path and one output port. The *icon* package provides the graphical attributes facilities. It allows a user to position the graphic representation or icon of a component on the display surface and to perform various manipulations on these icons such as select, move and delete.

Figure 6: System Form



Name : Sample

Time Domain : Discrete

Sample Time : 1.5 sec.

System Matrix : [1 2.1
-1 0.75]

Input Matrix : [0
1]

Output Matrix : [1.1 0.1]

Feedback Matrix : [0.0]

State Accessability :
State 1 - Yes
State 2 - No

Initial State : [0
0]

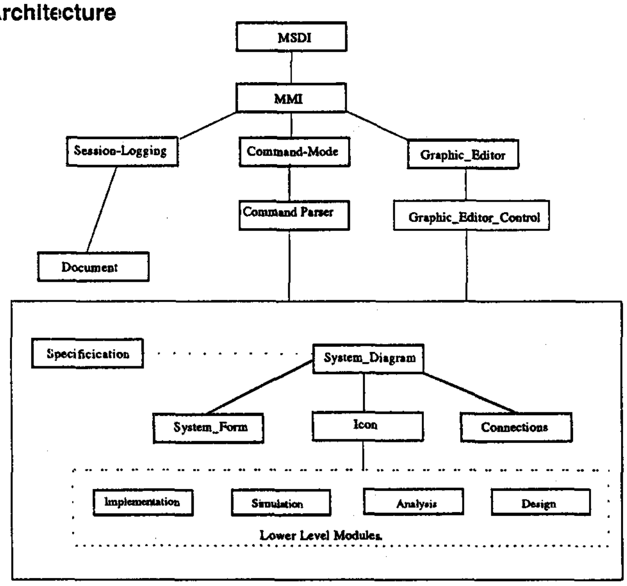
Input Type : m/sec

Output Type : volts

Comments : This is a sample system form that represents a discrete state space model. The model is second order with 1 input and 1 output. The input is in dimensions m/sec while the output is in volts.

The final architecture for the MSDI package is shown in Figure 7. Notice that this differs from most of the architectures used for existing CACSD packages. The OOD method places details of design method or simulation technique at a much lower level of abstraction.

Figure 7: MSDI Architecture



They only become apparent as operations on the *system diagram* or components of the *system diagram* - i.e. simulate the SD, design a controller for the SD, etc. . This is as it should be, as they are simply algorithmic procedures that operate on data passed to them. These design methods, analysis methods, etc. are isolated into separate packages and accessed as required. Adding a facility, say a new design method, is simple. Just add a new procedure to the package and another option to the design menu and function name to the commands.

4 User Interface

Design of the user interface is perhaps the most difficult part in designing a CACE system. It requires a delicate balance among many alternatives and apparently conflicting requirements. For example, an expert user usually prefers a terse command-driven mode of interaction whereas a novice prefers menu-driven or a detailed question-and-answer mode. In addition, a good user-interface should help turn a novice user into an expert user in a relatively short time. The approach used for MSDI is to allow 2 modes of operation. The first a command-driven environment with control-based syntax, data-structures and MSDI specific commands. The second, a menu-driven environment, using hierarchical menus. To maintain flexibility, either environment can be called from the other by a single command or key. The command mode is similar to the MATLAB interface, but supporting a more complete set of control oriented data types and removing some of MATLAB's language inconsistencies such as in polynomial/transfer function definition. An example of how MATLAB defines transfer functions versus the MSDI form is shown for the transfer function $H(s) = \frac{s}{s^2+2s-1}$

