DUBLIN CITY UNIVERSITY

School of Electronic Engineering

Control Technology Research Unit

An Integrated Environment for Computer-Aided Control Engineering

by

John Hickey, B. Eng.

A thesis submitted for the degree of
Master of Engineering

Supervisor : Dr. John Ringwood

September, 1989

An Integrated Environment for Computer-Aided Control Engineering

Abstract

This thesis considers the construction of a system to support the total design cycle for control systems. This encompasses *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 main contributions of this thesis are : its development of a model for CAD support for controller design, evaluation of the *software engineering* aspects of CAD development, the development of an architecture to support a control system design through its full design cycle and the implementation of this architecture in a prototype package.

The research undertakes a review of general design theory to develop a model for the computer-aided controller design process. Current state-of-the-art packages are evaluated against this model, highlighting their shortcomings. Current research to overcome these shortcomings is then reviewed.

The software engineering aspects to the design of a CAD package are developed. The characteristics of CAD software are defined. An evaluation of Fortran, Pascal, C, C++, Ada , Lisp and Prologue as suitable languages to implement a CAD package is made. Based on this, Ada was selected as the most suitable, mainly because of its encapsulation of many of the modern software engineering concepts.

The architecture for a *computer-aided control engineering* (CACE) package is designed using an *object-oriented design method*. This architecture defines the requirements for a complete CACE package including control-oriented data structures and schematic capture of plant models. The details of a prototype package using Ada are given to provide detailed knowledge in the problems of implementing this architecture. Examples using this prototype package are given to demonstrate the potential of a complete implementation of the architecture.

Digital Equipment Corporation

Acknowledgements

I am indebted to my project supervisor, Dr. John Ringwood, for his continued help and encouragement. Thanks to all the staff of the School of Electronic Engineering at Dublin City University for their patience and help, particularly during the "playing" with the coupled-tanks. To my fellow postgrads and friends at Dublin City University, I wish you good luck with your respective careers.

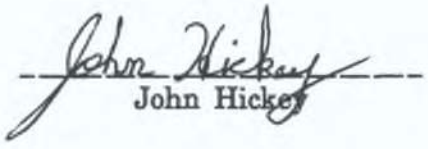
I must acknowledge the help given by Brian Rasmussen in proof-reading the thesis and helping turn it into a readable document.

Thanks to the people in Digital Equipment Corp. who gave me the time and the resources to complete this research.

Finally a special thanks to John, Peter, Marie, Charles, Jo and Seamus who encouraged me to get on with this write up and gave me the time and space to finish it.

Declaration

I hereby declare that this thesis is entirely of my own work and has not been submitted as an exercise to any other university.



John Hickey

Dedication

To my parents, as they enter their golden years.

CONTENTS

EXAMPLES

FIGURES

TABLES

CHAPTER 1

INTRODUCTION

This thesis is concerned with the development of a CAD package to aid a control engineer in taking a project from modelling to implementation. In this section, the objectives of the research are stated, motivation for developing a new CAD package for control system design is given and the thesis structure outlined.

1.1 Thesis Research Areas

The purpose of this dissertation is to present and support the thesis that an integrated *computer-aided control systems design* package (CACSD), or what is called in some literature *computer-aided control engineering* (CACE), supporting the complete design process can be constructed and would be of considerable use. The term CACSD is used throughout this thesis to refer to CAD packages that support only part of the design process (usually controller design using some numerical algorithm but not its actual implementation), while CACE is used to refer to a package supporting the complete engineering of a controller. This convention was adopted from James [1] and Taylor and Frederick [2]. This process encompasses *modelling* of the plant, *specification* of desired performance, *design* of required controllers and their *implementation* in hardware and software, plus the production of supporting documentation. The package developed is called MSDI. The name MSDI is meant to convey the intention to provide a system that covers Modelling, Specification, Design and Implementation of the controller.

Furthermore, this CACE package can be flexible enough to be an aid to engineers with varying degrees of proficiency, from the experienced professional to student or part-time users.

The thesis is concerned with the following topics :

- Definition of the design process.
- Using this definition as the basis for a CACE model to support an engineer at the various stages of the design process.
- Review of current CACSD packages against this CACE model.
- Evaluation of the software engineering issues for this CACE model.
- Development of an architecture to support the CACE model.
- Development of an implementation of this architecture.
- Evaluation of the prototype CACE package.

1.2 Motivation for Research

The research undertaken and the efforts to build a new CAD system which are described in this dissertation have been motivated primarily by the frustration of using existing packages which individually only address part of the controller design problem.

Another major factor was the noticeable superiority of other engineering areas in terms of CAD systems. Integrated packages with schematic capture of designs, simulation and verification capabilities plus implementation support have been seen since the late 1970's in VLSI. The adoption and integration of these ideas along with advances in related areas such as Artificial Intelligence and Graphics into a CACSD package was seen as an opportunity of increasing its power and usefulness.

As one looks to the 1990s the complexity of the controllers to be designed and the design solutions possible will grow. [1, 3]. A review of future complexity and technology trends for the design problem is given in Appendix C.

This project was initiated in anticipation of the continued growth and evolution of the design problem as a whole. As designs continue to increase in complexity and size the CAD environment must meet the increased demand for performance. As new design methodologies and implementation technologies continue to evolve, the CAD system must be in a position to evolve also, incorporating new algorithms and techniques. The ability to upgrade quickly and provide timely support of new technologies and processes is a key requirement of the next generation of CAD systems. It will not be sufficient in the next decade to merely tolerate change or cope with it through use of manual or bandaid stopgap measures. This new generation of CAD tools must be designed around the presumption that there will be a need for change thus ensuring that the system will be capable of evolving to meet the needs of future design teams.

Obviously another key motive was the desire to investigate and resolve the issues involved in a project covering such wide areas as design and CAD, software engineering, control theory and controller implementation issues.

1.3 Thesis Structure

The thesis is divided into three main sections: evaluation of the need for a new CACE package and definition of its requirements, development of the required software architecture and its implementation, and the evaluation of the prototype package.

Section 1, the need evaluation and requirements definition, is covered in Chapters 2 and 3. Chapter 2 defines the design process and develops a CACE model that would support this process at each phase, reviews current CACSD packages against this CACE model and highlights developments in other areas that can be used in CACE. Chapter 3 evaluates the software engineering issues involved in developing a CAD system and explains why Ada was selected as the implementation language for the package.

Section 2, development and implementation of system architecture, is covered in Chapters 4 and 5. Chapter 4 details the architecture of the package, while Chapter 5 details the implementation of the various components of this architecture in the prototype package.

Section 3, evaluation of the package, is covered in Chapters 6, 7. Chapter 6 takes several cases studies and shows how the package can be used. Chapter 7 comments on the overall research and highlights areas that need further work.

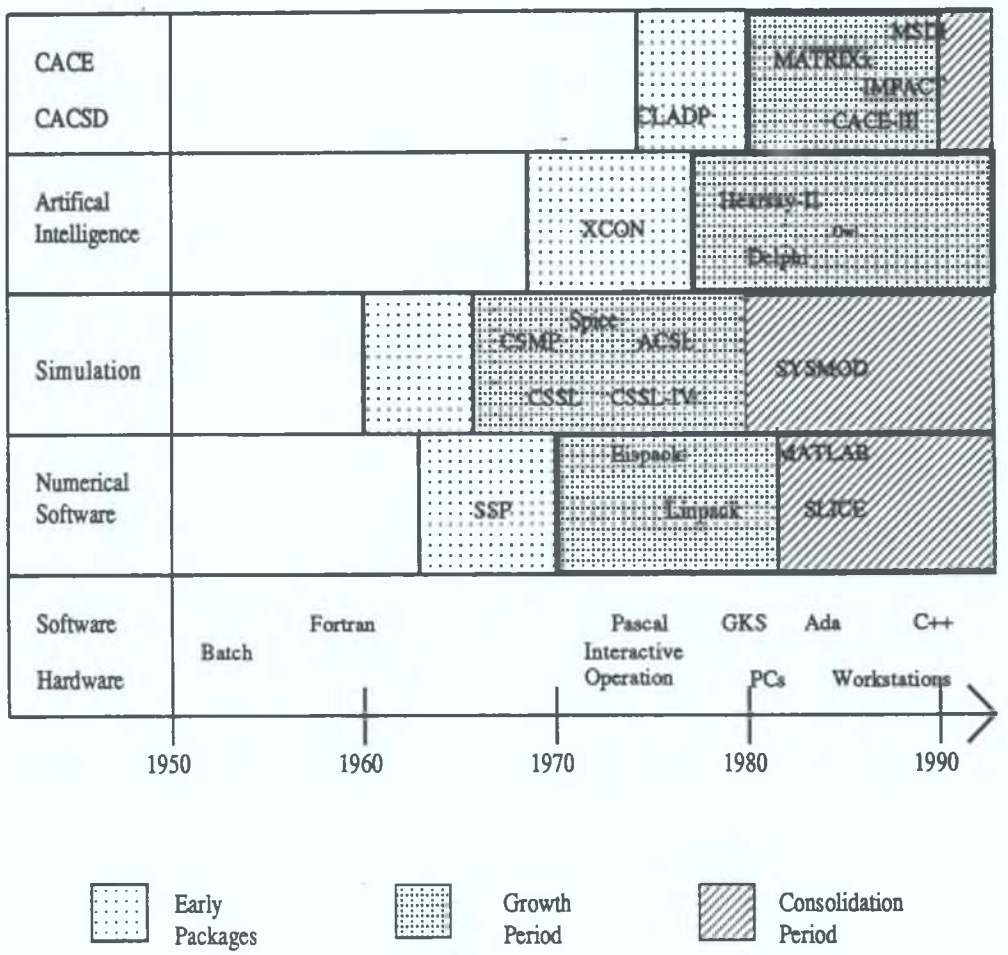
1.4 Historical Development of CACSD Tools

In the beginning (early 1960s), there was FORTRAN, paper tape, batch processing, split decks of programme cards and 24-hour programme turnaround. Line-printer style of plots of time and frequency responses were standard outputs. Elsewhere, optimal control theory was being developed and having a massive impact on control system design. Some of the work done on CACSD in this period is detailed by Melsa and Jones in [4].

In the 1970's there was a reaction against the high degree of automation contained in the state-space design methods, compared to the physical insights given by the frequency domain methods of Nyquist and Bode. Researchers turned back to frequency-domain techniques and Rosenbrock successfully fused existing mathematical methods and the emerging availability of interactive minicomputers and graphical terminals to tackle MIMO control system design, using the concept of diagonal dominance and the Inverse Nyquist Array method [5]. By the mid-1970's, libraries of control-related subroutines were being compiled. Most of the CACSD software was written in Fortran, used rudimentary graphic display facilities and was question-and-answer driven.

In 1980, Cleve Moler released his MATLAB software [6]. While not specifically intended for control system design, this software was a major advance in CACSD technology and was actively used as a basis of many other packages such such as such as PC MATLAB [see Appendix A]. Up to this, command-driven languages were felt to be too complicated for the occasional user. MATLAB was command-driven and overturned this notion, proving immensely popular with non-specialist users. A chart of major progress in CACSD and related fields is given in Figure 1.

Figure 1: Major Milestones in CACSD and related fields



The integrated design suites developed in the latter half of the 1980's are classified as either *comprehensive tools* or as *design shells* . The former tries to accommodate every possible user requirement while the latter more realistically provides the user with a framework and tools to realise objectives. CACSD software can be further classified as *code-driven* or *data-driven* . Code-driven is compiled and characterised by fast execution and reduced flexibility. Data-Driven software is interpretive, implementing operators and algorithms as data statements interpreted during programme execution and characterised by its flexibility and relatively slow execution speeds. Recent research and development in computer tools for control engineering are detailed in [8] and [55].

CAD has evolved over the last 10-15 years from simply being implementations of numerical algorithms to the powerful tools of today. Early CAD systems sought to automate the laborious and computationally intensive parts of the design process. Systems such as SPICE [9] and PNE1/6 [10] were nearly purely numerical software algorithms with very limited user-interfaces and only dealt with segments of the design cycle. During the 1980s the buzz-words in CAD have been *user-interface* and *integrated systems*. Commercial examples of systems that sought to implement these ideas are MATRIXx [11], the Federated CACD System [12] and Valid's SCALD system [13]. Another concept being actively researched is the use of AI

techniques to aid in the overall process, particularly Expert Systems. Expert Systems have been around for over a decade, from DEC's XCON of the 1970's to the speech-understanding system HEARSAY-II [14]. Two approaches have been taken in Expert System development for Control Design. One followed by James in [1] provides the designer with little scope for manipulating the design. The design follows a "Black-Box" type of approach where most of the action happens automatically. The second followed by Nolan [15] aims at building an expert system which functions as a designer's assistant aiding and prompting throughout, but with the user selecting the options and making the decisions.

The advances in hardware and software technology have greatly influenced current CAD architectures. Workstation technology allied to powerful graphical drivers and ideas such as increasing the speed of computation through parallel architectures have created opportunities for major steps forward in CAD.

CHAPTER 2

COMPUTER-AIDED CONTROL ENGINEERING

This chapter gives a definition of the control engineering design process. Then a CACE model is developed to support this process through its various stages. Current popular packages are evaluated against this model. Finally, the direction of the current research to overcome their shortcomings is summarised.

2.1 The Control Engineering Design Process

Design is one of the most complex activities that humans perform. Exactly how we perform this function is not fully understood and is the basis of much current research in psychology and artificial intelligence [16]. In control engineering, we are concerned with the *engineering design function*. Thus, design is defined as the process of turning a concept of a product (e.g. a car, rocket or microchip) into its ultimate realisation in hardware and software.

The next section reviews general design theory. Based on this, a model for the particular case of control design is derived.

2.1.1 General Design Model

The engineering design process uses the creativity of the designer to formulate alternative solutions to meet a specification. By applying his reasoning and intelligence he selects the one that optimises his resources (be they time, money, profits, bandwidth, rise-time , etc.) and satisfies the design goals. This is a very complex process that is full of conflicts and uncertainties as trade-offs between several different competing constraints need to be made. The resolution of these conflicts is based on the designer's skill and experience.

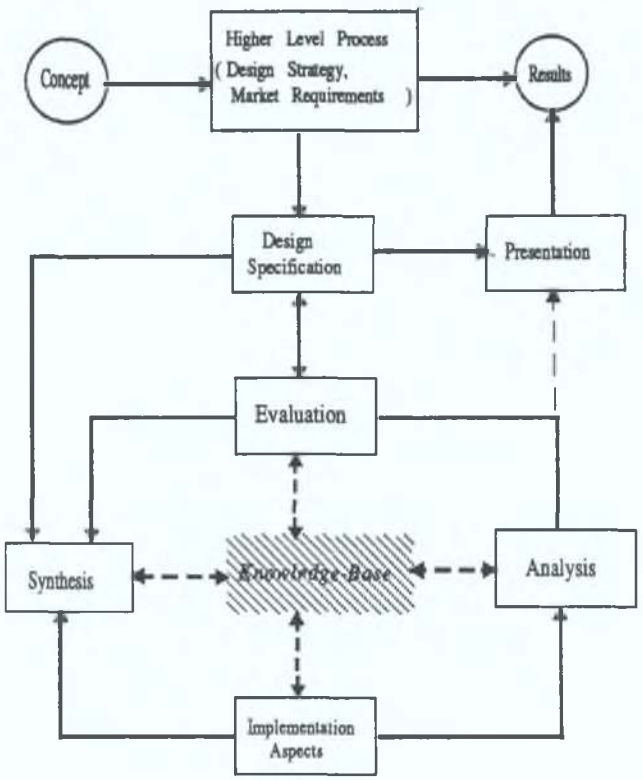
The outcome of this process is the hardware and software that combine to form the final product plus the supporting documentation, from design and performance reports to user manuals. This supporting documentation is probably just as important as the system itself. Good documentation is a prerequisite for the completion of a successful project.

During the last few decades various models of the design process have been proposed, from Asimov [17] who essentially formulated a sequential model, to Rodenacker who considered design a transformation process of meaning or attribute of an object[7]. The *system analysis and design technique* (SADT) by Ross [18] seems to be a combination of both. Lately, feedback between the various stages plus models for the intelligence required to perform the design have been included [19].

The model of the design process which it is felt covers all the major stages is shown in Figure 2. The major part of this model has evolved from many of the ideas currently being developed in the VLSI field where general design theory is much more advanced due to the complex problems to be solved [19, 20] and the financial support poured into

solving them. All design variables from product concept, market factors to technological and implementational constraints

Figure 2: Design Process Model



(speed, min. track width, etc.) are being investigated and included in the final evaluation of a design. Only through the close integration of these diverse functions into a unified system architecture, will it be possible to address all these constraints successfully during the design cycle. This is often referred to as *computer-aided-engineering* (CAE).

The basic design starts with a specification or goal. This specification, whether to control the speed of a motor to within 1% of its desired setting or to add two binary numbers is not important. The specification is derived from what are called *concept* and *higher level processes*. These blocks are included to cover the influence of markets, upper management and other factors which provide the incentives to produce a successful design. This specification is evaluated to check for completeness and that it can be satisfied within the resource constraints (i.e. time, money, people, max. response time, etc.). This evaluation can take on many forms from a project or design review [21] to a mathematical proof of consistency and completeness [22].

Man usually solves problems through building models using decomposition, abstraction and successive-refinement. Decomposition is a process by which a a complex object is broken down into smaller sections to make it easier to deal with. Abstraction is a method by which only the salient features or characteristics of an object are used to build up a simple model to reduce complexity. Successive-refinement here means the opposite of abstraction,

where greater detail is added to a model of an object bit by bit to give a more accurate representation of the object.

This process of first simplifying, then increasing the detail level of a problem (i.e. model abstraction and successive-refinement) is needed to reduce the design complexity. The psychologist George Miller stated that the limit to the number of entities humans can process at one time is roughly 7 plus or minus 2 [23]. Beyond this, our ability to manage the complexity of many entities falls off sharply. A clear example of the need and use of this process is in the development of VLSI circuits where initially the designer works at the behavioural level where only the overall function of the circuit is considered. Then this is implemented down to the logic level. Then even further detail is added when it is reduced down to fabrication mask where each function is designed down to the actual silicon level. This is an example of abstraction and successive-refinement. The original behavioural model should be contained in the final silicon structure but is usually impossible to discern because of all the low-level implementational detail. Usually decomposition of the problem also takes place as the behaviour is "broken" into separate and independent functions. Throughout design, this process of building up a model is closely related to the specification. The end result needs to satisfy this specification.

The synthesis/analysis/evaluation phase is a feedback loop where various design alternatives are investigated. During this cycle a lot of knowledge on the design is built up as we manipulate our model using decomposition, abstraction, etc. and this is represented as the *knowledge-base* in Figure 2. Often this cycle can also lead to the modification of the design specifications. Thus the two-way link between the specification and evaluation block as requirements and constraints are better understood.

The *implementation aspects* block covers technology to be employed, its constraints and limitations, the manufacturability and testability of the final design which includes such things as design for test, assembly, reliability and yield-oriented design [24, 25]. The effects and constraints of these underlying aspects often dictate certain modifications to the overall design strategy. This is often handled by design rules which guide the engineer to develop a practical solution. The product is designed using these rules of thumb and then simulated using a very detailed implementation model to verify operation. Only through extensive simulation can the final product be verified to meet the specification. Formal specification techniques have been used in some limited cases which, with the use of theorem provers, allow verification or "prove-correctness" of the final design [26]. This type of work is still only in the research stage but the advantages of such a facility is obvious as a rigorous or "nearly-rigorous" mathematical proof of the operation of the system could be derived giving far greater confidence in the final system, than simulation (which by its nature can only cover a certain number of test conditions), particularly in critical real-time or safety related systems.

The fall out of this feedback cycle is a body of results and achievements which need to be documented for presentation to the customer. The word customer here is used to define the originator of the design request. It could be an external person or body, a project team who define certain requirements that need a design solution, or marketing who define a product need. This documentation phase is often a very tedious and laborious task that can take up to 15 % of a design project's time [27].

2.1.2 Control Systems Considerations

Control systems design is a specific instance of the general design process discussed in Section 2.1.1. The basic goal of control design is to create or modify a given dynamical system so that it has the specified behaviour or set of attributes. In this section we will fill in the specific tasks that are performed in the various generic blocks of Figure 2.

For the discussion on the control design problem the concept and higher level process blocks are omitted. The detailed definition of the contents of these blocks is beyond the scope of this thesis. It is noted though, that these stages in the overall process exist and that their functions are important to the overall success of it. But, unfortunately, nearly all systems today neglect this aspect of the process and much more research needs to be done to define and clarify a quantitative relationship between them and the rest of the design process.

The design specification would generally take the form of both model formulation of the system to be created or controlled plus the specification of the goals of the final design (eg. max. rise-time, bandwidth, etc.). Both the concept of a model and of a design specification are important.

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 of the customer. This specification is generally non-static and evolves and changes during the design process as the designer understands the constraints better and the customer understands the cost of various parameters.

The model is basically 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 [28]. 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, limitations of formal model and areas of its possible extensions, etc. . Often this informal model is too complex for the designer to use effectively and results in some of its bounds being breached during the design process unwittingly by the designer unless he has a formal means of validating all decisions against this model.

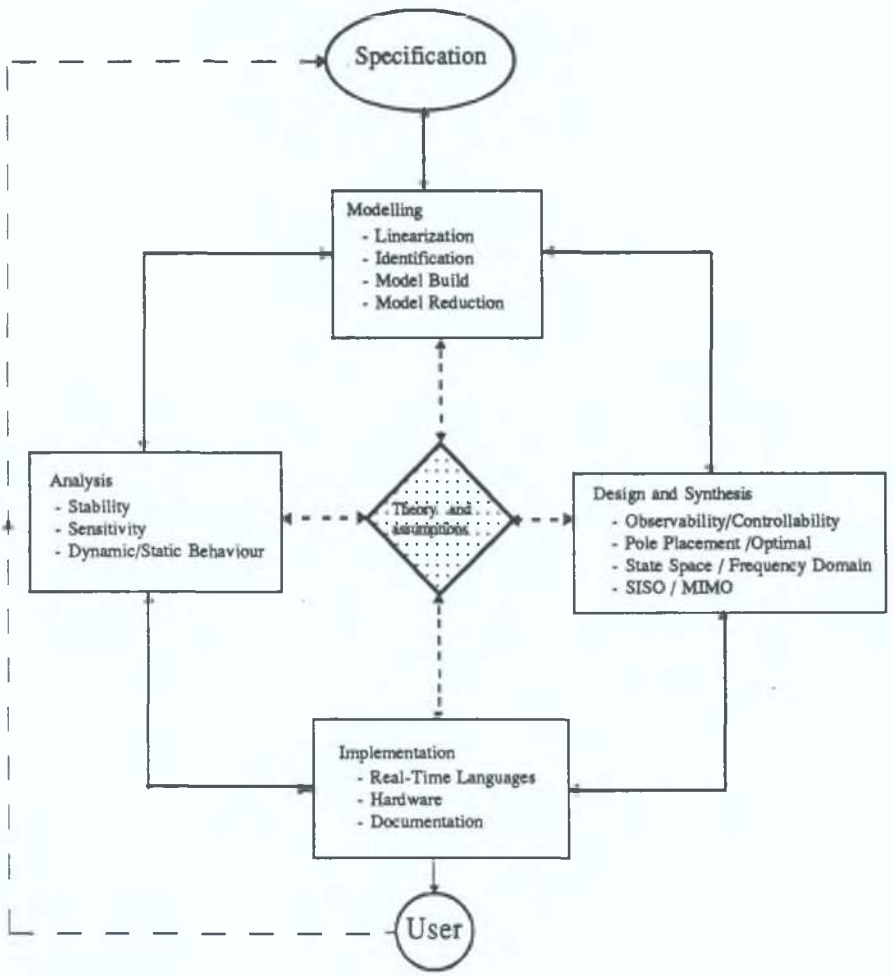
The synthesis / analysis / evaluation feedback loop in control system design is where the various design strategies are evaluated to see their effect on the model. This cycle requires the ability to reason about and manipulate the model, to decompose, recombine, extend, reduce, analyse and synthesize changes. Several different methodologies can be used depending on the level of analytical knowledge available. Evaluation in control design is usually done through simulation. This simulation shows the designer the performance of the system under various conditions in both the time domain and in the frequency domain. 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.

In our control systems design the implementation consists of hardware and software combinations. Both have a significant input on the design viability in terms of speed, memory, reliability, etc. constraints [73]. These constraints need to to be factored into the design process to ensure its ultimate success. Considering documentation as an output of the design

process, the presentation and implementation components of the general design model have been combined into one, called the implementation block, which will encompass all outputs of the controller design process.

This refined model of the *control systems design process* is shown in Figure 3. The knowledge-base components of Section 2.1.1 have been renamed as *theory and assumptions* to highlight the role of the informal model.

Figure 3: Control System Design Process Model



2.2 Computer-Aided Design Model

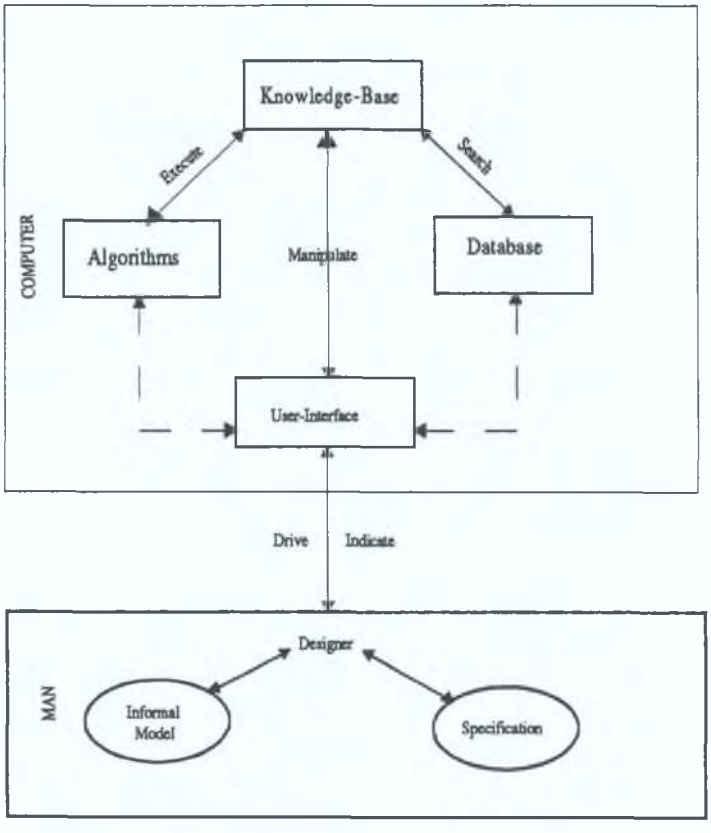
The objective of CAD is to support the design process in such a way as to increase overall design quality, reduce costs and increase designer productivity. The computer has enabled many new design methods and technologies to be used due to its vast computational ability. By introducing a computer into the design process we are increasing the complexity of it. The only justification for this is, if its introduction simplifies the overall process to a much greater extent. To achieve this most successfully, the designer and the computer need to be combined in an integrated fashion. To accomplish this we need to look at the various

strengths and weakness in terms of the design process, of both man and the computer. As Pang and MacFarlane in [29] define them we can see that the designer's function should be to set goals and argue from first principles in terms of abstract concepts. He should handle and interpret ambiguity, conflict of objectives and uncertainties in description and performance. Computers should be used to evaluate functions, execute complex procedures, search through complex data sets, correlate information, and generally perform laborious repetitive tasks. The key is to join both together in a way that frees the designer to concentrate on the task at hand rather than on how to operate the system.

2.2.1 General CAD Process

There are many different models for the CAD process but, in general, all recent models are variations of the one illustrated in Figure 4.

Figure 4: General CAD Model



The knowledge-base and the database can be thought of as the location where the model of the object being designed is stored. This model changes throughout the design process being transformed using the various algorithms available. This process of representation or model reduction, transformation and refinement is commonplace across all engineering design and the CAD process needs to support it.

The need to control these changes of the model is one of the reasons for the popularity of including AI into CAD systems. Most approaches use a knowledge base of some sort where information on the task at hand is stored. Then various inference mechanisms are used to make decisions based on this knowledge. These mechanisms can range from very simple as in SACON [30] to very sophisticated as in CONSULTANT [20]. Many commercial systems do not really have what could be called a knowledge base but merely an extensive database which can be searched and data selected to use in the execution of algorithms. Also systems vary between whether the knowledge-base, often in the form of an expert system, calls in the various algorithms and searches the database such as the approach followed by James in [1], or whether the designer directly calls in the algorithms and passes the desired data himself as in most current commercial systems such as CLADP and Control-C. Thus the dotted lines between the algorithms, database blocks and user-interface blocks in Figure 4.

The knowledge-base approach definitely offers the most effective method particularly in areas of well-defined problems. They lead to a reduction in the complexity of the problem by removing some of the entities that the designer has to deal with. These knowledge-base systems allow the designer to concentrate on the design task as he issues commands on what he wants to do rather than on how to do it. The problem with this, is that currently a lot of flexibility for the designer to experiment with various approaches and algorithms is lost. James [1] attempts to overcome this by allowing the user access to the numerical software directly if required, but the drawback of this is that the mode of operation is drastically changed thus causing the related designer reluctance to use these facilities.

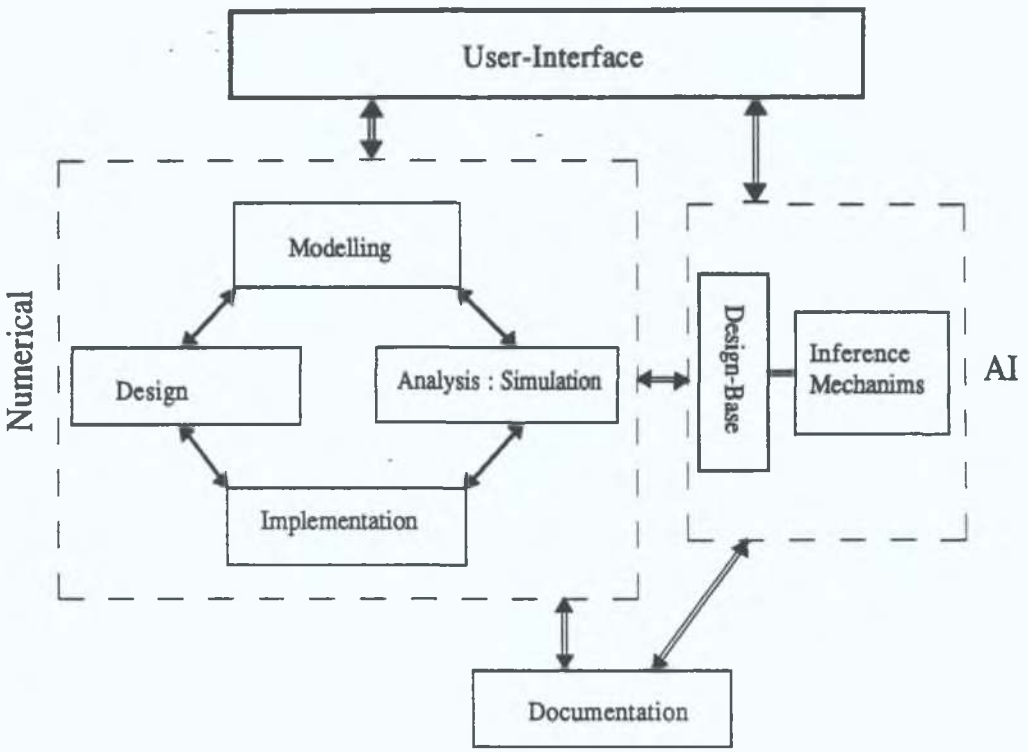
The user-interface is generally highly I/O intensive by nature. A large graphical content will be present plus symbolic manipulation based on the information being communicated, as research [31] has shown that most engineering information is best digested and interpreted in these forms. As the traditional medium for communicating control engineering information about dynamical systems has been pictorial i.e. block diagrams and signal flowgraphs, it is essential any modern CACE package provides this form of interaction with the designer. Most earlier CAD systems neglected this aspect.

2.2.2 CACE Process Model

Based on the previous sections, to be effective, the model of the CACE process needs to encompass model formulation, design specification, analysis and design methodologies and implementation details. These components need to be accessed through a flexible and consistent user-interface. The structure designed is shown in Figure 5.

The techniques needed to support each activity is discussed latter. But do note the need for close linkage between the activities. This linkage is maintained to the designer through the user-interface. Through this interface the designer is allowed to access various numerical routines and design objects.

Figure 5: CACE Process Model



The model presented in Figure 5 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. Only the objects being manipulated and goals being sought will differ. This structure will allow the incorporation of numerical software already available in an easy manner. This is the approach adopted by most recent researchers [1, 12, 32]. The advances in computing hardware and compiler technology make optimising algorithms for max. speed less important than overall functionality and flexibility. This approach also allows new numerical techniques to be incorporated into the system easier by increasing the modularity of it.

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 should be incorporated here e.g., the models, design decisions, why they were taken, rules of practice, etc. . This allows a full design's history to be reviewed and analysed plus also gives direct support to the documentation facility that allows customised reports to be generated.

The User-Interface will be similar to that described in Section 2.2.1 but with a control-theory based syntax supporting the usual data structures such as matrices, polynomials, state space descriptions, transfer functions , etc. . This User-Interface also needs to allow the ability to link into and use models and routines developed external to the package. This is important as many good models of systems already exist encoded in an algorithmic manner inside a subroutine and would be difficult and time-consuming to reformulate in terms expressible in a new CAD system.

2.3 Summary of CACE Model Requirements

A central theme of the MSDI project was that to enhance the overall design process. There must be a close integration of all steps in the process. Increasing the capability of one section in isolation from the other design activities (such as model capture, observation and interpretation of simulation results, data management, etc.) will not necessarily lead to a better design environment. A main focus of the MSDI system was to address this need. The key requirements for a CACE system to achieve this are tabulated in Table 1.

These requirements are broken down by design phase, and the key functions needed to support each phase. Then each key function has its major attributes defined. For example, the User-Interface for a modern CACE package needs to support Schematic Capture. This facility in turn needs to support the input of Block Diagrams, Signal Flowgraphs and Performance Graphs i.e. the major pictorial control system representations.

The final prototype produced does not attempt to support every one of these requirements, but a selected subset to demonstrate the power of an integrated CACE package. All the key functions are supported but all the major attributes are not. Over time it is hoped to flush out the package functionality as per Table 1.

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 MSDI system does not attempt to enforce any particular design methodology. Rather, it provides an operating environment which allows the specification and implementation of different design styles. This is achieved through the careful design of a set of atomic tools which can be easily assembled in a variety of ways, thereby supporting a large number of design methodologies and verification styles.

2.4 Evaluation of Current CACSD Packages

Most of the current commercially available CACSD packages are methodology focused with little development put into the overall need of the control engineer to perform his task. Researchers such as Rimvall [33], Goodfellow and Munro [34] in recent years have striven to overcome this problem by concentrating on the user-interface aspects of packages. But, in general, these packages cover only part of the design cycle, the control strategy formulation. 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. Then the controller is designed through an iterative process without reference to the the final implementation technology or consistency checks to validate design assumptions and actions. Generally the documentation facilities are very poor. See Appendix A for details of the packages examined. An evaluation of them against the CACE model defined in Section 2.2.2 is shown in Table 2.

Table 2: Evaluation of CACSD Packages

Function		Lund Suite	MATLAB[1]	MATRIXx[2]	CLADP	CACE-III
User Interface	Graphic Input	None	None	Good	None	None
	Dialogue Facility	Fair	Good	Good	Fair	Fair
	Macros Features	Fair	Good	Good	Fair	Poor
	Consistent	Fair	Good	Good	Fair	Poor
	Help Features	Good	Fair	Good	Fair	Good
	Failure Response	Fair	Fair	Fair	Fair	Fair
Modelling	Construction	Fair	Fair	Good	Fair	Fair
	Transformation	Good	Good	Good	Good	Good
Specification	Capture	None	None	None	None	Good
	Validation	None	None	None	None	Good
Design	SISO	V. Good	Good	Good	Good	Fair
	MIMO	Good	Good	Good	Good	None
	Simulation	V. Good	Fair	Good	Good	Fair
	Non-Linear Tech.	Good	Poor	Poor	Good	Fair
	Analysis	Good	Good	Good	Good	Fair
Implementation	Simulation	None	None	Poor	None	None
	Code-Generation	None	None	Fair	None	None

[1] Including control systems, identification and MIMO toolboxes

[2] Including system build and autocode facilities

Table 2 (Cont.): Evaluation of CACSD Packages

Function		Lund Suite	MATLAB[1]	MATRIXx[2]	CLADP	CACE-III
Documentation	Generation	Poor	Poor	Poor	Poor	Fair
Database Facilities	Control Data Structures	Poor	Poor	Poor	Poor	Fair
	'Foreign' Interface	Poor	Fair	Fair	Poor	Fair
	Management	Poor	Poor	Fair	Poor	Fair

[1] Including control systems, identification and MIMO toolboxes

[2] Including system build and autocode facilities

A table of packages that can be used for different parts of the design cycle is given in Appendix A.

2.4.1 Shortcomings of Current Tools

While numerous CACSD packages exist today, most are focused on a particular aspect of the design problem. The lack of packages that facilitate the implementational aspects of the problem is a prime example. The technology to be used has a major bearing on the overall success of the design. Such factors as speed, quantitization error, etc. need to be evaluated in a software breadboarding manner prior to hardware implementation. Usually several independent packages are tied together to attempt to cover this problem but the different structures and design goals of these packages make it very difficult to integrate them easily, see [12] for an example of this approach.

Little validation and recording of design decisions is done in current systems and usually the only form of verification of a design is through extensive simulation of a mathematical model of the system and its controller, divorced from its implementational aspects.

Other disadvantages that characterise current tools are :

- large monolithic programs which are hard to modify to fit varying design needs—verification and regression testing of these large programs has become a great challenge in itself.
- many different, hard to learn, mostly textual, user interfaces with conflicting, inconsistent, or redundant command syntax and semantics.
- inadequately integrated with the rest of the CAD tools such as waveform preparation, schematics capture (design capture including complete controller specifications), and postprocessing tools making sharing of models, pertinent data, calculations, and algorithms difficult.
- inadequate simulation data collection, observation, and storage of simulation history for future analysis.
- little provision for callable/shareable internal functions or procedures. (Almost no code is reused, so development and maintenance of the tools is expensive.)

- fundamental, often used, features and functionality may have been implemented through layers of less than optimal workaround code often making fundamental features very expensive timewise to use.
- inadequate accuracy and consistency checking of current models.
- Inadequate data structures, usually only the complex matrix, and database facilities.

2.4.2 Current Research to Overcome Shortcomings

Most of the currently used CACSD packages were designed around MATLAB and concepts that prevailed at the early 1980s. Some of the current efforts, such as the ECSTASY project [35], are attempting to overcome some of these disadvantages but as discussed in [55] none of these developments focus on the specification / verification or implementational needs of a general CACE system.

Rimvall's IMPACT (see Appendix A and [36]) focuses on the data structures problems while Barker, Chen and Townsend are driving research at the University of Wales, investigating the issues involved in the graphical input of control problems for their CES package [37].

A major step foward in CACE would be the formulation of standards for CACE packages to follow. Standards for command language, standard data formats and programme interfaces are currently being worked on by an International Federation of Automatic Control (IFAC) workgroup on CACSD standards.

2.5 Summary

A model of the CACE process from design specification to implementation has been developed. It indicates the need for the support of specification of the desired controller plus the need to add implementational detail into the overall problem to ensure that the final design does in fact meet the customer's goal. The design process was broken into the following stages :

- Model Formulation
- Design Specification
- Design / Analysis / Simulation , etc.
- Implementation
- Documentation

The requirements for a CACE to support this process were given in Table 1. This defined the starting point for the development of a new package for CACE called MSDI.

CHAPTER 3

SOFTWARE ENGINEERING ISSUES

The purpose of this chapter is to examine the requirements of the MSDI project from a *software engineering* viewpoint. It explains the reasons for the adoption of an object-oriented design approach for the software development and the selection of Ada as the major language to be used to implement the proposed CAD system. Also the need for a secondary language, FORTRAN, plus the requirement for a graphics package to support I/O interface development is explained. The development tools that were needed to support the design and coding phases of the project are detailed.

3.1 Characteristics of CAD Software

The end goal of this research project is to develop a working prototype of a CACE package that supports the entire process of designing control systems from initial problem formulation through to physical implementation of the controller. The system requirements , defined in Chapter 2 can be broken into categories to analyse the basic characteristics of the requirements in general. The categories are :

User Interface

The user-interface is highly I/O intensive by nature. A large graphical content will be present plus symbolic manipulation due to the nature of the information being communicated, as research [31] has shown that most engineering information is best digested and interpreted in these forms. Most early CACSD systems neglected this aspect as the concentration was on numerical issues, but in recent years the design of user interfaces has become a dominant issue in the development of CACSD tools [8] [36] [38].

Numerical Algorithms

These will be computationally intensive with some, particularly in the simulation area, being concurrent. An example of this is the splitting of a simulation of a large model into several parts and simulating each different piece on a different processor. Most will be dealing with matrix computations by the nature of Control Theory [39] [40].

Data-Base

The data-base refers to those parts of the system responsible for storage/retrieval of data from data structures built up by the system whether located in trees/graphs in memory or file manipulation [41].

Artifical Intelligence

AI will be a part of most sections of the system like the User-Interface but for the purpose of this analysis it is sectioned it off to itself. These will deduce if certain goals are possible from a given set of facts. These will be very computational intensive [22] [29].

3.2 Language Requirements

Thus from the brief analysis of the major components of the system, the implementation language, or combination of languages needs to support

a. Intensive I/O activity, alphanumeric plus graphical
b. Intensive computational activity
c. Database manipulation
d. AI paradigms formulation

Also other major characteristics of the system are .

e. It will be Large (> 50,000 lines of Code)
f. It will be designed for Long-Life , i e , up-gradable
g. Portable
h Incremental (grow bit by bit)

A study by Rimvall in [33] illustrates the evolution of CACSD packages from being almost solely numerical algorithms to the much more balanced distribution between the software components of today. A typical breakdown of code in modern packages is shown in Table 3

Table 3: Relative Size of Major Components of a CACSD Package

Function	Size
User-Interface	30 %
Numerical Algorithms	30 %
Graphical Software	20 %
Symbolic Software	5 %
Database / Error handling, Memory Management	15 %

Thus from this it can be seen that no one component of the system should dominate in the selection of the main design language or in the selection of the design method to be used to develop the system

3.3 Software Engineering Considerations

Many concepts developed in the last ten years have radically altered ideas on software development. It is cliche to say there is a software crisis [42] The development of a *common programming language* was one attempt to help overcome this crisis [43] The fundamental cause of the software crisis is that massive, software-intensive systems have become unmanageably complex Many of the symptoms of the software crisis cited by Fisher in [43] are unresponsiveness, unreliability, costliness, lateness, not portable and inefficient systems The importance of this crisis in most large projects today cannot be overlooked as software costs tend to be dominant by a 4.1 factor over hardware. In this research project it probably was a 40:1 factor. To overcome the crisis a structured approach throughout the project using modern software engineering concepts was required.

Recognising that change is a constant factor in software engineering development , four goals first presented by Ross, Goodenough and Irvine in [44] still dominate ·

Modifiability	- Controlled Change
Efficiency	- Use of resources in an optimal manner
Reliability	- Prevent failure in conception, design and construction plus recover from failure
Understandability	- Accurate Model of real world

To achieve these goals a structured design approach is required Many design methods have been developed in the last ten years but they can be divided into one of three classes [45]

Top-Down Structured Design	The system is designed from a functional viewpoint , starting with a top level design and progressively breaking this into a more detailed design This method is exemplified in [23]
Data-Structure Design	This method focuses on the data of the problem and its related structures This method is very popular in database type applications using variations on the Jackson Method [46]
Object-Oriented Development	This method focus on the objects of a problem domain and the interrelationships between them This method has grown out of work by Parnas, Liskov, Guttag and Abbot [47] This type of approach by its nature encapsulates many of the design principles of software engineering such as abstraction, information hiding, modularity, etc

The selection of the implementation language(s) for a project involves tradeoffs In general no one language will satisfy all design requirements perfectly The tradeoffs involved are the increased complexity caused by the introduction of another language to alleviate problems caused by deficiencies in one language versus design difficulties caused by working within the limitations.

The design method to be used and the implementation language are related Particular design methods are more suited to certain languages than others. Thus the implementation languages(s) was selected and then the design method

3.4 Language Evaluation

Using the characteristics of CACE packages defined in Section 3.1 and the principles of software engineering the following criteria (in decreasing order of importance) were selected to investigate the merits of various languages for this project.

a. Maintainability / Readability
b. Modularity / Hierarchical
c. Data Structures / Types
d. Efficiency
e. Concurrent Facilities
f. Portability
g. Error Handling
h. I/O Facilities
i. Familiarity

Maintainability and readability mean the ability of a language to aid in the process of maintaining a programme throughout its lifecycle. Such facilities as separate compilation effect this. The ability of a language to reflect the meaning intended by the algorithm designer is also important.

The modular/hierarchical criteria has two aspects : the language's support for subprogramming and the languages's extensibility in the sense of allowing programmer-defined operators and data types. By subprogramming, it is meant the ability to define independent procedures and subroutines and communicate via parameters or global variables.

The data structures/types criteria mean the ability of a language to support a variety of data values (integers, reals, strings, pointers, etc.) and nonelementary collections of these such as arrays, records and dynamic data structures such as linked lists, queues, stacks and trees. Also these criteria cover the ability of the compiler to support type checking to verify correct usage of types.

Efficiency covers both compilation and execution. Efficiency of execution will include both the time and space characteristics of languages.

The concurrent facilities criterion will measure the ability of a language to support multiple threads of controls.

Portability refers to the ability of easily moving a programme from one computer system to another. This includes compiler availability on various systems plus the inherent difficulties in porting various languages from machine to machine.