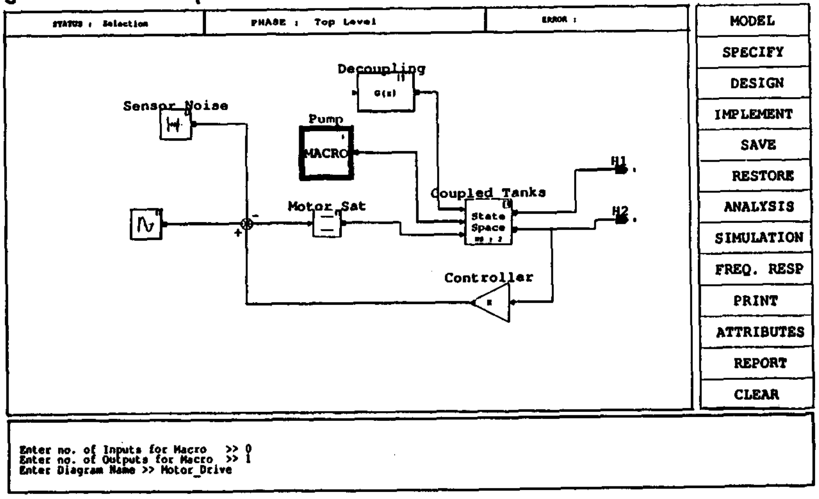
```
The MATLAB definition is :

        > num = [ 0 1 0 ]
        > denom = [ 1 2 -1 ]

while the MSDI form is much closer to the way it is written on paper:

        > G(s) = {s} / {s^2 + 2*s^1 -1}
```

Other transfer function forms such as zero-pole-gain can also be entered directly. The increased readability of this form is expected to overcome the increased number of keystrokes needed to enter it. All mathematical operations are entered as close to their written form as possible. Some examples are shown below:

Prod = A * x	Matrix multiplication by a vector or matrix
Transpose = B'	Matrix transpose
Open_Loop = G * H	Transfer function multiplication

The menu mode, implemented through what is called the *graphic editor* provides the same functionality through point-and-click menus. The main screen in the MSDI prototype is shown in Figure 8

Figure 8: MSDI Top-Level Menu



```
Enter no. of Inputs for Macro   >> 0
Enter no. of Outputs for Macro  >> 1
Enter Diagram Name >> Motor_Drive
```

The top-level menu allows the user to select the operation needed to be performed. The available options, and what they do are:

Option	Operation Performed
Model	Enter modelling phase for the current system diagram.
Specify	Enter specification phase for the current system diagram.
Design	Enter design phase for current system diagram.
Implement	Enter Implementation phase for current system diagram.
Save	Save the current system diagram to a file.
Restore	Recall a current system diagram from a file.
Analysis	Enter analysis phase for current system diagram.
Simulation	Simulate current system diagram.
Freq. Response	Compute frequency response of current system diagram.
Print	Print a system diagram to a graphic printer.
Attributes	Enter attributes setting menu to set system parameters.
Report	Generate a design report.
Clear	Clear current system diagram to empty.

4.1 Modelling and Design

The prototype implementation allows schematic-capture of block diagrams of plants or systems as the primary hierarchical method to describe models. The individual blocks can be linear or non-linear built up using icon menus and defining the internal behaviour of each block. All the MSDI atomic components defined in Table 3 are implemented. Copying, deleting, updating, combining of blocks, etc. is facilitated. The MSDI prototype allows the capture of both the formal and "informal" (e.g. state accessibility) models of objects and their constraints for direct use in design process through the system form. Model transformation tools to support linearization, model reduction, frequency to state space representations and vice versa and discretization, etc. are included in the prototype.

The design suite incorporates well-proven and numerically stable algorithms to support design of linear, time-invariant , MIMO systems in both frequency and time domains. The constraints of all design methods are built into the code to aid and warn the designer. Facilities to include new methods on-line by building up *command macros* are available.