The error handling criterion refers to the facilities a languages has for detecting and handling error conditions when they occur in normal execution of a programme. This refers to such things as hardware interrupts on overflow or memory exhaustion.

The I/O criterion covers both terminal I/O and file handling capabilities of a language. The reason I/O and file handling capabilities are of a relatively low weighting is because most systems offer these in their operating systems and if necessary these system routines could be directly called for the internals of a programme to perform I/O. This is the case for VMS

and UNIX. If the language is modular enough this approach could be localised to limit its effect on portability of the overall system.

The familiarity criterion refers to the history or maturity of a language. This includes available textbooks, design examples and experiences.

A summary of the languages considered is given in Table 4 More details on the reason behind the ratings are given in Appendix B

Table 4: Language Evaluation Summary

	Language						
Criteria Evaluated	Fortran 77	Pascal	C	C++	Ada	Lisp	Prologue
Maintainability/Readability	Poor	Fair	Poor	Poor	V Good	Poor	Poor
Modularity/Hierarchical	Poor	Fair	Fair	Good	V Good	Good	Fair
Data Structures/Types	Poor	Good	Good	V Good	Good	Fair	Good
Efficiency	Excell	Good	Good	Poor	Good	Fair	Fair
Concurrent Facilities	None	None	None	Good	Good	None	None
Portability	Good	Good	Good	Poor	Good	Fair	Fair
Error Handling	Poor	Poor	Poor	Good	Good	Poor	Poor
I/O Facilities	Fair	Good	Good	Good	Good	Poor	Poor
Familiarity	Excell	V Good	V Good	Poor	Poor	Good	Fair

From this evaluation, Ada stands out as the best current language to implement a CACE package. It nearest rival, C++, fails mainly in the area of readability and the availability of good compilers The availability of good compilers for C++ can be expected in the near future, but the language's terseness will always be a problem for large systems development compared to Ada

3.4.1 Benchmarking

To aid in the selection of the implementation language, the Whetstone Benchmark was run for four language, Ada, Fortran, Pascal and C to compare their run-time efficiency The results of this benchmark are shown in Table 5 All the compilers were VAX compilers, and the benchmark was run on a MicroVAX II system

These initial result indicated that Ada was the FASTEST language despite all its run time checks On investigation it was found that the OPTIMISE=TIME figures for Ada reflect the results of automatic Inlining of one of the benchmark subroutines. The OPTIMISE=SPACE gives a "truer" reflection of Ada's performance in real-world conditions

For the benchmark trials we can see that Fortran is generally the most efficient though overall there is no major difference between the performance of the languages. For more details on this benchmarking see [48].

Table 5: Whetstone Benchmarking on MicroVAX II

Language	Version	Optimisation	Checks	Stubbed Math	Stub Rating %	Real Math	Math Rating %
Ada	1 1	Time	None	1739	100	1086	100
		Time	All	1653	94	981	90
		Space	None	860	49	714	66
		Space	All	816	47	670	62
		Nooptimize	None	721	42	564	52
		Nooptimize	All	627	36	515	47
Fortran	4 1	Optimize	None	992	57	796	73
		Optimize	All	744	43	624	57
		Nooptimize	None	889	51	727	67
		Nooptimize	All	720	41	596	55
Pascal	3 2	Optimize	None	1011	58	787	72
		Optimize	All	755	43	628	58
		Nooptimize	None	955	55	782	72
		Nooptimize	All	716	41	625	58
C	2 1	Optimize	None	733	42	339	31
		Nooptimize	None	633	36	336	31

Note : The MicroVAX II was running VMS V4 3 and had 9 Mb of memory.

3.5 Language Selection

Ada was chosen as the main language in which to implement the system in because

a. 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), etc ,

b. the overall design environment that Ada provides (APSE) eased the job of project management,

c. the VAX/VMS implementation provides an implementation that is only slightly less efficient than Fortran in run-time performance;

d. the possibility of using Ada as the language for implementing the designed controllers, thus allowing the possibility of reusing large sections of code;

e. use of a language sensitive editor reduces the burden of its verbose syntax;

f. it is easy to interface with other languages.

These advantages made it suitable despite current lack of experience with Ada A clear example of its advantages (in terms of readability/maintainability and operator overloading) is shown in Example 1 and Example 2. These examples display two code fragments, one in

Fortran, and the other in Ada, for the solution of the optimal control problem using dynamic programming. This basically entails minimising the quadratic cost function of (1)

$$J = \frac{1}{2}\vec{x}_k^{\,t}\, Q\, \vec{x}_k^{\,t} + \vec{u}_k^{\,t}\, R\, \vec{u}_k \tag{1}$$

by recursively solving (2) and (3) backward in time to find the feedback matrix F, using P(∞) = 0, until the process converges to a constant matrix.

$$F_{k-1} = (R + B^t\, P_k\, B)^{-1}\, B^t\, P_k\, A \tag{2}$$

$$P_{k-1} = A^t\, P_k\, A - F_{k-1}^t\, (R + B^t\, P_k\, B)\, F_{k-1} + Q \tag{3}$$

where A and B are the system and input matrix to the state space representation of the system respectively

Note : The Ada coding is an extract from the MSDI implementation, while the Fortran coding is based on the implementation used for the CAD package developed in [49]

Example 1: Ada Implementation of Dynamic Programming

```
with MATRICE, use MATRICE,
--
procedure DYNAMIC_PROGRAMMING ( A, B, Q, R . in MATRIX, K   out MATRIX ) is
--=============================================================================
-- Computes the feedback matrix that minimises the performance index for A, B
-- and the user defined Q and R
--=============================================================================

     Pk    . MATRIX ( A'range(1), A'range(1) )   = ( others => (others => 0 0) ),
     Pk_1    MATRIX ( A'range(1), A'range(1) ),
     F       MATRIX ( B'range(2), A'range(1) ),

     J       Integer                           = 0,

     Stop_Condition     constant Long_Float     = 1 0E-16,

begin

     loop

        F     := ( INV ( R + Trans(B) * Pk * B ) ) * Trans(B) * Pk * A,
        Pk_1  := Trans(A)*Pk*A - Trans(F) * ( R + Trans(B) * Pk * B ) * F + Q,

        if Col_Norm ( Pk - Pk_1 ) < Stop_Condition then
              exit;
        else
              Pk   = Pk_1,
        end if;

     end loop;

     K := F,  -- Final controller,

end DYNAMIC_PROGRAMMING;
```

Example 2: Fortran Implementation of Dynamic Programming

```
subroutine DYNAMPROG ( A, B, Q, R , K, N, IP, SIZE )
*****************************************************************
* Module : Dynamic Programming        *
*           *
* This module uses the dynamic programming technique to solve*
* the optimal control problem.        *
*           *
*****************************************************************

     IMPLICIT DOUBLE PRECISION (A-K), DOUBLE PRECISION ( O-Z )
     INTEGER N, IP, SIZE, IER

     DIMENSION A(SIZE, SIZE), B(SIZE, SIZE), Q(SIZE, SIZE) R(SIZE,SIZE)
     DIMESNION K( SIZE, SIZE), SUM(SIZEE, SIZE), FN(SIZE, SIZE)
     DIMENSION AT(SIZE, SIZE), BT(SIZE, SIZE), PN_HOLD( SIZE, SIZE )
     DIMENSION FACTOR( SIZE, SIZE), WK( SIZE, SIZE )
     DIMENSION TEMP1( SIZE, SIZE ),TEMP2( SIZE, SIZE ), TEMP1( SIZE, SIZE )

     ' Form Transpose of A and B
     CALL MTRANS ( A, AT, N, N, SIZE )
     CALL MTRANS ( B, BT, IP, N, SIZE )

     J = 0

     ' Repeat computation of P and F until F converges

30   J = J + 1

          CALL MMUL ( BT, PN, TEMP1, IP, N, N, SIZE )          ' B'.PN
  CALL MMUL ( TEMP1, B, TEMP2, IP, N , IP, SIZE )      ' B'.PN B

          CALL MADD ( R, TEMP2, FACTOR, IP, IP, SIZE )         ' R + B' PN.B

          ' use IMSL routine Linv2f to compute Inverese of R + B'.PN B
          CALL LINV2F ( FACTOR, IP, SIZE, TEMP2, 4, WK, IER )

          ' Check Inverse Computed O K.
          IF ( IER  NE. 0 ) THEN
               PRINT *, '*** ERROR ***', IER
               RETURN
          END IF

          ' Finally compute FN = Inverse(R + B' PN.B )    B' PN A
          CALL MMUL ( TEMP2, TEMP1, TEMP3, IP, IP, N, SIZE )
          CALL MMUL ( TEMP3, A, FN, IP, N, N, SIZE )

C
C         Compute Pn-1 = A' Pn.A - Fn'.( R + B' Pk.B ).Fk + Q
C

          CALL MMUL ( AT, PN, TEMP1, N, N,N, SIZE )
          CALL MMUL ( TEMP1, A, TEMP2, N, N, N, SIZE )          ' A' Pn A

          CALL MTRANS ( FN, FNT, N, IP, SIZE )                  ' FN'

          ' Compute Fn' ( R + B' Pn.B ) Fn
          CALL MMUL ( FNT, FACTOR, TEMP1, N, IP, IP, SIZE )
          CALL MMUL ( TEMP1, FN, TEMP3, N, IP, N, SIZE )

          CALL MXCOPY ( PN, PN_TEMP, N, N, SIZE )               ' Store Pn

          '
          ' Now compute A'Pn A - Fn' ( R + B' Pn.B ).Fn + Q
          CALL MXSUB( TEMP2, TEMP3, TEMP1, N, N, SIZE )
          CALL MADD ( TEMP1, Q, PN, N, N, SIZE )
```

Example 2 Cont'd. on next page

Example 2 (Cont.): Fortran Implementation of Dynamic Programming

```
! Check to see if has converged
CALL MXSUB ( PN, PN_HOLD, TEMP1, N, N, SIZE )
CALL NORM ( TEMP1, CNORM, N, N , SIZE )

IF ( CNORM .GE. 1E-16 ) THEN
      GOTO 30
ELSE
      CALL MXCOPY ( FN, K, IP, N )
END IF

RETURN
END
```

The Ada implementation is clearly much more concise. Both packages use a Matrix facility. The Ada Matrix package, however, can use attributes (i.e. 'length which computes the size of a matrix passed in) which make them much more robust and not needing the size of the array used for the matrices to be passed in. Thus in Ada we declare the matrices to the exact required size, rather than in Fortran which we declare all matrices to be of a SIZE, and then pass this parameter to the Matrix subroutines.

The functions in Ada are also much easier to read and use. Compare Matrix addition, subtraction, multiplication and copy routines MADD, MXSUB, MMUL and MXCOPY of Fortran to Ada's use of operator overloading to give standard matrix equations, +, -, *, := . Also notice that all the intermediate storage variables disappear in Ada, leaving the compiler to do the work the programmer has to do in Fortran.

Pascal and C do a better job then Fortran in some areas. But this example would be coded similar to Fortran in them, if in a more structured fashion. This just one example of the advantages of using Ada. A quick comparison of Ada with C is shown in Table 6 in the area of array handling.

Table 6: Comparison of Ada and C array Handling Mechanisms

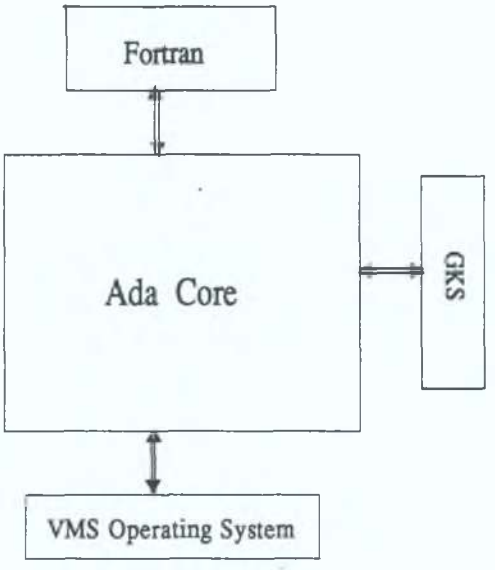
Array Operation	Ada Implementation	C Implementation
Assignment	X := Y;	For ((I = 0; Px =&X; Py = &Y), I = I+1, I < Y_Size) (Px++ = Py++);
Partial Assignment	X(5..7) := Y(3..4);	Similar loop as above
Comparison	X(I..J) < Y(1..4)	Similar to above but including explicit boolean check

The Ada language has much more powerful array manipulation than C, as illustrated above. Almost all the "tricks" used in C programming such as copying arrays using auto-incremented pointers are done in Ada using language features such as slicing (i.e. X(5..7)). It removes the need for the programmer to control these "tricks" and details this work to the compiler.

Only one CACSD package is currently implemented in Ada (IMPACT [55]). The easy interface of Ada under VMS 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 [50] was used to ensure portability between computer systems and easy maintenance. VAX GKS is an implementation of the *Graphic Kernal System*, a device independent graphic package. VAX GKS was considered the lowest level of a graphical entity. Thus terminal dependency was removed. The interfaces between the Ada Core , Fortran Libraries and GKS package are shown in Figure 6.

Figure 6: Schematic of Language Interfaces



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. These facilities are:

1. Definition of the primary data object, the symbolic expression.
2. Symbolic expression operators, such as create, select, manipulate and delete.
3. Packages which defined generic AI objects such as patterns, rules and rulebases.

EXPERT provided the facilities to do backward inferencing on a rulebase.

These two provided all the facilities that were need for the project, as AI was not considered a major component for the prototype to be constructed. The main reasons for using them was to avoid using another language, and to preserve the portability of the final package.

3.6 Design Method

The design method to be used for software development was based on the *object oriented method* developed in [51]. 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. This involved both composition and decomposition.

3.7 Development Tools

By using Ada to develop the CACE package much of the project management facilities were already provided (in APSE). Other tools were used to support other design phases. The list of tools used during the development were :

a. Language Sensitive Editor - To reduce typing overhead.

b. Ada Development Environment - To provide compiler and code management system to control programme updates (ACS provided this).

c. Programme Code Analyser - To help identify computational dominant sections of code.

d. Source Level Debugger - To help debug source code.

e. Cross-Compiler - To generate code for target system.

3.8 Computing Environment

The package currently runs under the VMS operating system, thus mainly limiting the system initially to the VAX architectures (e.g. 11/785, MicroVAX). The system is designed to be highly portable and has very little linkage to the VMS operating system or VAX architecture. This should allow it to be transported to other machines with minimal modification. To date this has not been tried.

The minimum hardware set the MSDI prototype is designed to run on with acceptable performance is :

- 0.1 MFLOPs processing power.

- 1 M primary storage.
- 10 M secondary storage
- High Resolution Bit-Mapped Graphics Display
- Graphic Input Device

The target environment used for package development was a VAXStation II system with a monochrome graphic monitor with high-speed network link to a MicroVAX II system. This environment provided graphical design entry and created a distributed processing environment.

Future workstation environments will be much more powerful with parallel processing and workstations linked to large mainframes to perform the computational intensive functions The target system used for the development was seen to be a scaled down version of such systems.

CHAPTER 4

MSDI ARCHITECTURE

The previous chapters suggest that the success of the next generation of CACE packages should be measured by how well they integrate all the relevant engineering activities in the modelling, specification, design and implementation of a control system.

This chapter describes and defines the architecture that was designed to support such a structure for MSDI First the reasons behind this architectural design are developed, and then the detailed functional requirements of the major components are given.

The final architecture design for MSDI was arrived at following decomposing the problem functionally and then by the *object-oriented-design* (OOD) method Functional decomposition and object-oriented decomposition produced different architectures The final design uses mainly the OOD architecture, using the parts of the functional architecture at a lower level.

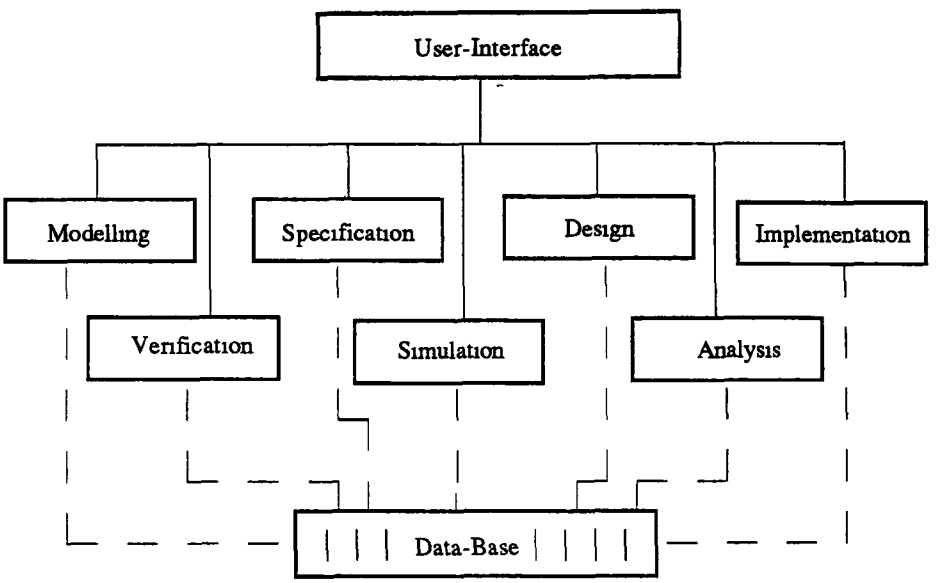
During the discussion on the architecture, two classes of control system representation of a plant model will be presented. One called the *behavioural model*, and the other called the *implementational model*. The behavioural model refers to an abstract, mostly technology-independent representation used to perform linear analysis and design Such models are the linear, time-invariant state-space and transfer function representations. The implementational model refers to a more detailed model incorporating the technology constraints used to implement a model An example would be the word-length of a microprocessor used to implement a control algorithm, or the A/D conversion time

4.1 Functional Architectural Design

The *functional architectural design* divided the CACE process into stages or functions seeking to modularize the system. This is the approach used for most CACSD packages to date The MSDI functional architecture designed, shown in Figure 7, does not differ radically from the architecture of packages such as CLADP or MATRIXx. This structure supports the main functional blocks as derived in Chapter 2 This model maps onto the CACE process model that it needs to support.

To achieve the goals identified in Chapter 2, the functional architecture was structured into eight major components user-interface, modelling, specification, design, simulation, verification, implementation and database modules. Some of the minor blocks such as analysis and documentation are not shown in Figure 7. This architecture allows individual modules to be developed separately and upgraded independently of each other The major interface between each block is the database, which itself is functionally decompoesd to allow incremental development of the system.

Figure 7: MSDI Functional Architecture



Notes
(1) Database sectioned to indicate its segmented structure
(2) Linkage between functional components is through database coordinated by the user-interface
(3) Direct access to database from user-interface not shown as this is considered a low-level feature

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 and computation of the gain matrix required to position the poles of a model at desired locations This type of architecture tends towards making data global - instead of localizing it where it is used, thus make the system more resilent to change Furthermore, a package based on this architecture is also less resilent because a change in data structure tends to ripple through the entire architecture, necessitating major code re-writing.

As part of the functional decomposition, data-types needed to support control system engineering were identified These data types are shown in Table 7 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 7

Table 7: 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

4.2 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 reponse 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 in this thesis is needed on the specification side, it was felt better to encapsulate it on its own for future modification.

4.2.1 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·

$$x_{k+1} = \Phi\, x_k + \mathrm{T}\, u_k \tag{4}$$

$$y_k = C\, x_k + D\, u_k \tag{5}$$

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 8.

Table 8: 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 substracted
Step Source	y(t) = Magnitude	
Pulse Source	$y(t) = \begin{cases} Magnitude & if\ t \leq Width\ ; \\ 0 & if\ t > Width\ ; \end{cases}$	
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	$y_k = \begin{cases} u_k & if\ Sat_{min} \leq u_k \leq Sat_{max}\ ; \\ Sat_{max} & if\ u_k > Sat_{max}\ ; \\ Sat_{min} & if\ u_k < Sat_{min}\ ; \end{cases}$	Max. and Min. Limits

Table 8 (Cont.): MSDI Atomic Components and their Evaluation Functions

Atomic Object Name	Evaluation Function	Comment
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*

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 state space models in series The EvFn for such a model is the same as in (4) and (5) with Φ, T, C and D defined as .

$$\Phi = \begin{bmatrix} \Phi_1 & 0 \\ T_2 C_1 & \Phi_2 \end{bmatrix}, \qquad T = \begin{bmatrix} T_1 \\ T_2 D_1 \end{bmatrix}, \qquad C = [\, D_2 C_1 \quad C_2 \,], \qquad D = [\, D_2 D_1 \,],$$

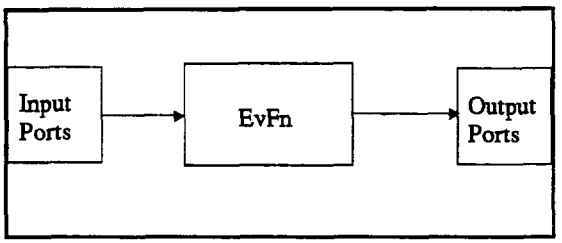
where the subscripts 1 and 2 refer to the first and second blocks Normally, a composite component will not be so simple, as various outputs of an atomic component will feed into different component blocks Some of these blocks may be non-linear, such as the saturator, preventing computation of EvFn into 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

The only composite component currently defined is

Macro-Block — a group of interconnected atomic blocks that define the composite EvFn The no of I/O are defined by the no of input and output blocks defined in the group

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 8

Figure 8: 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 3.