4.2 Specification and Verification

The specification component needs to support the building of the performance criteria that a design needs to meet. To provide this functionality the MSDI prototype specification component provides a set of "primary" indicators/criteria of system performance (bandwidth or pole locations, rise-time, max. overshoot, etc.) from which all other criteria can be defined. The ability to add more to this primary set in a structured manner is available , though it needs a lot more work to fine tune. A consistency checker identifies any conflicting specifications that require trade-offs. It is planned to add a "simulator" that allows the specification entered to be graphically displayed in terms of its time and frequency domain characteristics. The verification tools have been developed initially to run separately from all other tools but ultimately will run in the background of a design session continuously verifying actions in real-time. The current prototype has minimal verification tools (only really a checker that verifies no simulation breached a specification condition). A lot more research is needed to enhance these capabilities but as control engineering is currently a well-defined body of knowledge it is believe that over the next few years the specification /verification suite will approach the required level of capability.

4.3 Simulation

The simulation engine provided is designed to cover the needs of the designer at both the behavioural and implementational levels. Continuous, discrete and hybrid simulation are catered for. Both fast interactive mode of operation for initial design debugging and batch mode of operation for detailed implementation simulation at latter stages of the design process are provided. Both time domain and frequency response can be simulated. Facilities to drive the simulation with deterministic and/or statistical stimuli are incorporated. Simulation data can be examined and recorded by pointing at nodes of a schematic as well as standard I/O recording. This can be used to dynamically collect data during a simulation from various points of a system diagram, such as an actuator output or a state variable.

4.4 Implementation of Controllers

These facilities are designed to aid in the implementation phase of the design process. A catalogue of hardware and software components are available for incorporation into an implementation of a design. These components include details of their constraints and limitations. This component library can be updated with new components (both hardware and software) by the user. The prototype allows the user to define four types of hardware types : microprocessor, A/D, D/A and PID controller. The parameters that the user can define for these hardware components are shown in Table 5.

Table 5: Hardware Component Model Parameters

Hardware Component	Parameters
Microprocessor	Wordlength, computation time for floating-point, fixed-point and integer operations, truncation method
A/D	Conversion time , word-length, analog range
D/A	Conversion time , word-length, analog range
PID Controller	L, R and C

The software algorithms are direct implementations of the controllers designed. These can be selected as optimised controller code for state feedback, state feedback with full or reduced state estimation or a PID self-tuner. Alternatively they can be built up graphically by the user to develop alternative controller strategies.

To analyse the effect of an implementation on the overall performance of the system simulation is used. This is called implementational simulation as the implementation, with its finite arithmetic and wordlengths, and inherent conversion times, is driven by outputs of the plant and desired settings. This is different from behavioural simulation where the constant matrix is driven, in double precision arithmetic with delays to conversions and computation time ignored.

Ada's ability to create new types made implementation simulation possible. The implementation simulation is based on converting a connection signal (which is a double precision number) into the wordlength of the D/A, A/D and microprocessor. Fixed-point arithmetic is the most widely used in practice because the high speeds that can be achieved compared to floating point. The fixed point format supported in the prototype is the usual two's complement representation. Here the decimal value of a number is:

$$r = 2^{-B}\left[-b_{l-1}.2^{l-1} + \sum_{j=0}^{l-2} b_j.2^j\right], where b_j \epsilon 0, 1.$$

where b_j , j= 0, ... , l-2 represent the binary digits i.e. bits, b_{l-1} carries the sign information, l is the total wordlength, and B determines the location of the binary point.

To implement this fixed point type in the package, Ada's predefined fixed-point type is used declaring it to the range and absolute accuracy of the microprocessor. Then a representation clause is used to ensure the declared type is exactly the size required. A representation clause defines how a data type is mapped to the underlying machine. Example 2 illustrates how this is accomplished.

Example 2: Simulating various wordlengths for Fixed Point arithmetic

```
No_Bits     : constant    := 8;           -- define no. of bits for
                                          -- fixed-point type.

-- Define the max. and min. values in the range.
-- Note : Must account for sign-bit.
Min_Value    : constant  := -2.0 ** ( No_Bits - 1 );
Max_Value    : constant  :=  2.0 ** ( No_Bits - 1 ) - 1;    -- Account for
                                                             -- nonsymmetric
range.

type Bit_type is delta 2.0 ** ( - (No_Bits - 1)  ) range Min_Value..Max_Value;

     -- Define the representation clause to ensure exact No_Bits used.
     for Bit_type'small use ( 2.0 **( -(No_Bits - 1)  );
```

This Bit_Type may not be implemented in exactly the number of bits we define but the compiler ensures that any arithmetic is scaled to use the absolute delta of $2^{No-Bits-1}$. The delta refers to the size of spacing between the model numbers of a fixed point type. Thus any wordlength can be created as a new data type.