Example 3: 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 of
defined in equations (4) and (5)
```

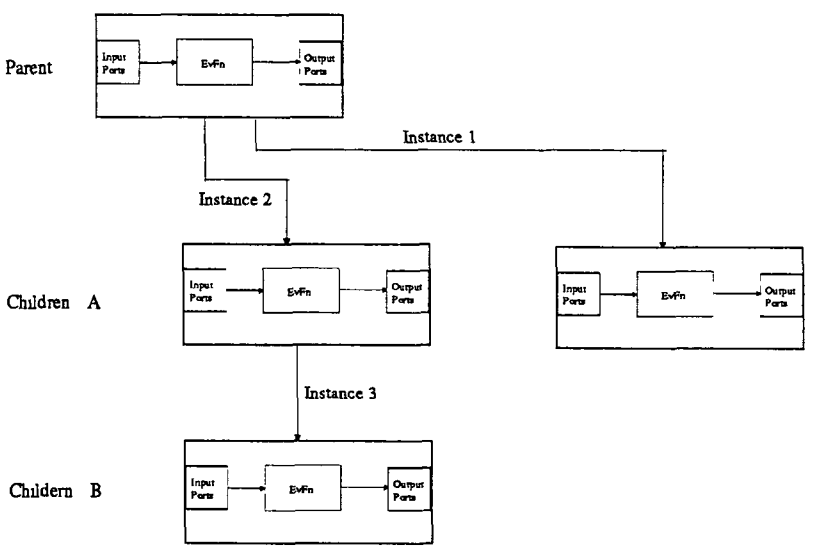
Based on these atomic components, looking at the design from a bottom-up view, several clear objects are seen to be required to support the MSDI system Most of these data types map directly onto atomic components. These objects, which from now on are considered available data-types, are shown in Table 9, plus their associated major operations They define the majority of the data types identified as needed for a control package in Table 7 Examples of the implementation of the generic matrix type, complex number and polynomial are shown in Appendix D

Table 9: MSDI Data Types Supported

Data Type	Operations	Typical Use
Complex Number	Arithmetic +, -, *, / as defined over complex field, standard functions sin, tan, cos,asin, atan,acos, extration of real/imaj _part, etc	Parameter value, part of an expression, variable value (e g in frequency response)
Generic Matrix	Matrix Arithmetic operations +, -, * , **, inverse, eigenvalue, eigenvector, standard functions	Provide instances such as real matrix or complex matrix used in state space representations and transfer function matrix for MIMO systems
State Space	Compute output, controllability,observability, pole placement, frequency reponse computation, series and parallel combination	A component of a plant or a controller
Polynomial	Polynomial arthmetic, evaluation	Define a denominator or numerator for a transfer function
Transfer Function	Conversion from State Space form, frequency response evaluation, series and parallel combination	A controller / compensator, component of plant model
Table	add to, take from, initialize, update	Storing of data (time responses, parsing table, etc)

Atomic components, as well as being used to compose a macro block, 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 9

Figure 9: Dependency Structure



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 10

Table 10: Class of Dependent Components

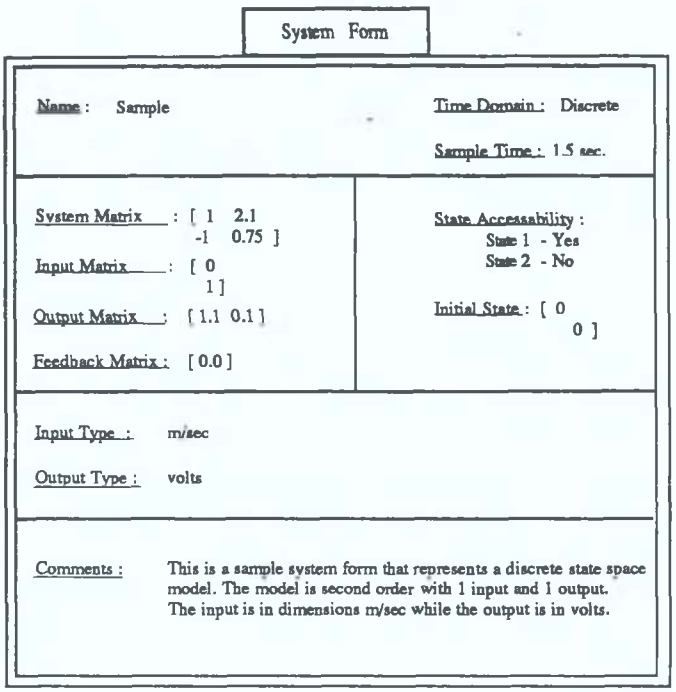
Atomic Component	Parameter(s)
State Space	Sample time, transfer function model
Transfer Function	Sample time, state space model
Controller	Linear model (i e state space, TF or macro)
Implementation	Controller, hardware (i e wordlength, multiplication speed)

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 This idea maps onto Maciejowski's data base ideas [58] The main difference is the increased flexibility as, what Maciejowski defines as a *system type* is distributed across the architecture and encapsulated in the objects that use or supply each data component For example, a continuous transfer funtion controls its discrete version. This means that changes to a data structure, such as a transfer function, are localized to one small section of the code

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. Originally it was planned to use a form to solicit and present this information as shown in Figure 10 This type of a form would allow concise and clear data entry and display Unfortunately, during the implementation phase VAX GKS V3 0 was found to contain a major bug in the implicit regeneration of Regis screens This meant that to implement this form as intended using only GKS, continual regeneration of the full screen was needed This would cause major delays if many objects are displayed on the screen and user irritation from the constant "flickering" of the screen As no form package (such as FMS or TDMS) was available to provide this data entry format, it was decided to use a prompt mode for data entry This meant that the behavioural model's various components are solicited individually, in a predefined order from the user This is not seen as a major drawback, except for experienced users, who might find this a bit inflexible A current release of VAX GKS V3 1 has this bug fixed, so it is possible to implement the original System Form idea

Figure 10: System Form



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.

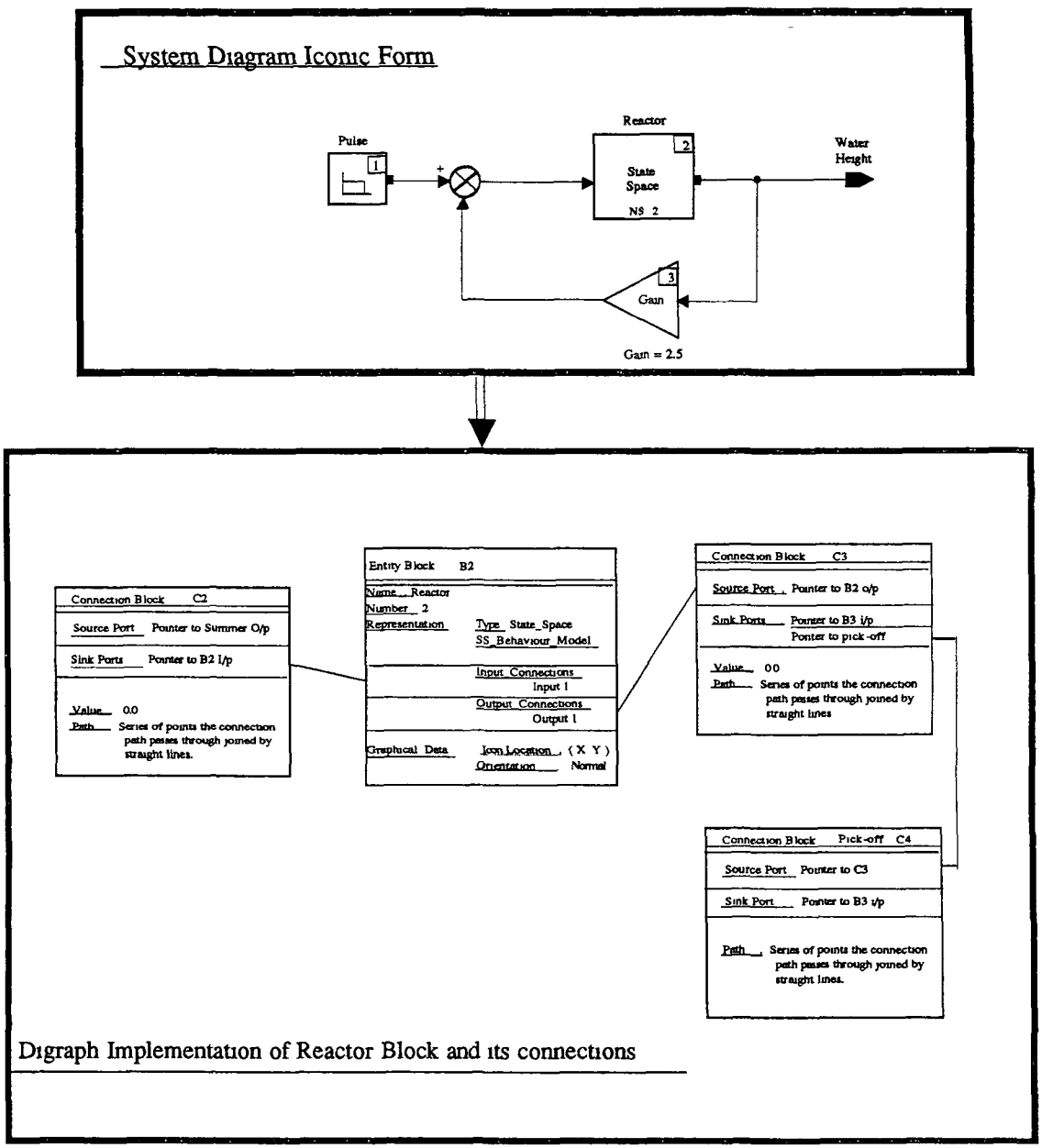
The *icon* package provides the graphical attributes facilities. It allows a user to position the graphic representation or icon of a component on the display surface and to perform various manipulations on these icons such as select, move and delete.

The *system diagram* type uses these three packages to implement its entity. The data structure used to represent the *system diagram* type is the *directed graph* (digraph). A digraph is defined as an ordered pair (**V, E**), where **V** is a set and **E** is a binary relation on **V**. **V** and **E** can be represented geometrically by a set of vertices $(v_1, v_2, ..., v_s)$ and a set of unidirectional edges $(e_1, e_2, ..., e_t)$ [61].

A typical data structure used to implement a digraph is a list of vertices and a list of edges. Each vertex is stored with an incoming edge table and an outgoing edge table. Each edge is associated with two pointers - one to its source vertex (or node) and another to its sink. This sort of doubly linked representation allows both forward and backward graph scanning and efficient searches for a vertex from an edge, or vice versa. There are many advantages in using such a structure for editing and manipulating the system diagram. The main drawback is increased complexity from simply generating a netlist. The OOD method helped decompose this complexity to manageable sections - i.e. *system form*, *connections* and *icon* packages.

Therefore the data representation of the *system diagram* consists of (a) a list of nodes (vertices) and (b) a list of unidirectional connections (edges). Each node record contains pointers to the records of its input and output connections, the type of behavioural block it is, its behavioural model, and its graphic data (i e where it is positioned on display surface, orientation, iconic representation, etc) A connection is a topological path from one component to another A path from a pick-off point in a connection is treated as a separate connection although it references the same signal Each connection record contains pointers to its preceding and succeeding blocks Figure 11 illustrates how a *system diagram* representation looks for a specific diagram. The link between each block represents really two links, one each way.

Figure 11: System Diagram Representation of a model



Note, that the digraph implementation is more complex than shown in Figure 11. Such things as the behavioural models and dependencies are not flushed out to avoid clouding its essential structure. In actuality, the various nodes of the System Diagram and the connections are multi-threaded (i.e. two way links between entities). The structure actually takes on a more n-dimensional characteristic which is not representable on two-dimensional media.

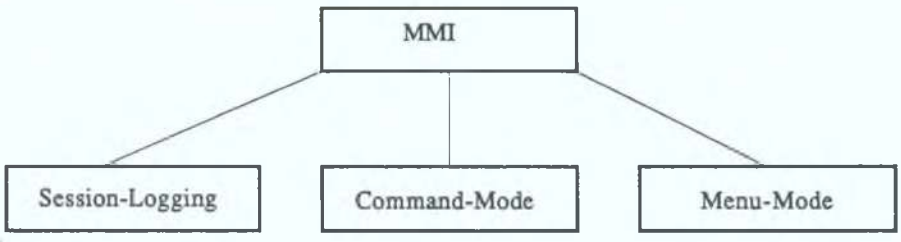
A signal leaving a block (or more precisely, a block's output port) may have more than one destination via connections, which are represented diagrammatically by a set of line segments as shown in Figure 11. Usually these line segments begin at a block's output port, fork at one or more pick-off points, and terminate at several input or sink ports. The path primitive can be described by a tree structure. The start point of the path is referred to as the root of the tree, the junction point of two connections as an internal node, and a sink port as a leaf. All these connections are represented by a so-called *p_node* types (p_node stands for partial node), which contain the co-ordinate of the point, a reference to the relative connection, a pointer to the parent or root node, and a list of children. The data structure of the path primitives allows the connections to be logically manipulated.

4.3 User-Interface Architecture

Design of a 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 system 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 the MSDI architecture is to allow 2 modes of operation. Firstly, a command-driven environment similar to MATLAB 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 basic user-interface structure, called the man-machine interface (MMI), is shown in Figure 12.

Figure 12: User-Interface Structure



The function of the *session-logging* facility is to record the activity of each session. This allows a session to be restarted, or be documented in some predefined form if required or to allow the recall of a previous command for execution/modification.

The command-mode component of the MMI defines a command-driven parsing action Commands are textually entered, parsed and acted upon if valid This is similiar to the MATLAB interface, but supporting a more complete set of control oriented data types Command primitives to construct and manipulate the System Diagram are needed

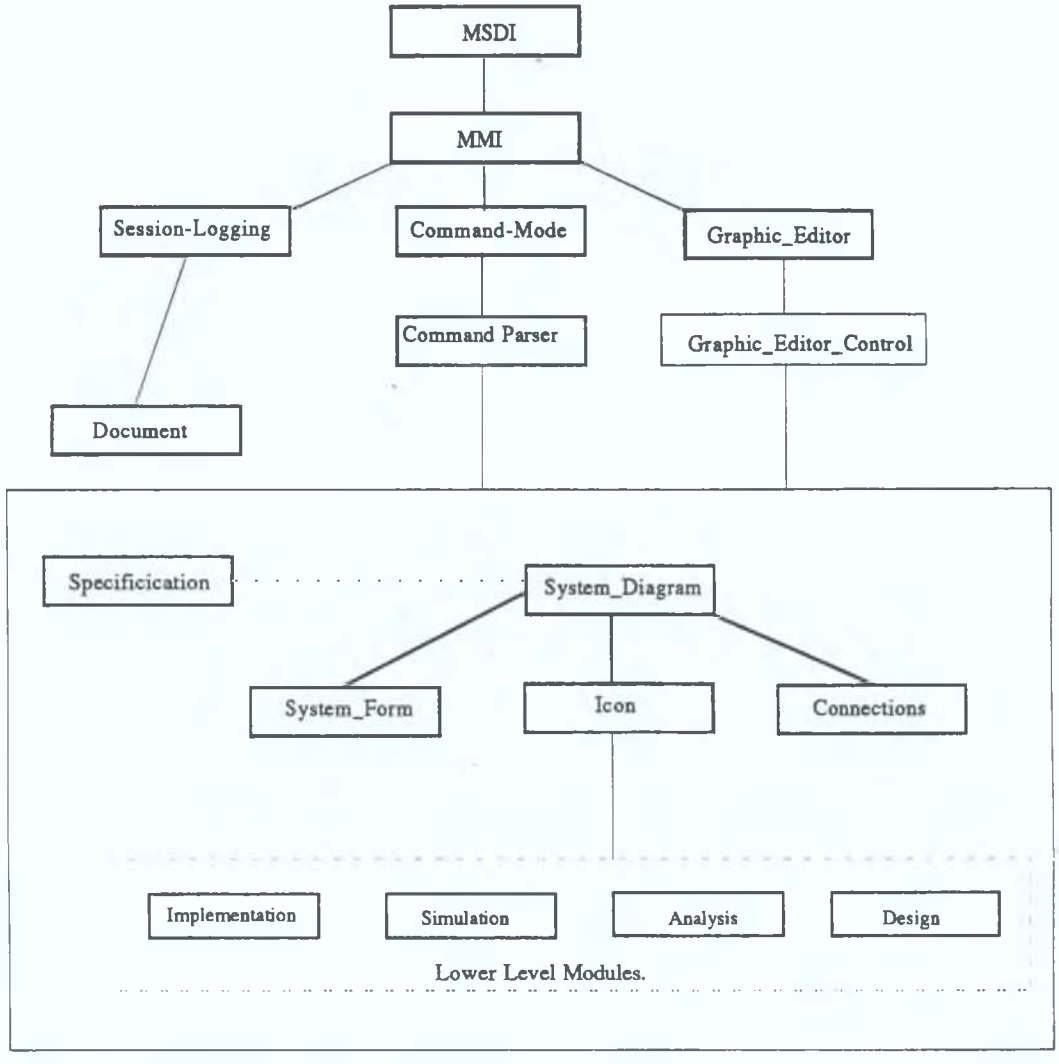
The menu-mode component has been designated as the *graphic editor* in MSDI, as it basically allows graphical manipulation of the *system diagram* The menus are organised to allow the engineer to sequence through the phases of a design, seeing the options available to him and complete operations in a minimum no of keystrokes

The functionality in both modes is the same The command mode can be seen as a very flexible interface to manipulate entities at a much lower level, while the menu-mode is highly structured and requires much less knowledge to use. The prototype package has put much greater emphasis on the menu-mode, as this is seen as the primary method of introducing a user to a new package.

Both modes of MMI share a common parser This is to allow consistency of syntax in either mode, such as matrix definition and manipulation and function calls The command mode is seen to have commands that equate to menu options, such as to manipulate the System Diagram, simulate it and delete it.

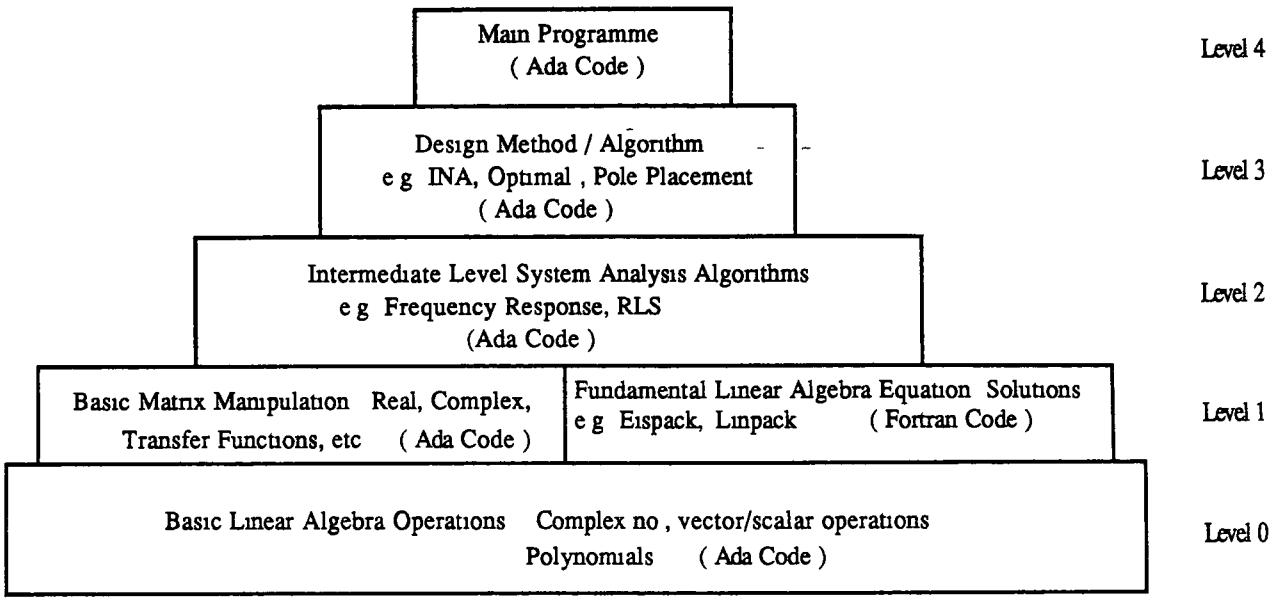
The final architecture for the MSDI package is shown in Figure 13

Figure 13: MSDI Architecture



Notice that this differs from most of the architectures used for existing CACSD packages. Typically these are decomposed functionally. The OOD method places most of these functions at a much lower level of abstraction. 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. Figure 14 show how these designs are composed from lower level parts.

Figure 14: Mathematical Software Structure



A design method at level 3, is seen to be composed of routines at the lower levels The structure of MSDI, with levels 0-4, allow the easy integration of new methods as the majority of the code has already been written, particularly for level 0,1 and 2

4.4 MSDI Functional Specification

The functionality this architecture was designed to support is categorised in the following sections by design phase (i.e modelling, design, etc) These functions are distributed across the architecture through the *system diagram* entity They are grouped by design phase to give a clear picture of the overall functionality designed-in. Also this can be seen as a specification for a complete MSDI package

4.4.1 Modelling

The MSDI Command Language (CL) supports performance (i e specification) and implementation architectural modelling as well as the more traditional behavioural modelling An important aspect of the MSDI CL is that it is viewed as a continuum between these modelling levels This view is justified since performance evaluation can be thought of as a high-level, almost pattern-independent verification of system model, whereas behavioural simulation is normally considered to be a pattern-driven exercise of the same model The MSDI CL could also address the needs for high-level analog modelling In particular, means could be provided for algorithmically describing complex analog components such as OPAMPs, ADCs and DACs. These models would cater for the implementation structures we will be using The prototype developed does not support all these models but the OOD method used allows additional models to be added. The modelling needs are addressed by .

- Schematic-capture of block diagrams of plants or systems as the primary hierarchical method to describe models

- Individual blocks can be linear, or non-linear built up using icon menu and defining the internal behaviour of each block.
- Icons include all standard models · state space, transfer function, saturation elements, tables, noise sources, summations, continuous /discrete forms, etc
- Copying, deleting, updating, combining of blocks, etc is facilitated
- Modelling of linear behavioural models and controller implementations (in terms of hardware and software used) is incorporated. Non-linear behavioural models need to be added, but are not implemented in the prototype.
- Capture of both the formal and "informal" (e.g state accessibility) models of objects and their constraints for direct use in design process
- Facilities to interface to subroutines developed external to the MSDI system to be incorporated as models
- Identification facilities to analyse experimental data
- Model transformations tools to support linearization, model reduction, frequency to state space representations and vice versa and discretization
- Database to store models

Also the command language of the user-interface will include structured access to these facilities i e menu-mode of operation

4.4.2 Specification

The specification functionality defines the criteria that a designer requires his design to meet. To provide this functionality the MSDI specification component will be based on

- A set of "primary" indicators / criteria of system performance (e.g bandwith or pole locations) from which all other criteria can be defined
- A facility to add more to this primary set in a structured manner.
- Consistency and completeness checkers of a specification
- Formulation of criteria in executable form so as to allow simulation of specifications
- Access through the CL of MSDI
- Close integration with the verification tools

4.4.3 Design

The design suite incorporates well-proven and numerically stable algorithms to give the designer

- Methodologies to support design of linear, time-invariant , MIMO systems in both frequency and time domains.
- Design methods for both controllers and observers.
- Facilities to include new methods on-line and easy integration of such methods into the package's "standard" suite at a latter date
- The constraints of all design methods built into methodologies to aid and warn designer

4.4.4 Simulation

The simulation engine provided is designed to cover the needs of the designer at both the behavioural and implementational levels. The functions of this simulation component are

- A multi-simulator approach that allows for all levels of simulation behavioural and implementational Both continuous, discrete simulation, and hybrid (continuous and discrete) are catered for
- A simulation platform capable of taking advantage of multiprocessing environments This is envisaged to be through using Ada's tasking facilities to distribute a simulation run over several processors.
- 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
- Both time simulation and frequency response
- Standard interface for other simulators and tools
- Facilities for both deterministic and statistical stimuli
- Examination and recording of simulation data 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
- In-built consistency checkers to maintain integrity of the simulation both numerically and conceptually
- Generation of warning messages when specification criteria are breached or errors detected

4.4.5 Verification

The verification tools will be developed initially to run separately from all other tools but ultimately seen to run in the background of a design session continuously verifying actions in real-time The requirements of this component are

- Formal reasoning mechanism to examine and prove that specific design criteria are attained, can or cannot be attained when sufficient information on system and design requirements are available
- Verify completeness of simulations to verify design criteria
- Check consistency and completeness of design strategy and implementation as against specifications
- Verification of validity of design decisions

4.4.6 Implementation

The facilities to be included to aid in the implementation phase of the design process are

- Catalogue of hardware and software components available for incorporation into an implementation of a design.
- Inclusion of details of their constraints and limitations

- Facilities to update component library and include new components (both hardware and software).
- Integrate cross-compiler and low-level debuggers to support development of the implementation on the target architecture
- Software blocks for control strategies, drivers for hardware platforms, numerical subroutines, jacketing software, fail-safe routines and inter-processor communication routines

4.4.7 User-Interface

The MSDI user-interface will be the basis for tool integration It must provide a consistent user interface, both textual and graphical, to all the design tools The capabilities of this interface are

- The schematics environment that allows designers to enter, display and modify the schematic-level description of their design, via a graphical interface.
- Designers able to specify their design hierarchically —able to place multiple copies of a box, the contents of which are specified only once, elsewhere
- The system able to provide and support .
  - a. Input from the designer.
  - b output from simulations, displayed in graphical and tabular forms
  - c. A structured command language that encompasses modelling, specification, design and implementation
  - d Facilities to perform basic mathematical manipulation such as matrix calculations
  - e. Controlled database examination and manipulation facilities
  - f On-line help facilities
  - g. Macros and other features to allow for extendibility of commands and functions on-line
  - h. A path between the modelling component and the simulation tools support
    - a fast interactive simulation loop and a batch oriented simulation loop
    - input to the simulation system—designers will be able to communicate to the simulation system by pointing to a symbol on their schematic, rather than by typing in a textual name
    - highlighted paths on the schematic
    - path extraction for simulation (i e extract a sub-diagram for simulation)

4.4.8 Overall Performance

The performance of the MSDI system 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. An attempt to quantify this, using the prototype package, is done in the next two chapters

Specific performance criteria for the system that can be defined are :

- Response time of the system to commands to be optimised for quick updating of the user on what is happening - within 1 sec. of command i.e. if performing an action that takes more than 1 sec. to complete then the user is informed.
- All algorithms and commands to be robust with error recovery incorporated All errors will be logged into a file to facilitate analysis
- The documentation facility includes reports generation of a session activity.
- The accuracy of routines and models as they progress through the design cycle are computed and available to the user
- Security of the system with controlled access of system facilities designed-in to overall structure
- Error-handling and reporting functions included.

CHAPTER 5

MSDI PROTOTYPE IMPLEMENTATION

This chapter outlines the details of the prototype implementation of the MSDI architecture Major design decisions are explained Algorithms used are outlined and reasons for their selection given.

The final sections of this chapter give details of the limitations of the prototype and functionality not included Most of these functions were not considered essential to demonstrate the power of an integrated CACE package Some particulars are given on how these can be included to the package quite easily

The description of the prototype starts at the user-interface and works down into the internals of the package. The main direction taken for the prototype was to support design of embedded digital controllers The mode of operation is mainly directed towards menu-mode with the package producing Ada code that can be compiled to implement the designed controller

5.1 User-Interface

The user-interface (also referred to as the man-machine interface) of the prototype provides both the modes of operations as defined necessary in the previous chapter *command-mode* and *menu-mode* They are both linked by a common parser. The next section describes the parser developed for MSDI and how it is structured. An informal introduction to parsing theory is given followed by the MSDI translation grammar. Then the details of how command-mode and menu-mode operate are given

5.1.1 Introduction to Parsing

Parsing is one of the most common functions performed by computer software systems Most people learn parsing in their primary school English classes, when they learn that a sentence of the English language is composed of at least two phrases (subject and predicate) and can possibly have more (direct and indirect objects, subordinate clauses, etc) Students of elementary English are taught to diagram sentences, that is, decompose them into phrases, each of which represents a unique syntactic unit, and that phrases themselves decompose into subphrases that decompose into subsubphrases and so forth, producing a diagram that any computer programmer would readily recognise as a tree. This tree structure is known as a syntax tree, or parse tree.

Parsing the English language is currently beyond the capability of computer software due to the language's rich syntax and the potential for writing ambiguous sentences A sentence with more than one parse is said to be ambiguous.

Languages with ambiguous sentences are not generally useful for communicating with a computer. Programs that accept input from a file (such as a compiler for a high-level language) or from a terminal (such as an information retrieval system) must have their input in an unambiguous form that conforms to well-defined parsing rules. The collection of all allowable inputs is called a language in a formal sense, and the rules for parsing are collectively known as a grammar The grammar actually consists of four distinct components a list of terminal symbols, a list of nonterminal symbols, a goal symbol and a list of productions.

The terminal symbols of a language are the words and punctuation symbols of the language For example, in English "computer" "control" and "design" are all terminal symbols. The terminal symbols can be thought of as the leaf nodes of a syntax tree.

The nonterminal symbols of a language are the names of the phrases that compose sentences of a languages In English, these are names like "sentence", "subject", "predicate", "direct object" and "subordinate clause" The nonterminal symbols are the names that label the nodes of a syntax tree that are not leaves. The set of all terminal and nonterminal symbols taken together is called the vocabulary. A sequence of symbols of the vocabulary (both terminal and nonterminal) is called a string

The goal symbol of a language is the nonterminal symbol from which all other phrases of the language are subordinate It is the label of the root node of the syntax tree In the English example above, the goal symbol is "sentence", although in reality sentences can be grouped into larger phrases called "paragraphs", which can be grouped into larger phrases called "chapters", etc People rarely diagram whole paragraphs In this formal sense, the English language consists of the set of all syntactically correct sentences

The productions of a grammar are the rules by which one string can be derived from another A production consists of two strings, a left-hand-side string and a right-hand-side string The meaning of a production is that an occurrence of the left-hand-side string in some other string A may be replaced by the right-hand-side string to produce a new string B When this is done, the production is said to be applied, and the new string B is said to be derived from the old string A in one step There must be at least one production whose left-hand-side consists only of the goal symbol Those strings that consist only of terminal symbols and that are derived from the goal symbol by repeatedly applying the productions of the grammar are known as the sentences of the language

A very special class of grammars is when in each production the left-hand-side string consists of a single nonterminal symbol Such a grammar is called a context-free grammar The class of all languages that can be described by context-free grammars are called context-free languages These languages are particularly easy for computers to parse, because the time it takes to parse a context-free language is proportional to the cube of the length of the sentence in the worst case By applying further restrictions to the grammar, it is possible to guarantee that the language can be parsed in time linearly proportional to the length of the sentence One such set of restrictions, which will be discussed in much greater detail in the next section is called the LL(1) property

Most high-level programming languages are described using a context-free grammar A grammar that is context-free is considered a good source of documentation for a computer language, a grammar that is not context-free is usually considered poor documentation

The purpose of a parser (as a component of a computer program) can now be explained it takes as input a sentence of a language, and outputs a derivation of that sentence from the goal symbol of the language. In the case of many grammar-driven parsers, each production is known by the order in which it appears in the grammar, and is so assigned a unique number In this case, the output of the parser is the sequence of numbers of the productions in the order that they are applied in deriving the input sentence from the goal symbol The bare minimum output from a parser is simply an indication of whether or not an input sentence is in the language being parsed; in this case, the parser is simply a recognizer

5.1.2 LL(1) Parsing

LL(1) is a special class of context-free grammars that process input sentences linearly proportional to the length of the input string. The name "LL(1)" signifies that a parser for the grammar can operate by ·

- performing a single left-to-right scan of the input sentence;
- always substituting for the leftmost nonterminal symbol while deriving the input sentence from the goal symbol; and
- only scanning ahead one symbol to decide which production to use when substituting for the leftmost nonterminal symbol

This parsing method is likewise called "LL(1) parsing", and the subset of the context-free languages that can have LL(1) grammars is known as the "LL(1) languages"

In a context-free derivation, every nonterminal symbol is capable of producing a unique set of strings of terminal symbols independently of where in the derivation the nonterminal symbol appears Some nonterminals are capable of producing the special string that contains no symbols at all, known as the "empty string" A production that substitutes the empty string for a nonterminal symbol is known as an "empty production".

The following is the context-free grammar for a language whose sentences are expressions The special symbol "–>" is used to separate the left-hand-side of the production from the right-hand-side A right-hand-side that consists only of the symbol "empty" is used to indicate an empty production. The symbol "$" designates an end-of-input marker This grammar satisfies the restrictions for LL(1) grammars, which are described below

1. G –> E $
2. E –> T E'
3. E' –> "+" T E'
4. E' –> empty
5. T –> P T'
6. T' –> "*" P T'
7. T' –> empty
8. P –> id
9. P –> "(" E ")"

Note : id represents a class of identifiers.

This list of nine productions fully defines the grammar for the language of expressions. The terminal symbols are exactly the symbols that appear only on the right-hand-sides of productions, that is,

$, "+", "*", id, "(" and ")".

The nonterminal symbols are exactly those symbols that appear on the left-hand-sides of productions, that is,

G, E, E', T, T' and P.

These can be thought of as standing for goal, expression, expression-tail, term, term-tail and primary, respectively. The goal symbol must be G because it appears only on a left-hand-side and not on any right-hand-side.

Fundamental to the notion of LL(1) is the concept of head symbols. Each nonterminal symbol has associated with it a set of phrases that it can produce, and a set of terminal symbols that can follow it in some sentence. For a given nonterminal symbol s, its head symbols are exactly those terminal symbols that can start a nonempty string derived from s, plus those terminal symbols that can immediately follow an empty string derived from s.

In the example above,the head symbols are:

- for P, id and "(";
- for T, the head symbols of P, that is, id and "(";
- for E, the head symbols of T, again, id and "(";
- for G, the head symbols of E, id and "(";
- for E', "+", ")" and $;
- for T', "*", "+", ")" and $.

The head symbol of E' is "+" when production 3 is applied, and when production 4 is applied, E' produces an empty string. Production 1 shows that when this happens, it can be followed by $, the end-of-input marker. Likewise, when production 9 is applied and E' produces an empty string, it must be followed by a ")". The head symbol of T' is a "*" when production 6 applies, and when production 7 applies and produces an empty string, it must be followed by an E'.

The properties that make a grammar LL(1) are :

i Every production for a given nonterminal symbol must have a distinct set of terminal head symbols.

ii A production's right-hand-side may be empty. The head symbols of empty productions are determined by what may follow the empty phrase.

iii No left-recursive productions are allowed; that is, if s is the left-hand-side symbol of a production, the right-hand-side string may not begin with s or with a nonterminal from which can be derived some string that begins with s.

It can be seen that the sample grammar meets these restrictions. None of the productions are left-recursive. For each nonterminal symbol, it can be shown that each production having that symbol on the left-hand-side has a distinct set of head symbols. For example, there are two productions with E' on the left-hand-side, productions 3 and 4. "+" is the head

symbol of production 3 and when production 4 is applied and E' produces an empty string, it must be followed either by a ")" or by the end-of-file marker ($).

5.1.3 LR(1) Parsing

LR(1) parsing, and its derivatives SLR(1) and LALR(1), is a very different technique for recognising context-free languages Basically, input symbols are pushed onto the parser's stack (shifted) until it is determined that a production applies, at which time the top elements of the parse stack are collapsed down to a nonterminal (reduced). Thus, the entire right-hand-side string is seen before recognising the nonterminal that it is derived from The goal symbol is derived if the parse succeeds Since the terminal nodes are recognised in the parse tree before the nonterminal nodes, this method is called bottom-up parsing The name "LR(1)" signifies that the parser operates by :

- performing a single left-to-right scan of the input sentence;
- deriving the input stream from the goal symbol by substituting at each step for the rightmost nonterminal symbol (the steps occur in reverse since, in a bottom-up parse, the goal symbol is actually derived from the input stream);
- only scanning ahead one symbol to decide whether to shift or to reduce

5.1.4 Parsing Method Selection

Other parsing schemes are available such as precedence parsers but the design choice for the grammar parser for MSDI was between LL(1) and LR(1) because of their power and efficiency LL(1) was chosen over LR(1) because of its direct implementation approach and the need for large tables for LR(1) The limitatation of no left-recursive productions for LL(1) was not a major problem in the development of the MSDI grammar.

In summary, LL(1) parsing was seen as a simple, elegant way of parsing a context-free language Bearing in mind the LL(1) restrictions, it is easy to write a grammar that immediately conforms to those restrictions. LL(1) parsing is called predictive (or top-down) because a production applies when its head symbol is being scanned, therefore the nonterminals in the parse tree are recognised before we actually know which subphrases lie under them.

Up to now, the parsing action discussed has merely to recognise a valid input sentence The goal of the parsing is actually to translate valid input sentences into their defined actions This is accomplished easily by extending the context-free grammar into a *translational grammar* (TG). The translation grammar has added constructs to perform actions defined in the sentence As a leftmost derivation of the input string is constructed, actions are executed as directed by calls to various *action routines* which are defined as part of the language. These action routines make use of a parser *symbol table* which contains definitions of current symbols or variables The structure of the MSDI parser symbol table is shown in Table 11.

Table 11: MSDI Symbol Table Components

Record	Function
Name	Access Key to table for addition, deletion, etc. of data
Type	Data Type - Real, Complex, Matrix, String, Polynomial, etc.
Value	Actual value of parameter - depends on type

This symbol table was implemented as a hash table. The maximum no. of entries allowed in the prototype implementation was 250. This is easily extended.

5.2 MSDI Translational Grammar

To implement the LL(1) parser, a *grammar analyser* and *skeleton translator* (called GAST) was written. It takes as input a grammar file written as shown in Example 4 and produces two files : an internal format for the grammar (which is produced by the grammar analyser) and a procedure *perform*, which structures the calls into action routines. This latter file is in Ada, and is embedded into the skeleton translator (ST) along with the action routine definitions. The skeleton translator is a template which needs production rules and actions added. Once the ST is fleshed out with these grammar specifics it is just compiled. This allowed flexible development of the MSDI Grammar.

Example 4: Grammar Analyser Input File

```
Express       -> Term Expresses

Expresses     -> ASOP Term Expresses        ! addition/subtraction operations
              -> "null"

Term          -> Factor Terms

Terms         -> MDOP Factor Terms          ! multiply/divide operations
              -> "null"

Factor        -> "(" Express ")"
              ->  Operand

ASOP          -> "+"
              -> "-"

MDOP          -> "*"
              -> "/"

Operand       -> Identifier
              -> Number
```

The grammar analyser (GA) reads in the grammar from the file line by line. It then checks that each line is a valid production. Each left-hand-side (LHS) must be separated from the right-hand-side (RHS) by "->". Successive productions with the same LHS may follow each other without the LHS being specified. The previous LHS is used as a default.

An algorithm by Lewis, et al, reproduced in [63], is used to check that the grammar is LL(1) and compute the head symbols for each production. Terminals are divided into two categories : literal and group. A literal stands for itself in a production, such as "+", "-" or "=". A group stands for a set of actual tokens such as *Identifier* which represents the whole set of identifiers in productions.

A scanner, called Get_Token, is one of the predefined action routines. It has been designed as a generic template that is instanced with the grammar delimiators and the *keyword* group Keywords are a group of reserved words that denote specific actions Table 12 lists the keywords defined for the prototype plus their functions The other symbols are also included in this table.

Table 12: MSDI Prototype Mathematical Symbols and Functions

Symbol or Keyword	Function
+, -, /, * , (,)	Standard mathematical operations for real, complex, polynomial, transfer functions and matrix types
cos, sin, tan	Standard mathematical functions for real, complex, polynomial, transfer functions and matrix types
', diag, ident	Matrix Transpose, construction of diagonal matrix and identity matrix
inv	Matrix inversion
eigen	Computation of eigenvalues and vectors of a matrix
root	Compute roots of a polynomial
eval	Evaluate a polynomial for a specific value
pole	Compute poles of a transfer function
zero	Compute zeros of a transfer function
svd	Compute singular value decomposition of a matrix

Calling Get_Token returns the next token in the parse sequence The scanner itself reads in the the input character by character until a delimiator is reached (such as a space, tab, or a delimiator token such as +, - or *) It then checks if this input just read in is a valid token This is basically a preliminary parse driven by the grammar shown in Example 5 During this scan of the token, it determines whether a token is a literal or a group i e its class If it is a hteral then the token's class is set to "empty". Otherwise the class is set to the name of the group i e Identifier, Numeric or Keyword In either case, the actual characters that compose the token in the input are assigned to the token's value When the end of the input sentence is reached the class of the token is set to EOS (end-of-sentence) This signals to the parser that the input sentence has been fully scanned Thus a Token is composed of (a) a class and (b) a value

The scanner also provides facilities to indicate where an error in an input has been detected (i.e the point at which the parse of a legal input aborted). This separation of the main parsing and the scanning of a sentence into legal tokens is illustrated in Figure 16 The main reason for doing this was to ease the development of the main grammar It could also have been incorporated into the main MSDI grammar.

Example 5: Scanner Grammar plus Head Symbols

Productions			Head Symbols
OPERAND	->	IDENTIFIER	[A..z]
	->	NUMBER_TYPE	[0..9]
	->	SPECIAL_SYMBOL	[S_S]
IDENTIFIER	->	LETTER LIST	[A z]
NUMBER_TYPE	->	NUMBER NUMBERS FRACTION_PART	[0 9]
	->	. NUMBER NATURAL_NUM EXPONENT	[.]
SPECIAL_SYMBOL	->	S_S	[S_S]
LIST	->	LETTER LIST	[A z]
	->	NUMBER	[0 9]
	->	"null"	[Delim]
NUMBERS	->	NUMBER NUMBERS	[0..9]
	->	"null"	[,E,e,Delim]
FRACTION_PART	->	NUMBER NATURAL_NUM EXPONENT	[.]
	->	EXPONENT	[E,e,Delim]
EXPONENT	->	"E" SIGN_NUMBER	[E,e]
	->	"null"	[Delim]
SIGN_NUMBER	->	+/- NUMBER END_NUMBER	[+,-]
	->	NUMBER END_NUMBER	[0 9]
NATURAL_NUM	->	NUMBER NATURAL_NUM	[0 9]
	->	"null"	[E,e,Delim]
END_NUMBER	->	NUMBER END_NUMBER	[0 9]
	->	"null"	[Delim]

NOTE

S_S	=	Special Symbols i e (+, -, *, / , (,), =, [,], ")
NUMBER	=	0, 1, 2, 3 ,4 5, 6, 7, 8 ,9
LETTER	=	A, B . Z, a, b z
Delim	=	Deliminators as defined previously
% %	=	Action Routine

5.2.1 LL(1) Parsing Action

The main LL(1) parser is an implementation of a simple machine The machine contains a push-down store and a stream of input symbols. In its initial state, the stack contains the goal symbol The significance of this is that the parsing machine is predicting that it will see an entire sentence The stream contains a sequence of terminal symbols that hopefully form a sentence Initially, the machine is looking at the first terminal symbol at the beginning of the stream.

The LL(1) parsing machine makes simple, discrete moves. In each move, it only looks at the symbol on top of the stack and the symbol currently at the head of the stream. The top-of-stack symbol represents the phrase that the parser expects to see next in the input stream The action taken depends on whether this symbol is a terminal symbol or a nonterminal symbol :

CASE 1

Top-of-stack symbol is a terminal. In this case, the top-of-stack symbol must match the current input symbol. If so, pop the top symbol off of the stack and advance the input one symbol; otherwise, signal a syntax error.

CASE 2

Top-of-stack symbol is a nonterminal. In this case, the current input symbol must be a head symbol of a production of the stacktop symbol. If so, pop the top symbol off of the stack and push each symbol of the right-hand-side string onto the stack in reverse order; otherwise, signal a syntax error.

If after performing either of these actions the stack becomes empty and the entire input stream has been read, the input stream is known to contain a syntactically correct sentence

The following example illustrates the mechanism of the LL(1) parser In this example, the expression grammar will be used and the parser will recognise the sentence "id * id + id $"

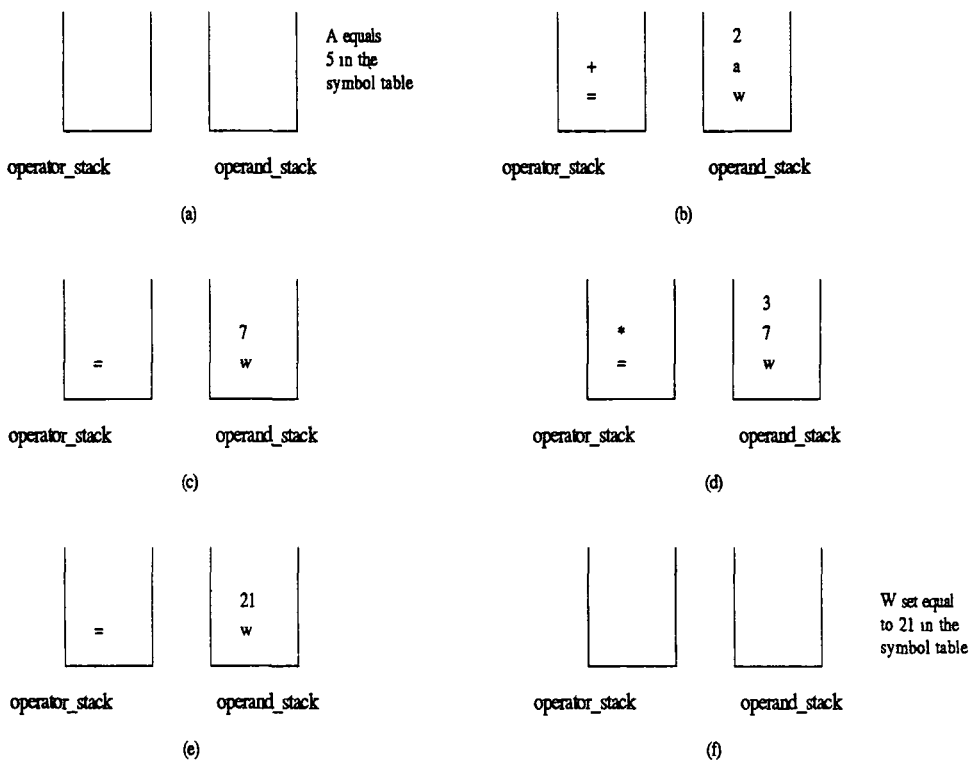
Table 13: LL(1) Parsing Action

Step	Stack	Input
0	G	id * id + id $
1	$ E	id * id + id $
2	$ E' T	id * id + id $
3	$ E' T' P	id * id + id $
4	$ E' T'	* id + id $
5	$ E' T' P	id + id $
6	$ E' T'	+ id $
7	$ E'	+ id $
8	$ E' T	id $
9	$ E' T' P	id $
10	$ E' T'	$
11	$ E'	$
12	$	$

At this point, the input is accepted as being a valid sentence of the expression language The LL(1) parsing machine as implemented above has the following optimisation· if a production applies whose right-hand-side string begins with a terminal symbol, that terminal symbol must match the current input symbol, so instead of pushing the terminal symbol onto the stack and popping it off in the next step, the input is simply advanced one symbol In the example, that optimisation eliminates five steps.

Complex expressions cannot be directly evaluated. They need to be broken up into smaller subexpressions which are computed in an order defined by operator precedence. The proper combination of the results of evaluating these subexpressions yields the value of the original expression. The algorithm used in the MSDI translational grammar for subexpression evaluation requires two auxiliary stacks, called *operator_stack* and *operand_stack* The first stack holds operators, the second holds operands of these operators, or more correctly it holds their names (or pointers). An example is used to explain the operation of these two stacks Assume the input to be parsed is "w=(A+2)*3" where A has been previously defined as 5. Figure 15 shows the changes to these two stacks during the calls to the action routines.

Figure 15: Example of Changes to Stacks during Parsing



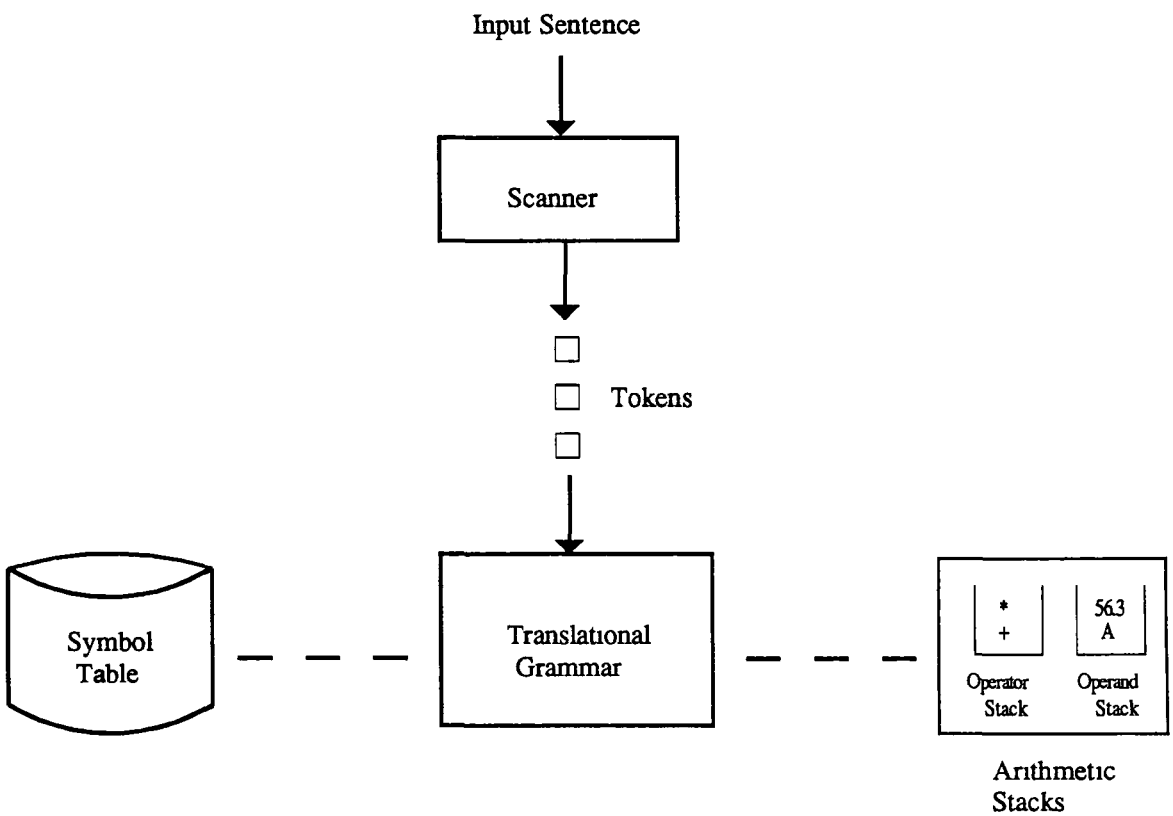
Initially both stacks are empty Then the five symbols are pushed onto the stacks { w, a, 2 } onto operand_stack and { =, + } onto operator_stack Note the brackets are not pushed onto the stack but are parsed to set up a computation of subexpression When ")" is reached the subexpression is evaluated and result stored top-of-operand_stack (i e 7) Then the next operator and operand are added to stack and the operation performed The mathematical precedence of operations for MSDI is contained in the grammar and follows standard mathematical rules

Table 14: Operator Precedence

Operators	Precedence
^, '	Highest Precedence Operators
*, /	Multiplying Operators
+, -	Unary Adding Operators
+, -	Binary Adding Operators

The complete translational grammar used in the prototype is shown in Appendix E Figure 16 illustrates the structure of the translational grammar component of the MSDI user-interface

Figure 16: Tokenizing Input



The prototype has a natural matrix environment. A matrix is entered similar to the way it is written on paper. It starts with a "[" and ends with a "]" The enclosed rows and columns define the matrix Similar to MATLAB rows can be separated by "," or started on new lines. For example, the following are both valid ways to enter a 3x3 matrix using the prototype

a. A = [1 2 3; 4 5 6; 7 8 9]

b.