During simulation the continuous signal is converted by the A/D. If saturation occurs (i.e. continuous value outside defined range for the A/D) the max. or min. of the A/Ds analogue range is used. This simulates what would happen in a real piece of hardware. The output of the A/D is the input converted to its fixed-point type. This quantizes the input as happens in real hardware. The algorithm is executed using this input. The mathematics of the algorithm are computed in the microprocessors wordlength. Then the output of the controller algorithm is fed into the D/As which perform in a similar fashion to the A/Ds but convert the fixed-point type to the continuous (i.e. double precision) type.

Computation delays are simulated by delaying outputting the result of the algorithm by a certain time. This time is computed by the programme as a sum of the delays in the A/Ds, D/As and time spent performing computations in the algorithm. Overflow/underflow condition handling can be user-defined. Obviously floating-point implementations can be handled much easier.

5 Code Generation

The prototype also supports the automatic generation of code for controller implementations. This code generation process first outputs a generic matrix package which will support the mathematical operations required. Then the D/A and A/D driver programmes are written. These programmes are entered by the user when he adds a A/D or D/A type to the library. After this the control algorithm with the fixed-point or floating point type is written output. The controller parameters are written into this procedure. This entire file, written in Ada, can then be compiled using an Ada compiler. The package is tailored to use XD-Ada, a package that can cross-compile to several target microprocessors.

This implementation support - both simulation and code generation - is seen as one of the strong points of the prototype implementation. Using this method the implementation effects on a controller's performance can be gauged. This is preferable to trying to analytically determine the effects of round-off or using "hardware-in-the-loop" methods such as used by *Matrix$_x$*. Thus a "software breadboarding" of a controller can be carried out prior to selecting, buying or building any hardware.

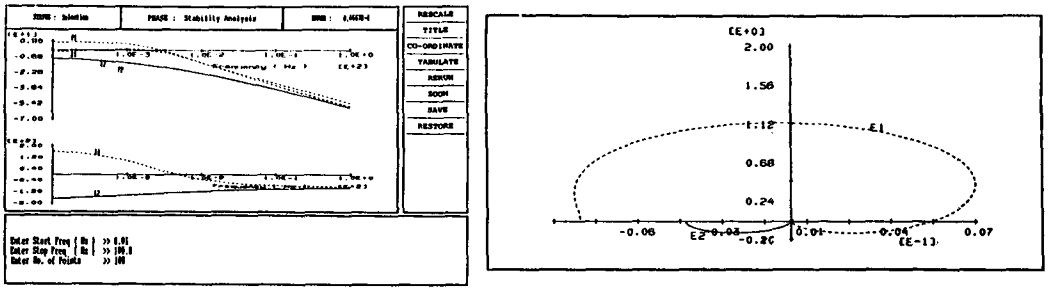
6 DESIGN EXAMPLES

The first plant model is a two-input and two output system. The model definition was added in command-mode as follows :

```
> G11(s) = ( s-1 )/( (s+1)*(s+2)*1.25 )
> G12(s) = ( s ) /( (s+1)*(s+2)*1.25 )
> G21(s) = ( -6 )/( (s+1)*(s+2)*1.25 )
> G22(s) = ( s-2 )/( (s+1)*(s+2)*1.25 )
>
> G(s) = [ G11(s) G12(s)
       >> G21(s) G22(s)  ]
>
```