```
A  =  [  1  2  3
         4  5  6
         7  8  9  ]
```

Polynomials and transfer functions are entered in a similar manner Rather than use the approach implemented in MATLAB and its derivatives, a syntax closer to the written form is used. MATLAB's syntax requires that coefficients are entered in arrays ordered from the highest power to the lowest power, including padding zeros where no coefficient exists in the transfer function 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}
```

or

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

Note { } denote a polynomial expression.

This is much closer to how transfer functions are expressed in written form The increased readability of this form is expected to overcome the increased number of keystrokes needed to enter it. This is particularly true for infrequent users of the package The prototype removes the need for compressing MIMO transfer functions into a two-dimensional form - i e with a common numerator for all transfer functions in the matrix Individual elements of the transfer function matrix can have different numerators. Internally, when required, the package computes a common numerator for the transfer function matrix.

The transfer function can also be entered in factored or zero-pole-gain for such as

```
G(s) = k * ( {s+1 }*{s+2} ) / ( {s+3}*{s^2+3*s-4} )
```

The approach taken internally is to multiply this expression out when it is to be used in algorithms such as conversion to a state space form A more sophicated approach is needed to preserve the entered form to reduce round-off errors being introduced Transfer function representations have been proven to be more sensitive to the coefficients of the terms than state space models, i.e. sensitive in terms of attributes like pole and zero locations to small changes in coefficients.

Matrix, polynomial and transfer function 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

In summary, the mathematical environment allows matrix, polynomial and transfer function operations to be written directly They are expressed as close as possible to the way they would be on paper Dimensioning of the variables and their conversion between different representations is automatically accomplished by the software

5.2.2 Prototype Parser Performance

To check the run time performance of the parser a test case was run It consisted of matrix assignment of a 10th order matrix The measurement of the time used in each procedure was recorded using the VAX Performance Code Analyser (PCA) for 100 test runs The performance data collected is shown in Figure 17

This test run indicated that the parser spent 44 % of its time tokemzing the input in the scanner Further analysis revealed that the most of this 44 % spent in procedure Token_ String was used calling the routine rather than in executing it To improve the scanner run-time performance all the scanner procedures were inlmed as well as the action routines of the parser. Inlining means that the compiler is directed to expand a routine where it is called rather than calling the routine and passing parameters The test was re-run The performance data for this new run with the scanner routines and parser action routines all inlined is shown in Figure 18

The time taken to complete the parse decreased by 32% The trade-off is the increased storage space needed for the executable image of the test programme (which consists of a loop that iterates through the parse 100 times) The inlined test program executable file was 10 % bigger than previously As storage space is not a major issue on a VAX and the parser component only represents about 8-9% of the final executable image for the prototype, the inline method was used This increased the size of the final prototype from 1849 blocks to 1869 blocks A block equals 512K.

Figure 17: Initial Parser Performance in parsing "A = [a 10th order matrix]"

```
                    VAX Performance and Coverage Analyser

            Parsing of A = [10th order matrix] 100 times - Nolining

                 CPU Sampling Data (13895 data points total) - "*"

 Bucket Name           +----+----+----+----+----+----+----+----+----+----+
 TG\TG\SCAN\            |
  TOKEN_STRING__103    |************************************************** 44.9%
 TG\TG\TG_GRAMMAR\      |
  LIST_OF_ELEMENTS .   |*****************                                   15.2%
 TG\TG\TG_GRAMMAR\      |
  UNSIGNED_FACTOR      |************                                        11 2%
 TG\TG\SCAN\            |
  GET_TOKEN__102       |****                                                3.2%
 TG\TG\SCAN\            |
  TOKEN_CLASS          |**                                                  2 1%
 TG\TG\TG_GRAMMAR\      |
  MATRIX_DEF .    .    |**                                                  2.1%
 TG\TG\SCAN\GET_TOKEN__102\
  READ_DELIMINATORS    |**                                                  2 1%
 TG\TG\SCAN\GET_TOKEN__102\SCANNER_GRAMMAR\
  OPERAND   .          |**                                                  2.1%
 TG\                    |
  TG                   |*                                                   1 1%
 TG\TG\SCAN\GET_TOKEN__102\SCANNER_GRAMMAR\
  EXPONENT      .      |*                                                   1.1%
  IS_DELIMINATOR       |*                                                   1 1%
  NUMBERS              |*                                                   1 1%
 TG\TG\TG_GRAMMAR\      |
  ASOP    . .    .     |*                                                   1 1%
  FACTOR    .    .     |*                                                   1 1%
                       +----+----+----+----+----+----+----+----+----+----+

PLOT Command Summary Information
Number of buckets tallied.                                  14

CPU Sampling Data - "*"

Data count in largest defined bucket                      6238    44 9%
Data count in all defined buckets                        12436    89 5%
Data count not in defined buckets                         1459    10 5%
Portion of above count in P0 space.                       1459    10 5%
Number of PC values in P1 space                              0     0 0%
Number of PC values in system space                          0     0 0%
Data points failing /STACK_DEPTH or /MAIN_IMAGE              1     1 1%

Total number of data values collected.                   13895    100 0%

Command qualifiers and parameters used
  Qualifiers
    /CPU_SAMPLING /DESCENDING /NOMINIMUM /NOMAXIMUM
    /NOCUMULATIVE /NOSOURCE /NOZEROS /NOSCALE /NOCREATOR_PC
    /NOPATHNAME /NOCHAIN_NAME /WRAP /NOPARENT_TASK
    /FILL=("*","O","x","@",":","#","/","+")
    /NOSTACK_DEPTH /MAIN_IMAGE
  Node specifications·
    PROGRAM_ADDRESS BY ROUTINE

No filters are defined
```

This profiling of the parser code in Figure 17 and Figure 18 is include not only to demonstrate the effect of inlining but also to show how all the other parts of the prototype were tested for various design options and algorithm performance. As the prototype is developed this type of testing will be become increasingly important to tune the package.

Figure 18: Parser Performance in parsing "A = [a 10th order matrix]" with scanner routines inlined.

```
                VAX Performance and Coverage Analyser

        Parsing of A = [10th order matrix] 100 times - Inlined Routines

          CPU Sampling Data (9401 data points total) - "*"

Bucket Name          +----+----+----+----+----+----+----+----+----+----+
TG\TG\               |
 APPEND_COL . . . . .|************************************************ 11.9%
TG\TG\TG_GRAMMAR\    |
 LIST_OF_ELEMENTS    |*********************************************     10 9%
ADA$ELAB_MSDI\       |
 ADA$ELAB_MSDI       |************************                           5.9%
TG\TG\SCAN\          |
 TOKEN_STRING__103   |*********************                              5.0%
TG\TG\TG_GRAMMAR\    |
 NEXT_ROW .        . |*********************                              5 0%
 UNSIGNED_FACTOR     |*********************                              5.0%
MMI\MMI\             |
 PROCESS_COMMAND__11 |*************                                      3.0%
TG\TG\TG_GRAMMAR\    |
 FACTOR        . . . |*************                                      3.0%
 MATRIX_DEF    .   . |*************                                      3.0%
MENU_PARAMETERS_\    |
 MENU_PARAMETERS_    |********                                           2 0%
TG\TG\SCAN\          |
 GET_TOKEN__102      |********                                           2.0%
TG\TG\SCAN\GET_TOKEN__102\SCANNER_GRAMMAR\
 IDENTIFIER .        |********                                           2 0%
 OPERAND       . .   |********                                           2 0%
TG\                  |
 TG .   .          . |****                                               1 0%
TG\TG\               |
 PARSE__11           |****                                               1 0%
TG\TG\SCAN\GET_TOKEN__102\
 SCANNER_GRAMMAR   . |****                                               1 0%
TG\TG\SCAN\GET_TOKEN__102\SCANNER_GRAMMAR\
 DELIM_TOKEN         |****                                               1 0%
 IS_DELIMINATOR      |****                                               1 0%
 KEYWORD             |****                                               1 0%
 LIST      .   .     |****                                               1.0%
TG\TG\TG_GRAMMAR\    |
 EXPRESSES       .   |****                                               1.0%
 FUNCTION_OP       . |****                                               1 0%
TG_\                 |
 TG_     .       .   |****                                               1 0%
                     |
                     +----+----+----+----+----+----+----+----+----+----+
```

5.2.3 Command-Mode

The command-mode operation of the MSDI prototype is not as well defined as the menu-mode It basically operates similar to MATLAB The anomalies in the MATLAB syntax have been removed (i e. using the symbol ' to transpose of a matrix, operator overloading, etc) to give a language closer to the way control engineers actually write.

The list of keywords available is given in Table 15. These represent a limited capability No primitive commands to manipulate the system diagram are included. The reason for these limitations is that menu-mode was seen as the primary interface mode for the prototype.

Table 15: Command Mode Keywords

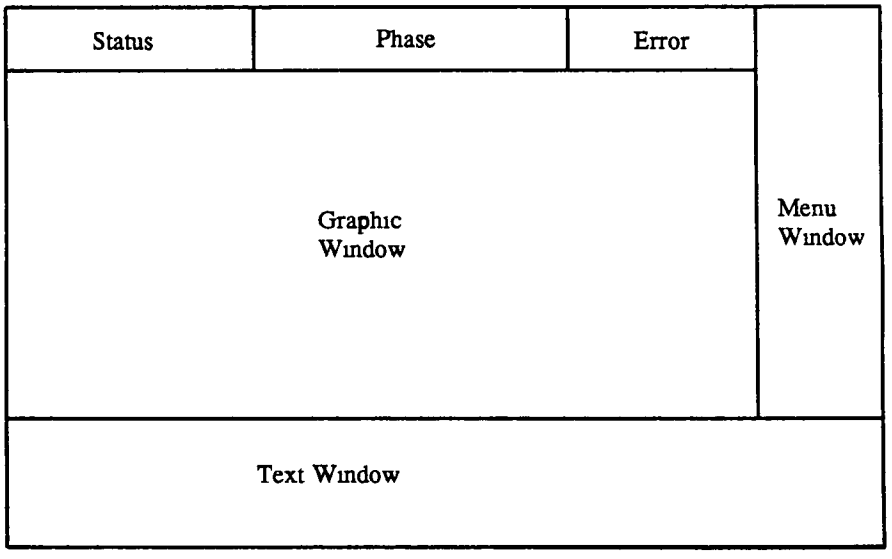
Command Name	Syntax	Description
Discretize	(I, T) = Discretize (A,B, Period)	Discretize a matrix pair using a sample time period
Simulate	simulate(I, T, C, D, Length)	simulates a discrete model described by the 4-tuple through Length sample periods
Freq_Response	freq_response(A, B, C, D, Start_ Freq, Stop_Freq, No_Points)	Computes frequency response and draws it in Bode form
Pole_Place	K = pole_place (A, B, Poles)	Computes the state feedback matrix to locate the closed-loop poles as defined by the vector of complex poles
Show	Show A or Show cos(A+2)	Displays the value of a variable or expression
Ged	ged Ctanks	Starts up menu-mode using system diagram named Ctanks The default SD is empty
Exit	exit	Terminates command-menu either exiting package or returning to menu-mode

A major shortcoming seen in using this type of mode was that the keywords could not be abbreviated. Long descriptive keywords were initially selected to ensure clarity. Either short keywords or allowing abbreviations is needed

5.2.4 Menu-Mode

The menu-mode is driven by what is called the *graphic editor* This editor allows access to the system diagram and supports its manipulation The graphic editor consists of 4 distinct parts - *menu window, graphics window, status window* and *text window*. The screen layout of these areas/windows is shown in Figure 19

Figure 19: Graphic Editor Windows



Menu-mode makes extensive use of VAX GKS. The VAX GKS library of routines provides control functions, output functions, input functions and segment functions that the prototype uses GKS allows low-level graphic primitives to be drawn such as lines, markers and fill areas. The inbuilt co-ordinate systems means the application can work in *world co-ordinates* (defined by the application programme) while GKS handles translating these into device co-ordinates for displaying objects on the screen Transformations are available in GKS to affect the composition of the graphical picture For a full description of VAX GKS refer to [50].

Each window is implemented in VAX GKS as a separate view and transformation. This allows easy reference among the windows Each window is defined and controlled by separate packages - for example the graphic window is controlled by the Icon package and the Graph package, while the menu window is controlled by the Graphic_Editor_Control package These packages, along with the generic window package are the only packages that refer to VAX GKS. This isolates VAX GKS's interface to the prototype to just four packages A different graphic driver could potentially be used to port the prototype, to say a PC, with only these four packages needing to be re-tested.

The menu window is the section of the terminal display that is reserved for the various menus The main menu active at a particular time will be displayed here Each menu option is displayed in a segment This was the method chosen over using a GKS choice logical device because it gave increased flexibility. Individual options can be made detectable as required

The graphic window is where various system diagram entities (i e blocks and their interconnections) and other graphs (eg. simulation) will be drawn The system diagram was implemented as defined in the previous chapter using the *system form*, *icon* and *connections* packages

The text window is where alphanumeric interaction using the keyboard is displayed It consists of three lines of text To provide this function consistently across various terminals, a generic windowing package was written. This package uses single character data entry to solicit input from the user Single character entry was used to allow special keys such as Control-W to be used. The special keys defined and their functions are shown in Table 16 This windowing package was latter enlarged to provide a common windowing environment across terminals such as VT240, VT340s which do not have any windowing facility and VS200/GPXs which have VWS windows The prototype implementation only uses this windowing package and none of a terminal's inbuilt functionality if it has any This generic package is used to provide pop-up windows for text I/O and graphic display in the selected area of the terminal surface

Table 16: Special Keys and their functions

Key(s)	Function
[CTRL] W[2]	Refresh the entire screen
[CTRL] P[2]	Spawn to command mode in a pop-up window
[CTRL] U[2]	Exit or abort for a pop-up window or action
PF2	Display Help
PF3[1]	Decrease cursor step size
PF4[1]	Increase cursor step size
PF17	Copy screen to a file

[1] Only defined on VT240s and VT340s

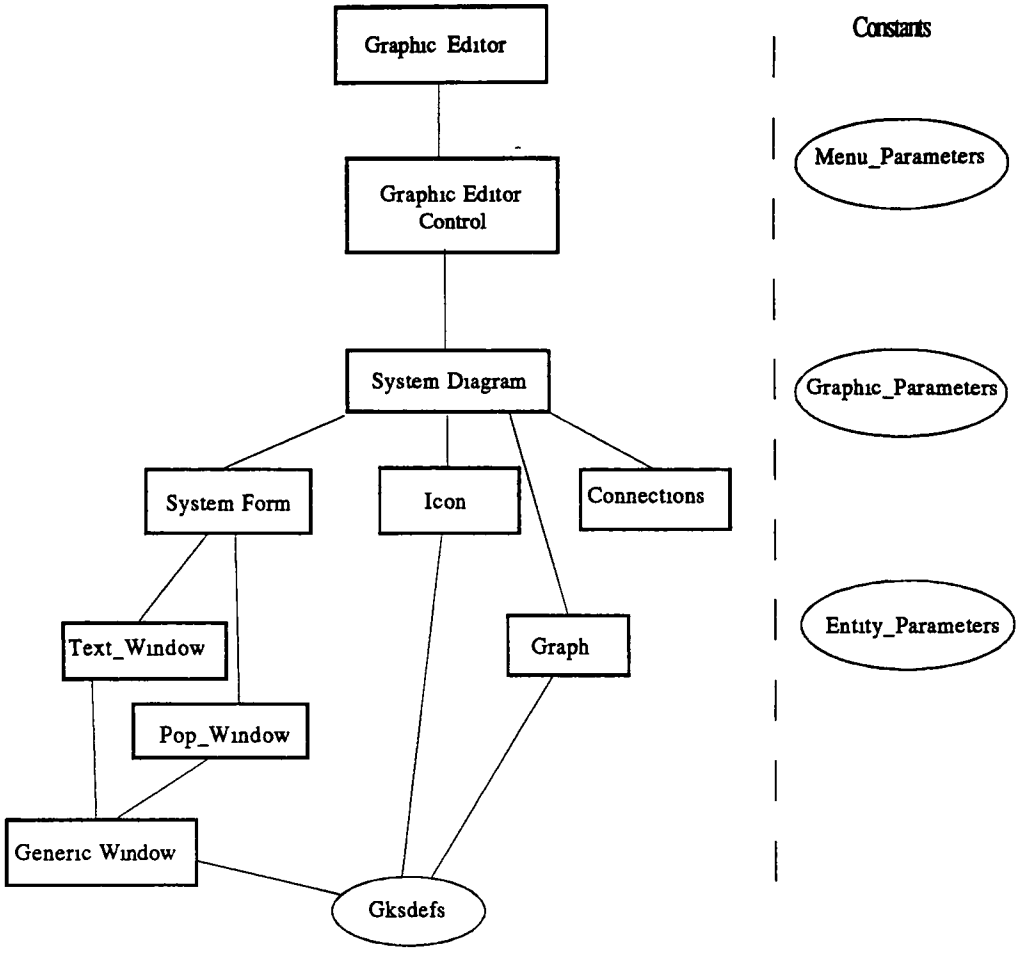
[2] [CTRL] followed by a letter means that the control key is pressed down at the same time as the letter key

On VT240s and VT340s the arrow keys are used to move the graphic cursor around. VS200s and GPXs use the mouse The left button triggers a selection or pick operation, while the middle button triggers an abort or break operation

The status window displays indicators of the current status of the package and on which phase it is in. This window is subdivided into three parts· *status, phase* and *error*. The status part displays the status on the current option i e. selection, working, complete or error The phase part displays the current phase the system is in i e modelling, specification, design, simulation, etc The error index part displays the error index or uncertainty computed for any numerical calculations This index is only displayed following numerical calculations

The software structure of the graphic editor is shown in Figure 20. The Graphic_Editor_Control package provides the menu control functions - display menu, selection of an option, perform option and deletion/exit from menu It controls the opening and closing of GKS for the package The Graphic_Editor_Control package also initialises the MSDI predefined windows, views and transformations

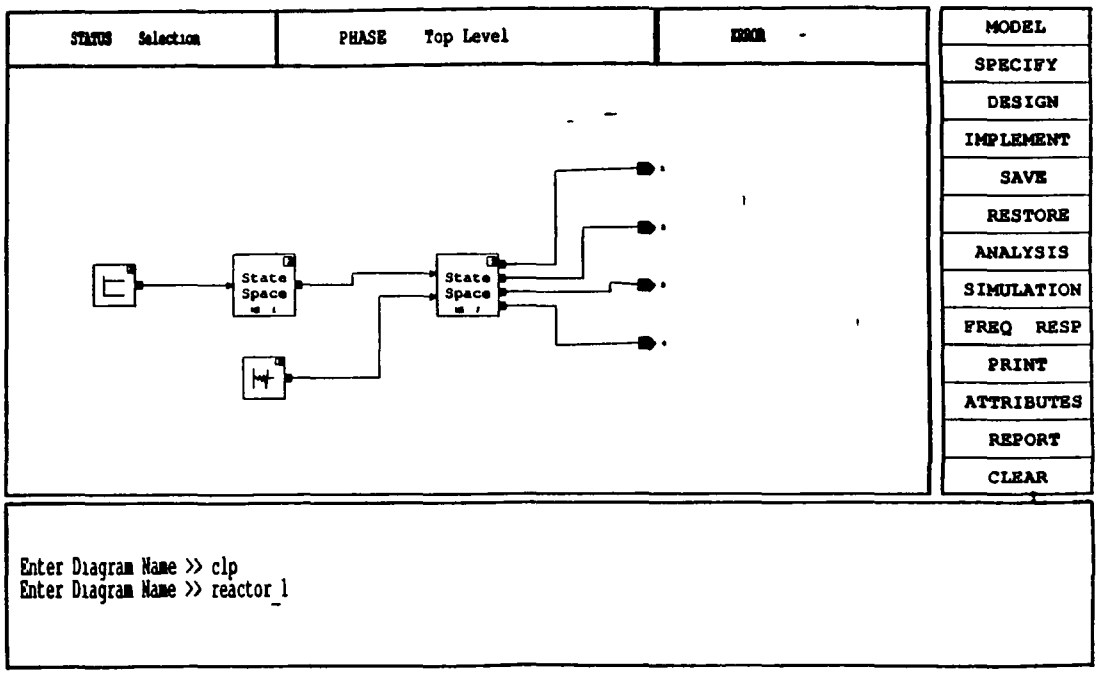
Figure 20: Graphic Editor Software Structure



To ease the changing of the graphical output of the prototype or porting it to be used on other terminals, the key parameters for the prototype have been isolated into the Menu_ Parameters, Graphic_Parameters and Entity_Parameters packages Thus, for example, to change the size of an icon only its size m the Graphic_Parameters package needs to be changed

Figure 21 shows the initial display a user sees on entering menu-mode

Figure 21: MSDI Top-Level Menu



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

The term entering a phase means that the user enters a sub-menu that provides the relevant options for that particular operation or phase.