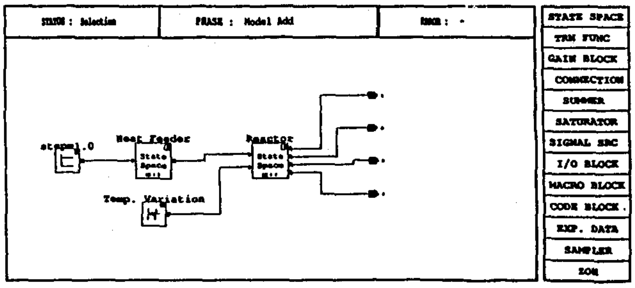
Some typical plots the package can generate such as primary indicators (i.e. the CVD and SVD for frequency range of interest) and generalised nyquist diagram are shown in Figure 9.

Figure 9: Design Example 1 : Primary Indicators/Generalised Nyquist Diagram



The second example is based on the model developed in [8]. The 7th order linear model of bed one was used to demonstrate how to design a controller. The iconic representation of the system diagram for this model is shown in Figure 10.

Figure 10: Design Example 2 : Reactor Diagram



A controller was design for the reactor model using the optimal control design facilities. The controller was computed by defining Q and R for the cost function as follows :

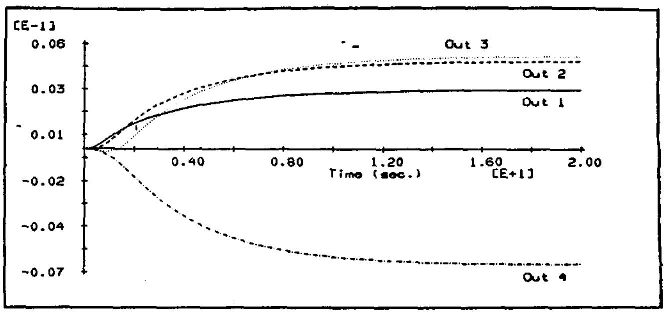
```
> Q = diag( [ 2.0 0.1 1.0 0.2 0.5 5.0 1.0 ] )
> R = diag( [ 5.0 1.0 ] )
```

The designed feedback controller was given as :

```
K = [ 0.376684  0.53378  0.50994  0.15654  1.467645  -7.8578  9.636727
      7.437257  0.34005  0.72053  0.68374  0.320174   0.2849  3.151921   ]
```

The step response for the compensated system is shown in Figure 11.

Figure 11: Design Example 2 : Step response of compensated plant



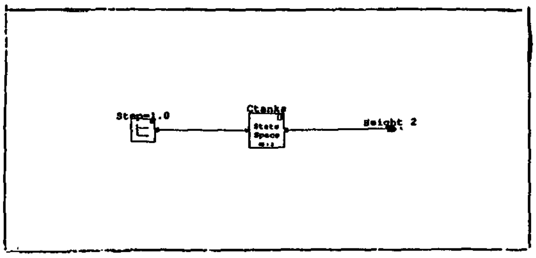
7 Implementation Design Example

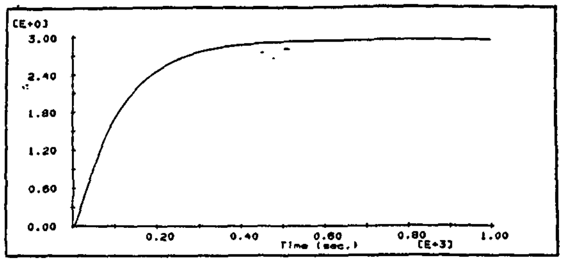
The last example of using the prototype is the design of a controller for a *coupled-tanks apparatus*. The coupled-tanks apparatus is a laboratory experimental rig that captures the basic characteristics of fluid level control problems [3]. The model for the plant was derived using the identification facilities. A step was applied to the pump drive and 50 output measurements were made at 5 sec. intervals. The model identified was :

```
G(z) =    1.946838E-2 z + 4.268818E-2
        ------------------------------
         2
        z  -1.498736 z + 5.197121E-2
```

This model has poles at (0.9539167, 0.5448191299). This model was transformed into a state space model. The representation of the system and response of the model to a unit step response is shown in Figure 12.

Figure 12: Coupled-Tanks Representation and Open Loop Unit Step Response



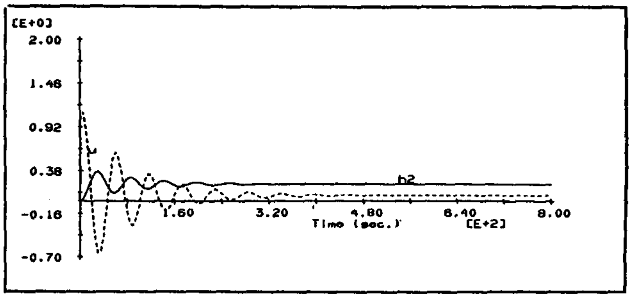


A controller was designed using pole placement. The desired poles were set to (0.6 +/- 0.2). The controller state feedback matrix computed was :

```
K = [ 0.370287919734865  -0.1012640238418579 ]
```

The step response of the compensated closed loop plant is shown in Figure 13.

Figure 13: State Feedback Compensated Plant Step response



Then an implementation was defined using a 12 bit D/A and a 8 bit A/D with the microprocessor being defined as 4 bit, 8 bit and 32 bit in turn. The simulation of these implementations for a unit step response are shown in Figure 14.

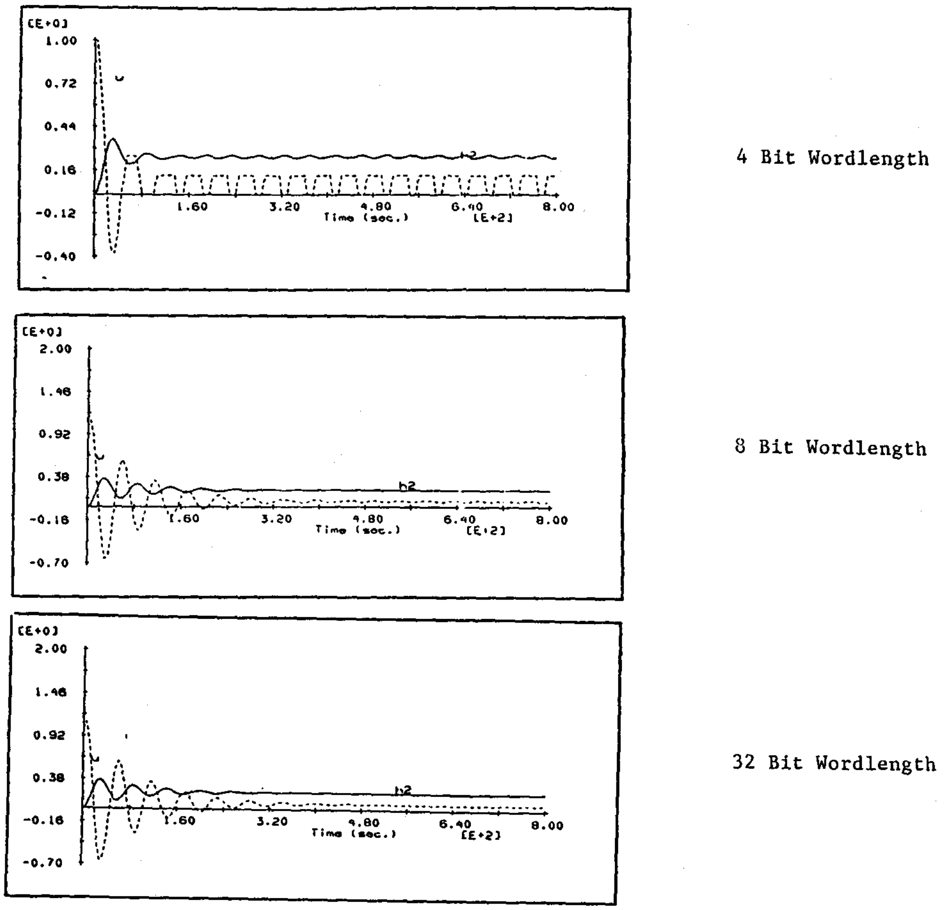
The reason for the difference in performance between the three implementations was caused by the different wordlengths used. Roundoff occurs both in the controller parameters and for the calculations made during the test runs (and can be examined by the user). The differences in the coefficients of the feedback gain matrix K is shown in Table 6. Notice that the coefficients have changed significantly for the 4 bit wordlength implementation.