The attributes option takes the user into a submenu that allows system parameters to be modified if required. Attributes such as the colour of an icon, its size or the text font can be changed.

The report option automatically generates a report of the design session using the information stored in the session log. This report displays the system diagram, defines the contents of each block, draws simulations and frequency plots saved during the session and design options selected. This report is outputted in a straightforward manner in a form to be processed by VAX Document, a word processor package. VAX Document uses predefined tags to format a report into paragraphs, tables and graphic files. For more details on VAX Document see [72]. The report is outputted, including these tags, with the graphics being stored in file in Postscript form. To get a hardcopy of the report the VAX Document processor is used to read the file and convert it into the device specific format.

A menu is exited by generating a break condition. This is achieved by pressing [CTRL] U on VT240s and VT340s and the middle button on the mouse on VS200 and GPXs.

5.3 Modelling

The modelling phase is where the user develops/constructs a model of the plant. Controllers are not added in the modelling phase as they were considered to be a different type of object. After using the prototype for a few design studies, this was found not to be the case. In reality the controller should be considered just another object to be added to the system diagram and should be added from the same menu as a state space model or signal source. The design phase should be an option in the modelling phase, giving access to the design methods.

The plant can be modelled by adding, deleting, copying, moving, modifying, rotating or transforming entities. The prototype supports all the atomic components defined in Figure 8 and interconnections between them.

The user fills in the behavioural template of the object to be added and then dynamically locates the icon on the graphic window. Other operations such as modifying, moving, examination and deletion simply allow the user to select (using the graphic cursor, which is defined as a moving cross) the object from the graphic window and then perform the operation.

5.3.1 Discretization

The transformations available in the prototype are :

1. Discretization - of both state space models and transfer functions.
2. Transform a state space model into transfer function form.
3. Transform a transfer function into state space form.
4. Identification of a transfer function from input-output data.

The only discretization method implemented in the prototype is based on the the discretization of a state space representation producing a zero-order hold equivalent. The discretization algorithm uses the Pade Series approximation of an exponential of a matrix. Transfer function discretization is implemented by internally converting the transfer function to its state space representation, discretizing this state space model and then converting the discrete state space model to its transfer function representation.

The discretization method used assumes that a continuous model is driven by a zero-order hold and the output is sampled. This is the normal case in computer control D/As normally drive the plant with a constant signal between sampling intervals and A/Ds sample the output. The "simulation" or numerical solution of the continuous state space model also uses this algorithm during a simulation of the system diagram.

The discretization process requires the solution of $\dot{x}(t) = A\,x(t) + B\,u(t)$, with x(0) = x_o

The solution for this can be written as

$$x(t) = e^{A(t-t_0)}\,x_o + \int_{t_0}^{t} e^{A(t-\tau)}\,u(\tau)\,d\tau \tag{6}$$

Using (6) and assuming a constant input between sampling intervals the Pade series approximation was used. This algorithm was selected based on a survey of computational techniques for matrix exponentials given in [65]. The Pade approximation for e^A is defined by·

$$R_{pq} = [D_{pq}(A)]^{-1}\,N_{pq}(A) \tag{7}$$

where

$$N_{pq}(A) = \sum_{j=0}^{p} \frac{(p+q-j)!\;p!}{(p+q)!j!(p-j)!\;A^j}$$

and

$$D_{pq}(A) = \sum_{j=0}^{q} \frac{(p+q-j)!\;q!}{(p+q)!j!(q-j)!\;(-A)^j}$$

The roundoff error difficulties and the computing cost is minimised by exploiting a fundamental property unique to the exponential function $e^A = (e^{A/m})^m$

The algorithm chooses m to be equal to the largest power of 2 that sets the $\frac{\|A\|}{m} \leq 1$, where $\|\ \|$ defines the column norm of a matrix Using this $e^{A/m}$ can be reliably computed, and then the matrix ((e $^{A/m}$) ^{m}) can be computed by repeated squaring

For situations where the Pade algorithm fails numerically, a backup routine is tried This is the Taylor series approximation based on the Talyor series expansion for an exponential defined by

$$e^A = I + A + \frac{A^2}{2!} + \tag{8}$$

This method requires about double the work to compute the discrete model compared to the Pade approximation But it uses a different approach which does not require the calculation of a matrix inverse This gives us a redunancy built-in to make the discretization process more robust

Many other discretization methods are used today by engineers such as pole matching and bilinear transformation. These would need to be available in a complete MSDI implementation

5.3.2 Transformation of State Space Representation to a Transfer Function

To convert from state space representation to frequency domain representation Leverrier's algorithm is used The transformation is changing a 4-tuple (A, B, C, D) to a transfer function of the form:

$$G(s) = C\,(sI - A)^{-1}\,B + D$$
$$= C\,\frac{adj(sI-A)}{|sI-A|}\,T + D$$

Leverrier's algorithm computes adj(sI-A) and the coefficients of | sI-A | recursively. It calculates the coefficients in the expansions

$$adj(sI - A) = R_{n-1}\,s^{n-1} + \quad + R_0$$

and

$$|sI - A| = s^n + a_{n-1}\,s^{n-1} + \quad + a_0$$

as follows.

$$R_{n-1} = I \qquad a_{n-1} = -tr(A\,R_{n-1})$$
$$R_{n-2} = A\,R_{n-1} + a_{n-1}\,I \qquad a_{n-2} = \frac{-1}{2tr(A\,R_{n-2})}$$

$$R_0 = A\,R_1 + a_1\,I \qquad a_0 = \frac{-1}{ntr(A\,R_0)}$$

where n is the order of the system, tr means the trace of the matrix and I defines an (nxn) identity matrix

Leverrier's method gives a straight foward error check on numerical performance from

$$Error = A\,R_0 + a_0\,I$$

Other methods are available that are more efficient In general, they involve the conversion of A, by similarity transformation into companion form The coefficients of det(sI-A) and adj(sI-A) are easily obtained from this form But this method is subject to numerical difficulties and there does not seem to be any convenient way of checking the results

5.3.3 Transformation of a Transfer Function to a State Space Representation

The converse problem of converting from transfer function form to state space form (also called realisation) implemented in the prototype assumes that the transfer function matrix is in strictly proper form (i e $G(s) \rightarrow 0$, as $s \rightarrow \infty$) The prototype ensures transfer functions are maintained in this form, by dividing them out if the order of numerator $\geq$ order of denominator and adding the integer part of the result to the feedforward term

The algorithm used expresses each column of G(s) (or G(z)) as a polynomial vector divided by a common denominator Defining the least common denominator of the i^{th} column as

$$s^{n_i} + d_{in_i-1}s^{n_i-1} + \quad + d_{i0}$$

and the corresponding numerator vector as·

$$q_{in_i-1}\,s^{n_i-1} + \quad + q_{i0}$$

the state space representation is defined as ·

$$\mathbf{A}=\begin{bmatrix} A_1 & 0 & & & 0 \\ 0 & A_2 & 0 & & 0 \\ \vdots & & & & \cdot \\ 0 & & & 0 & A_m \end{bmatrix}$$

$$\mathbf{B}=\begin{bmatrix} B_1 & 0 & & & 0 \\ 0 & B_2 & 0 & & 0 \\ \cdot & & & & \cdot \\ 0 & & & 0 & B_m \end{bmatrix}$$

$$\mathbf{C}=[C_1 \qquad C_m]$$

where:

$$A_i=\begin{bmatrix} 0 & 1 & 0 & & & 0 \\ 0 & 0 & 1 & 0 & & 0 \\ & & & & : & \\ -d_{i0} & & & & & -d_{in_i-1} \end{bmatrix}$$

$$B_i=\begin{bmatrix} 0 \\ \\ 1 \end{bmatrix}$$

$$C_i=[q_{i0} \qquad q_{in_i-1}]$$

This state space form is controllable by construction but not necessarily observable Another conversion algorithm should be added to ensure the construction of an observable form

5.3.4 Identification

The other transformation included in the prototype was least squares identification of a transfer function from experimental data The identification is broken down into two stages

1. Choice of model structure best suited for fitting the data
2. Analysis to derive the coefficients associated with the chosen model order

The identification is implemented recursively to avoid the construction of large matrices This algorithm derives the coefficients of a discrete transfer function

$$G(z)=\frac{b_{n-1}\,z^{n-1}+\quad+b_0}{z^n+a_{n-1}\,z^{n-1}+\quad+a_0}$$

The identification procedure is an iterative one. First the order of the model to be identified is selected. Then the coefficients for this model are computed using the recursive least squares algorithm. A figure of merit is then calculated which indicates how closely the shape of the model characteristic compares with that of the plant. This figure of merit calculated is the loss function:

$$V = e^2 = (A(z).(y - ym))^2.$$

where A(z) is the denominator of the transfer function computed and y is the output sequence used to identify the model. The terms ym are obtained by driving the computed transfer function with the input sequence used to identify the model. The loss function V will always decrease when the model order is increased. To test if the reduction in the loss function is significant when the number of parameters is increased from n_1 to n_2 the following test is used:

$$S.F. = \frac{V_1 - V_2}{V_2} * \frac{N - n_2}{n_2 - n_1}$$

where V_i is the value of the loss function for a model with n_i parameters, and N is the number of input-output pairs. It has been shown that the quantity S.F.(i.e. the *significant factor*) should be at least 3 for the corresponding reduction to be significant with a 5% confidence level (see [66]).

5.3.5 Macro Block Definition

The define option allows the creation of a *macro* block. It takes the current system diagram and forms an EvFn based on its composition. This macro is stored under a unique name in the database. It can be recalled at any time for inclusion into a diagram. An *empty macro* can also be created. This is a macro which only has its number of I/O ports defined. No internal structure or EvFn is defined. This is achieved by adding a macro not already in the database. A macro template is added to the system diagram. The contents of this macro block can be defined at a later stage.

This facility was included to allow top-down design strategies to be tested. This is where a design or plant model is not defined until latter on in a design cycle. During the use of the prototype, this empty macro facility was not found to be very useful.

An extract option allows a sub-set of the current system diagram to be selected for use in a simulation or design work. This allows a sub-set of a complex design to be extracted and worked on in isolation.

Other options in the modelling phase allow titles to be added to the diagram or individual nodes. Notes can be made on decisions taken. These notes are recorded in the database in sequential order. These notes can be recalled at various times or printed out in the design report.

An auto-relayout option has been included to aid the user to produce aesthetically acceptable diagrams for an arbitrary layout. The algorithm used is basic. It snaps icon block centres to predefined grid points on the graphic window, and sets all connection lines to change direction at 90 degrees. This usually helps "clean" up a diagram. To achieve a much higher level of performance as would be needed by a commercial system, an algorithm based on

tree or flow diagram layouts would need to be developed. Currently work on this aspect of CACE is being investigated in the University of Wales, under Chen, Barker and Townsend for their CES package. Another good starting point is [67]

5.4 Specification

The specification section as outlined in the previous chapter requires a set of primary indicators of performance that can be combined to form other criteria This is a major research area on its own. Solutions to this problem have been defined by researchers for very specific problem areas. An example is James's work in [1] on the design of SISO lead-lag controllers

To support the design facilities in the prototype the following criteria were defined to specify performance. The are divided into time domain and frequency domain criteria. The time domain criteria are.

- M_o - Max overshoot of response divided by final value.
- t_s - time to settle within 2% of the final value

The frequency domain criteria included are .

- BW - Bandwidth
- Gain Margin
- Phase Margin

The prototype only aids in specification for SISO systems. The time domain specifications assume the user is trying to approximate a second-order system It uses the defined M_o and t_s to compute the damping factor and natural frequency using

$$\varsigma = \frac{|\ln(M_o)|}{\sqrt{\pi^2 + (\ln(M_o)^2)}}$$

$$w_n = \frac{3\ 912023}{\varsigma\ t_s}$$

For continuous models, the poles are computed using the normal second-order equation The equation poles for the discrete model are (assuming they are located inside the unit disc)

$$z = e^{Tsample*s} = e^{(-w_n\varsigma + jw_n\sqrt{1-\varsigma^2}*Tsample)}$$

where T_{sample} is the sample period of the discrete model These models give the dominant pole locations.

The specification section of the MSDI package needs to be researched further It needs to be built up and encompass verification. The verification could use temporal logic together with other AI techniques to prove characteristics of a model This would be particularly worthwhile for systems with safety implications This is a field of much active research today, particularly in the realm of software engineering As control theory is a much more mature field it should be possible to use the developments in theorem proving and temporal logic to great advantage

5.5 Design : Analysis

The design phase is where controllers are added to the system diagram to improve overall performance. The final objective of controller design is to create an implementation that will meet the specified goals. Theoretical formulation is only one step on the road The prototype package concentrates on supporting the development of digital controllers All the design methods available support linear-time-invariant MIMO models. The design methods included are :

1. Pole Placement
2. Optimal Controller design
3. Inverse Nyquist Array
4. Observer design

The algorithm used for pole placement determines a state feedback matrix K for a MIMO continuous or discrete state space model by preliminary reduction of the model to Hessenberg canonical form using orthogonal similarity transformations. If all the states are not defined as accessible an observer can be designed Observers are designed using the same algorithm as pole placement as the control problem and the state estimation problem are equivalent Either full order or reduced order observers can be designed depending on the number of states accessible

The optimal control problem solved is as shown in (1), (2) and (3). The prototype solves this using the Dynamic Programming algorithm outline in Example 1.

Both the pole placement and the optimal design methods allow either state feedback or output feedback to be used For output feedback, an observer is first designed to estimate the states and then a state feedback controller is designed

The Inverse Nyquist Array (INA) algorithm allows the array to be drawn by itself or to have the Gershgorin circles superimposed to indicate dominance The user iteratively specifies the compensator required to give the required performance after decoupling I/O pairs. This method only works for a model with the same number of inputs as outputs

The INA algorithm makes use of a key algorithm in the package which computes the frequency response of a model This algorithm computes the response using the state space representation This state space model is converted into upper Hessenberg form using Eispack routines From this converted form the response is easily calculated using

$$G(jw_i) = C(jw_i I - A)^{-1}B + D$$

for all w_i of interest

This algorithm is based on the work of Laub in [68]

The analysis options are used in conjunction with the design options to assess the performance of system diagram / block in the system diagram The analysis functions available allow the calculation of poles and zeros, eigenvalues and eigenvectors, controllability and observability. These analysis functions are based on Linpack and Eispack Fortran routines The controllability and observability functions use the singular value decomposition (SVD) to compute the rank of the controllability and observability matrices :

Controllability Matrix = $[B, \quad AB, \qquad A^{n-1}B]$

Observability Matrix = $\begin{bmatrix} C \\ CA \\ . \\ CA^{n-1} \end{bmatrix}$

The frequency response of of linear models is given using the algorithm described above The results can be plotted in Bode or Nyquist form The user selects the frequency range of interest and the no. of points to be computed in this range The algorithm takes care to account for phase crossing of the +/-180 degree line.

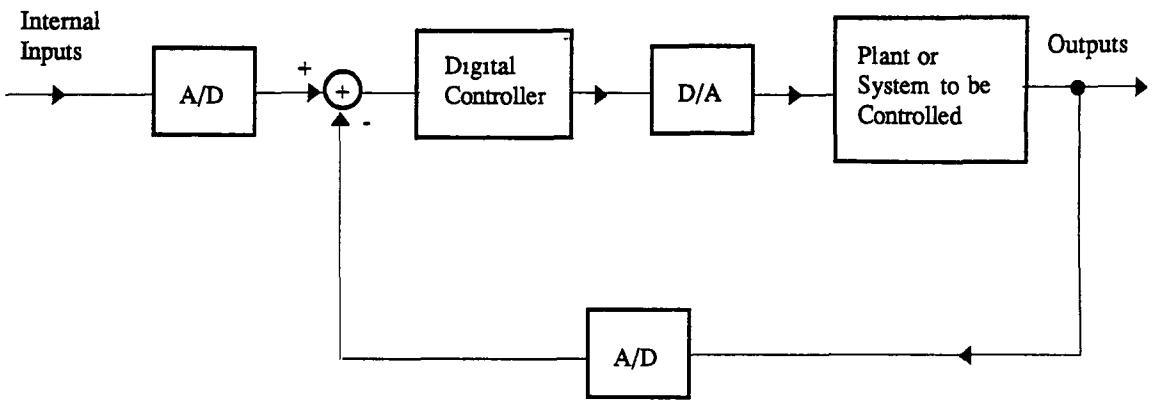
One extra option added to analyse performance gives the "primary" indicators of performance defined by McFarlane and Pang in [29] These are frequency domain indicators They are composed of the *characteristic value decomposition* (CVD) and *singular value decomposition* (SVD) of the model. The CVD is an indicator of the stability of the model (i e via the generalised Nyquist criteria) and the SVD is an indicator of the closed-loop performance of the model Also the difference between the magnitude of the characteristic gains (from the CVD) and the magnitude of the principal gains (from the SVD) is an indicator of the robustness of the system i e performance in the presence of modelling errors McFarlane and Pang have termed these as the primary indicators of a model as they provide a complete description of the system This option allows the graphical presentation of the CVD and the SVD of the model to the user Tabulation or computation of a specific value of one of the loci is available from the graphical representation of these CVDs and SVDs

5.6 Implementation of a Controller

The implementation of a controller is where the hardware/software needed to implement the designed controller is selected. The prototype does not provide support for continuous controllers. It only supports digital controllers

The digital controller structure the prototype supports is shown in Figure 22 It is composed of A/Ds and D/As to interface to the continuous plant (if needed) and a microprocessor executing an algebraic algorithm Choosing an appropriate microprocessor, A/D, D/A and algorithm are important to achieve the performance required Factors such as computation speed, wordlength and conversion time are important

Figure 22: Digital Control System



The prototype provides a library facility where the user can add definitions of available microprocessors, A/D, D/A The algorithms available are built into the package and cannot be added to. These algorithms are state feedback, state feedback with full or reduced state estimation, a a PID self-tuner The parameters that the user can define for these hardware components are shown in Table 17

Table 17: Hardware Component Model Parameters

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

The algorithms are direct implementations of the controllers designed For example for output feedback the observer and state feedback algorithm computes first the estimated state vector $\hat{x}$ via

$$\hat{x}_{k+1} = \Phi\, \hat{x}_k + \mathrm{T}\, u_{p,k} + K\, [y_{p,k} - H\hat{x}_k]$$

(see Astrom and Wittenmark in [39]), where u_p is the vector of control inputs to the plant and y_p may contain plant measurement variables as well as reference inputs or measured external disturbances , in the case of reference and disturbance modelling The observed state vector is then used in·

$$u_{p,k} = -L\, \hat{x}_k$$

where L is a constant state feedback matrix

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 ouputs 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.

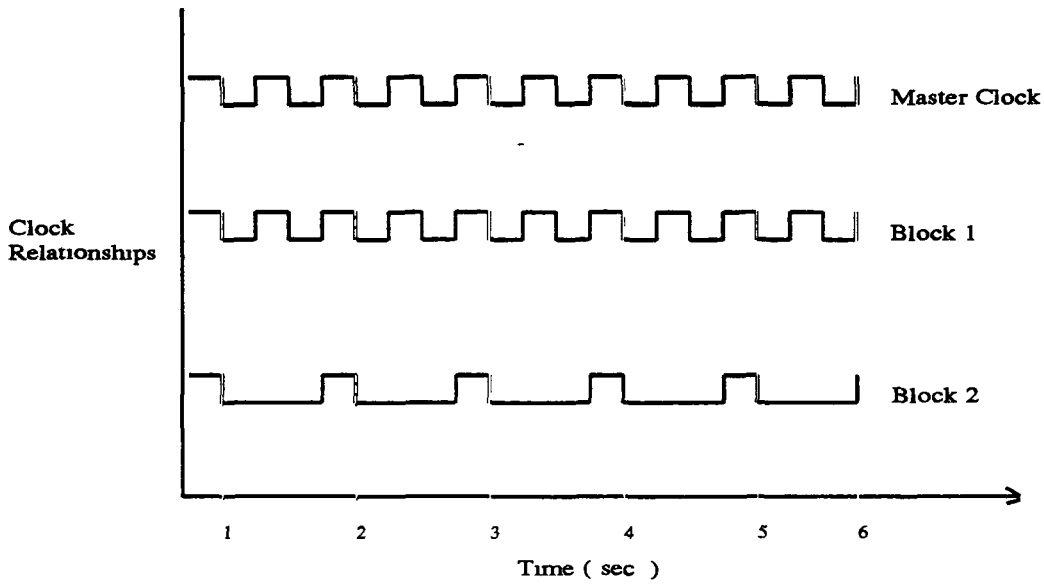
5.7 Simulation

Simulation of the system diagram can be performed at either the behavioural level or the implementational level.