Table 6: Coefficients in 4, 8 and 32 bit controllers

Wordlength	K(1,1)	K(1,2)
4	0.250	0.000
8	0.3671875	-0.093750
32	0.370287919734865	-0.1012640238418579

The roundoff in the feedback gain matrix is really pronounced in the 4 bit implementation. This together with the roundoff involved in the computations is the reason for the changed performance. Based on this, a minimum of an 8-bit wordlength microprocessor should be used to implement this controller.

Figure 14: Simulation of 4, 8 and 32 bit based state-feedback controllers



The code generation facility in the prototype was used to produce an implementation of the controllers and was run on the test apparatus using 4, 8 and 32 bit fix-point mathematics. The actual response obtained was as predicated by the simulation of the implementations [3]. The computation time for the coupled-tanks example was not a factor. The sample time of 5 sec. far exceeds the time to perform the simple calculations, even for the case of output feedback. This facility would be useful where a very fast process such as a robot is being investigated.

8 Conclusions and Future Research

A prototype package was presented that implemented an architecture to support the CACE process model. This prototype concentrated on developing the *system diagram* object. A control-based syntax was used for the input parser to allow the user enter data in a way as close as possible to the way he would write it on paper. This prototype provided facilities for the simulation of a controller's implementation and a code generation facility to increase an engineer's productivity. This code generation facility relieves the engineer of the need to write new code every time he wants to implement a controller. A few brief examples illustrated the way the package could help in the various stages of the design process.

The prototype contains over 65,000 lines of Ada code and the relative sizes of its major components are shown in Table 7.

Table 7: Relative Size of Major Components MSDI Prototype

Function	Size
User-Interface	34 %
Numerical Algorithms	26 %
Graphical Software	28 %
Symbolic Software	2%
Database / Error handling, Memory Management	10 %

The performance of the MSDI package needs to be judged on its impact on the overall design cycle. This impact cannot be measured directly without a significant amount of surveying and usage of the MSDI system. This measuring of performance, the enhancing of the specification/verification facilities and the porting to X-windows are the next steps in the package's development.

9 REFERENCES

1. Jamshidi, M. and Herget C.J., (Eds), "Computer-Aided Control Systems Engineering", North Holland, 1985.
2. Denham, M., "Design Issues for CACSD Systems", Proc. of IEEE, Vol 72 No. 12, pp. 1714-1723, Dec. 1984.
3. Hickey, J., "An Integrated Environment for Computer-Aided Control Engineering", M.Eng Thesis, DCU, Dublin, Ireland, 1989.
4. Hanselmann, H., "Implementation of Digital Controllers - A Survey", Automatica, Vol. 23, No. 1, pp. 7-32, 1987.
5. Munro, N., "Ecstasy - A Control System CAD Environment", Proc. of Control'88, Oxford, UK, 1988.
6. Hickey, J., "Survey of Current CACSD Packages", Research Report CTRU8804, Control Technology Research Unit, Dublin City University, Jan. 1988.
7. Booch, G., "Software Engineering with Ada" , Benjamin/Cummings Publishing Company, 1986
8. Edmunds, J.M., "A Design Study using the Characteristic Locus Method", in Design of Modern Control Systems, IEE Control Engineering Series 20, Peter Peregrinus Ltd., 1982