For any simulation the system diagram is analysed first to ensure it can be simulated - i.e all entities defined (e.g no empty macro template) Then the system diagram is ordered This ordering starts at what are classified as independent entities An independent entity does not have its current output dependent on its current input. These entities are signal sources, state space with no feedforward matrix, transfer functions and macro blocks with no inputs Then the diagram is recursively passed over selecting blocks that can be simulated if their input is driven by an output previous ordered above them When all the blocks are reachable (i e inputs are driven by an output in the ordered list) a master clock time is selected.

The programme computes the lowest common divider for all the different sample times in the diagram The user, can if desired, override this automatic clock period setting Each entity in the ordered list has its own individual clock which is initialised to zero Then at each tick of this master clock (during the simulation run) the ordered diagram is sequentially stepped through First the individual clock of the entity is checked to ensure that its output should be computed at this time Each pass over the ordered diagram adds the master clock period to the entity's individual clock. When an individual clock is found with its clock $\geq$ its defined sample period, the output is computed and the clock re-set to zero An example of this is shown in Figure 23 for a diagram with two entities Block 1 has a sample period of 0 5 seconds Block 2 has a sample period of 1 0 second The master clock is computed to be 0 5 seconds Thus the output of block 1 is computed at every tick of the master clock while block 2 's output is only computed every second tick

Figure 23: Simulation Timing



Note Outputs computed on the trail edge of a pulse

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 A 4-bit fractional two's complement representation is

0 875	0 111
0 125	0 001
0	0 000
- 0 125	1 111
- 1	1 000

This example also illustrates the asymmetrical range of fixed-point arithmetic. This needs to be included in our simulation model.

The main advantage in using fixed-point (i.e. in two's complement representation) compared to floating point is the simplicity of the hardware for adding and subtracting. No distinctions need to be made as to what the signs and magnitude of operands are and a single adder unit plus a simple complementer circuit is sufficient to perform addition and subtraction. This is the main reason why most digital controllers use fixed-point today.

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 (i.e. VAX Ada compiler not defaulting to a larger wordlength). A representation clause defines how a data type is mapped to the underlying machine. Example 6 illustrates how this is accomplished.

Example 6: 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 cluse 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.

A data type is create for 2, 4, 8, 16 wordlengths. Any hardware defined in between these ranges is mapped on to one of these types. 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 an real piece of hardware. The output of the A/D is the input converted to its the 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.

If overflow or underflow occurs in any of the calculations in the fixed-point arithmetic the value to be calculated is set to the max. or min. available in the defined wordlength.

5.8 Code Generation

The prototype also supports the automatic generation of code for a 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 design controller parameters are written into this procedure.

This entire file is written in Ada can then be compiled using a Ada compiler The intention is to have a cross-compiler to target small microprocessor implementations such as would be used in embedded systems Such a cross-compiler would be XD-Ada from System Designers To test the prototype a PC target was used compiling the code with the Janus PC Ada compiler from RR Software The principle is the same The Janus Ada compiler and linker provide a command to trim off all procedures not used from an executable file This means that using the generic Ada package does not hit run-time performance An example of this is shown in the next chapter

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 from the effects of round-off, etc When non-linear modelling is included in the package, a full "software breadboarding" of a controller can be carried out prior to selecting, buying or building any hardware.

5.9 Limitations of Prototype

The prototype supports the main techniques need to complete a design from scratch except non-linear modelling and simulation This is the major limitation of the prototype

Also novel algorithms cannot be added to the library for inclusion in implementation simulation and code-generation This needs to be added particularly if the package is to be used for research work on implementation strategies

The command-mode is not as well defined as the menu-mode Also command macros need to be added This could be accomplished similar to MATLAB M files by parsing a file line by line This would be slow compared to online commands This would be a problem for design methods being implemented as macros One idea tested during the project was the compiling of each macro into an executable file Each macro command would be parsed into Ada code, compiled and linked into executable file (during the evaluation performed the object file had to be exported due to the nature of the Ada compilation system) This did not produce very satisfactory results because of the activation time associated with swapping one executable file for another and then back again To obtain the performance needed the conclusion drawn was that the macro command should be parsed into an intermediate form which can be parsed faster by the MSDI command parser This intermediate form would be syntatically proven already when the macro was defined so much of the work required to parse it would not be required again This is area that needs further research

Another limitation is the lack of support for continuous controller implementations Models of standard 3-term controllers could be included in the library of hardware.

5.9.1 VAX Ada and VAX GKS Performance Issues

One of the key points discovered during the project was the different way VAX Ada and VAX Fortran store arrays on memory. This means care needs to be taken when interfacing Ada to a Fortran routine. Arrays being passed to Fortran need to be transposed before being passed to the Fortran routine. Arrays being passed back from the Fortran routine also need to be transposed. This is only a problem for arrays of two or greater dimensions.

The VAX Ada implementation of certain constructs led to poor performance and had to be removed where possible in critical code sections. Two examples are the 'image/'value attributes. These create very poor runtime code. Another was when a large array (greater than 1000 storage elements) is initialised when it is declared. For this situtation, the current compiler V1.4 produces 10 times the amount of code as when the array is initialised using a loop in the body of the procedure. These issues are tied to the use of the VAX Ada compiler V1.4.

One of the problems found with VAX GKS V3.* was that rounding errors often meant that borders around a window or view were drawn off the screen. To overcome this, each transformation's world co-ordinates were defined with an added component (min. - 1.0e-6, max. + 1.0e-6) to eliminate the rounding effect. The active world co-ordinate range used was then (min., max.).

The prototype package has been tested on VT240, VT340, VS200 and GPX terminals. Some problems were seen on the VS200 using V2.2 of VWS (such as dropping characters when typing fast). Using a V3.0 or higher removed these problems.

5.10 Summary

The final prototype produced was able to take a design problem from initial modelling to final implementations. The main body of a full MSDI package has been implemented. A fuller set of analysis and design tools needs to be included. Also the command-mode of operation needs to be brought up to the level of the menu-mode.

The prototype contains over 65,000 lines of Ada code. The relative sizes of the major components outlined in chapter two are :

Table 18: 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 current executable image for the prototype (when compiled using /opt=time/debug) is 1869 blocks. This image could be reduced by approx. 30% when V2.0 of the VAX Ada compiler is released. All V1.* versions did not support code sharing between generics. V2.0 does support sharing of code between generics. As an object-oriented design method was used for the prototype, heavy use of generics has been.

CHAPTER 6

DESIGN EXAMPLES

This chapter demonstrates the use of the prototype package The first two examples concentrate on the type of analysis that can be performed on a model The third example takes a design from modelling to final implementation and evaluation of performance

First a transfer function representation of a MIMO system is analysed using frequency domain techniques Then a control system is designed for a state space representation of an exothermic catalytic reactor Finally a complete design using the prototype is presented for a coupled-tanks apparatus This complete design is from initial modelling to the final implementation of the controller in hardware and software

6.1 Frequency Domain Design Example

The plant model, taken from [69], 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)  ]

>
```

This model has two poles at -1 and -2 It is a minimum phase system but it is conditionally stable The primary indicators (i e the CVD and SVD for frequency range of interest) are shown in Figure 24 A simulation of the step response at input 2 of this uncompensated plant is shown in Figure 25 A generalised nyquist diagram for this plant model is shown in Figure 26

Note :

E1 and E2 on the diagrams represent the CVD curves
P1 and P2 on the diagrams represent the SVD curves

Figure 24: Design Example 1 : Primary Indicators

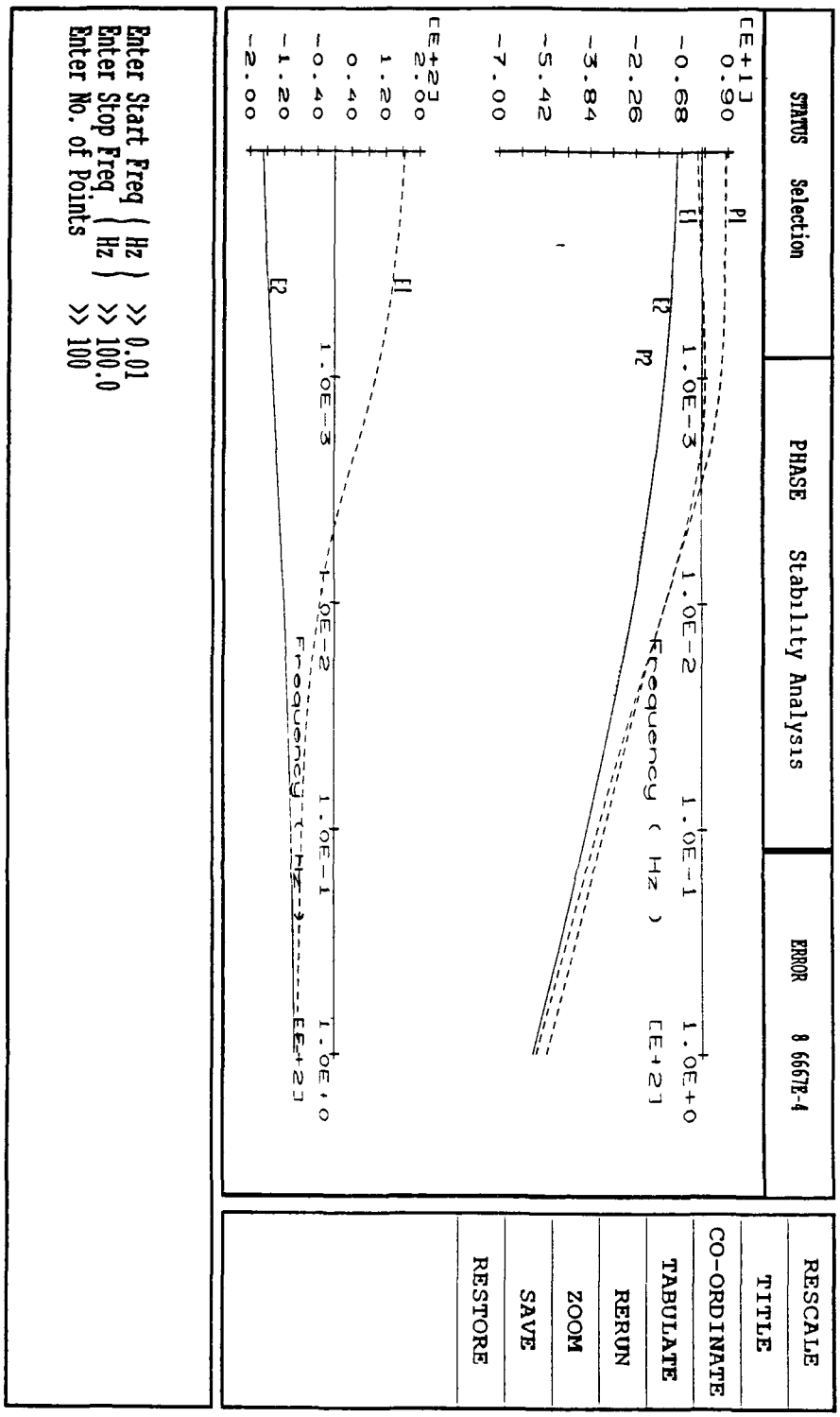
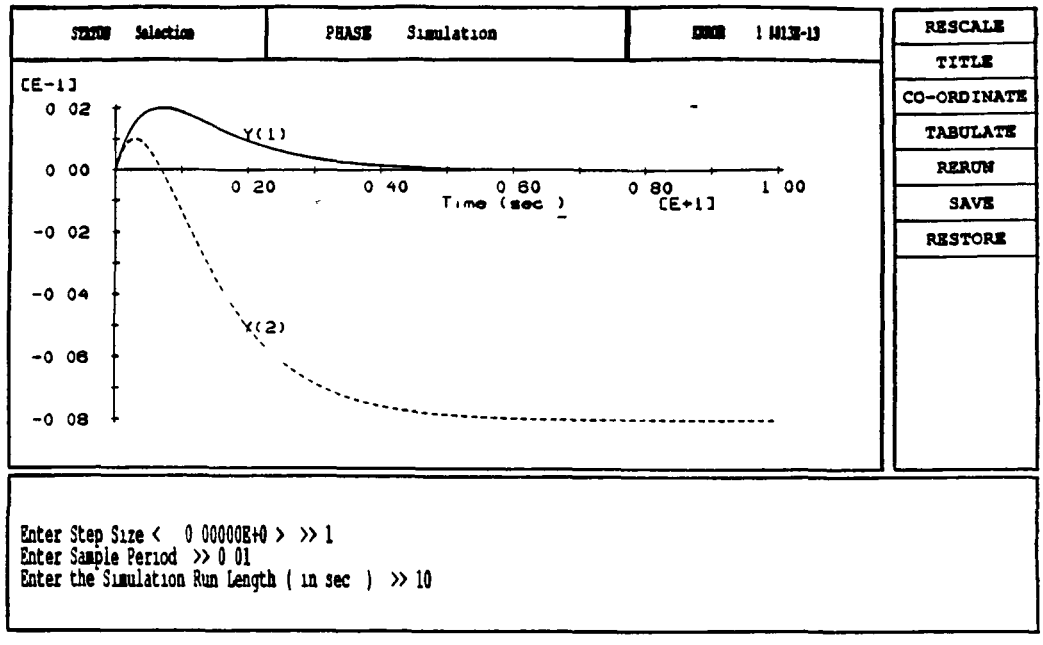
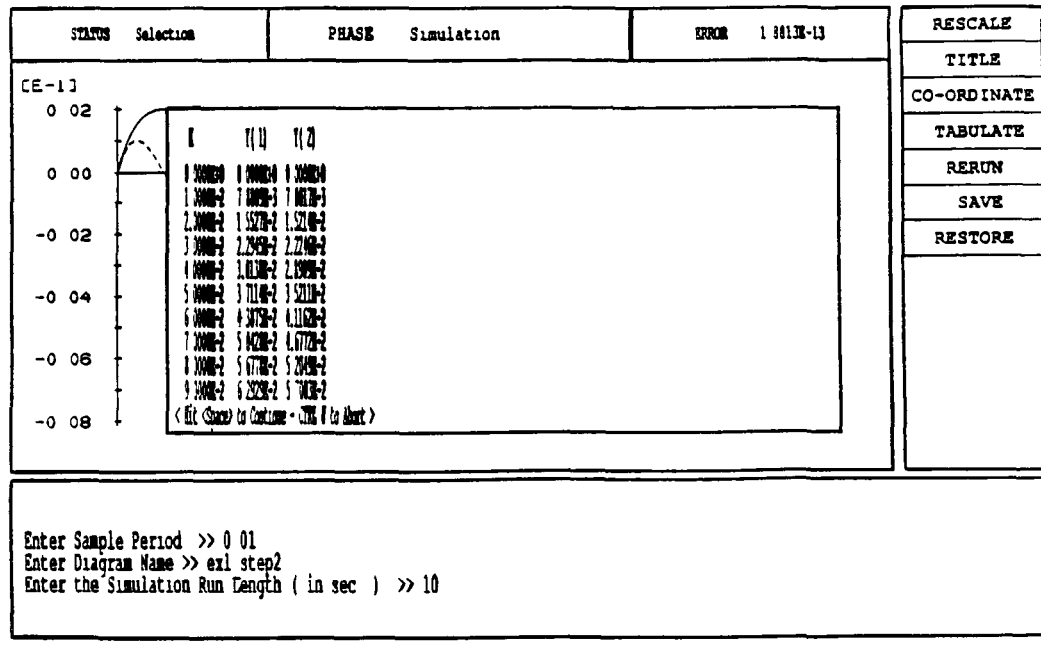


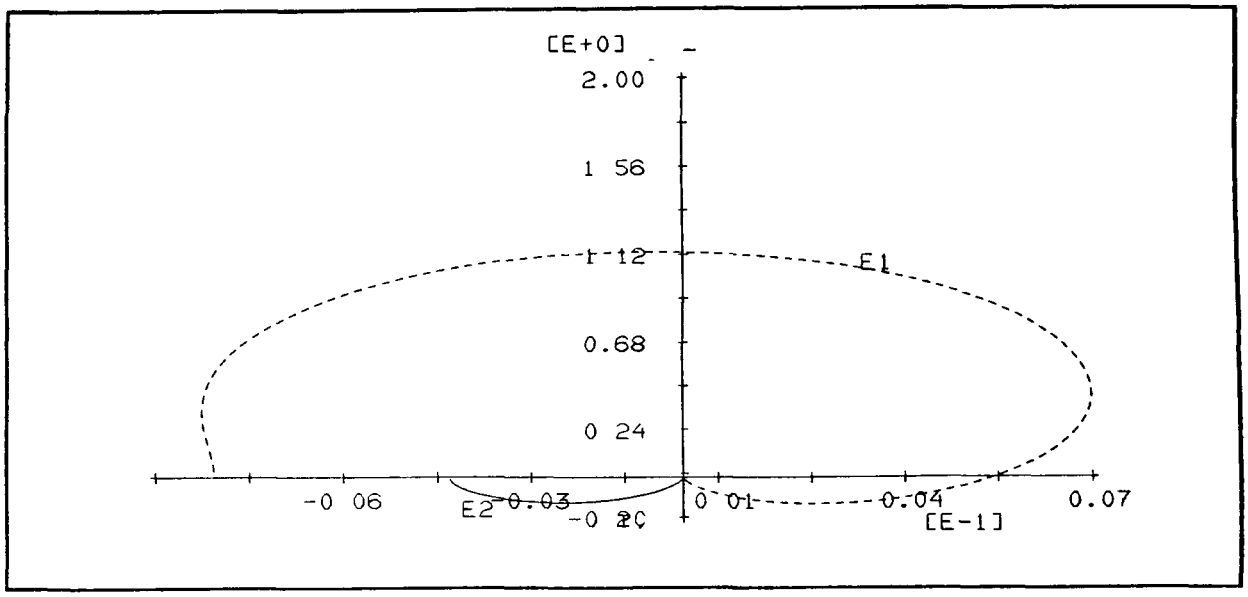
Figure 25: Design Example 1 : Step response of uncompensated plant





Note : Example of Pop-up window displaying table of simulation results.

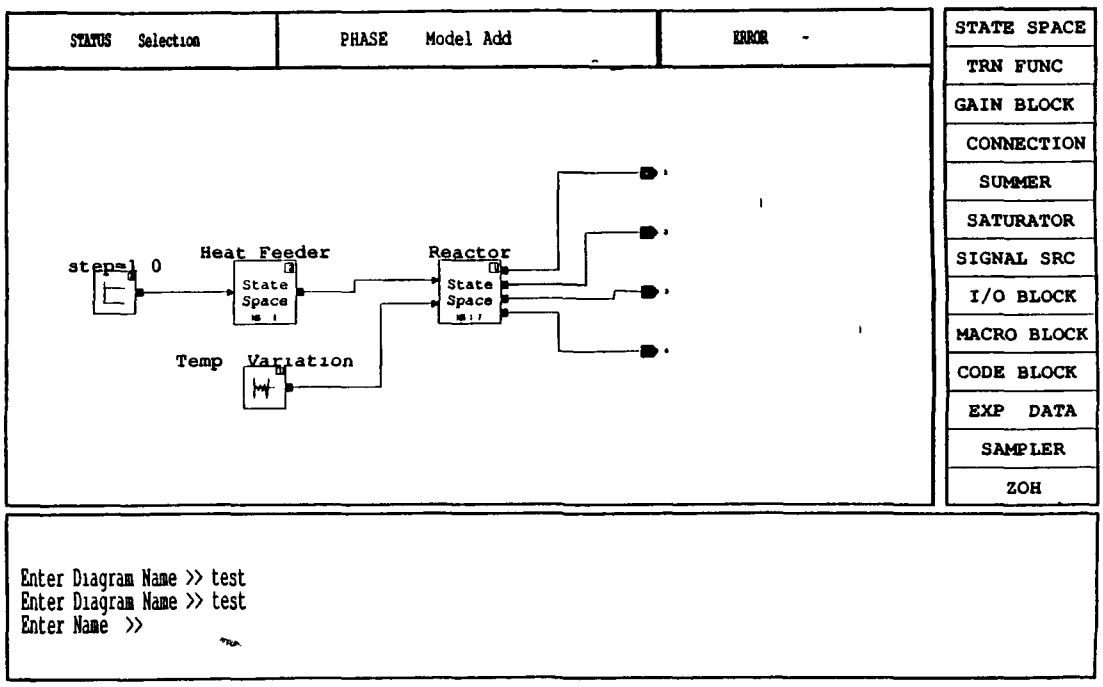
Figure 26: Design Example 1 : Generalised Nyquist Diagram



6.2 Time Domain Design Example

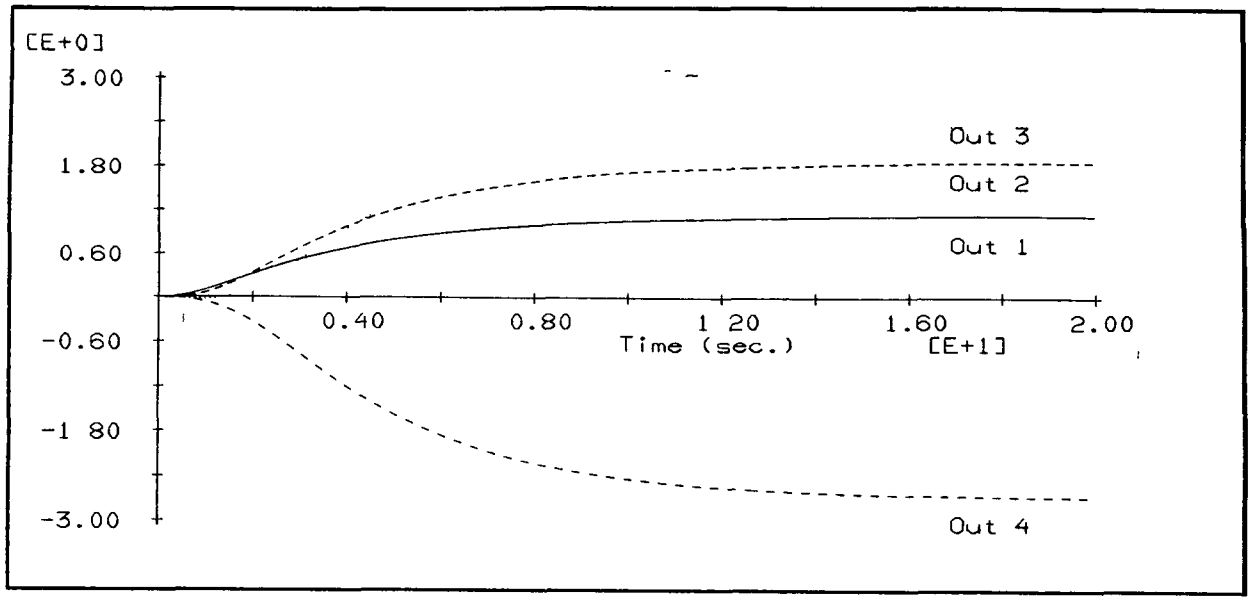
The second example is based on the model developed in [70]. The linear model of bed one was used to demonstrate how to design a controller The iconic representation of the system diagram for this model is shown in Figure 27 The elements of the state space models is shown are Appendix F

Figure 27: Design Example 2 : Reactor Diagram



The unit step response for this plant model is shown in Figure 28.

Figure 28: Design Example 2 : Step response of uncompensated plant



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 .

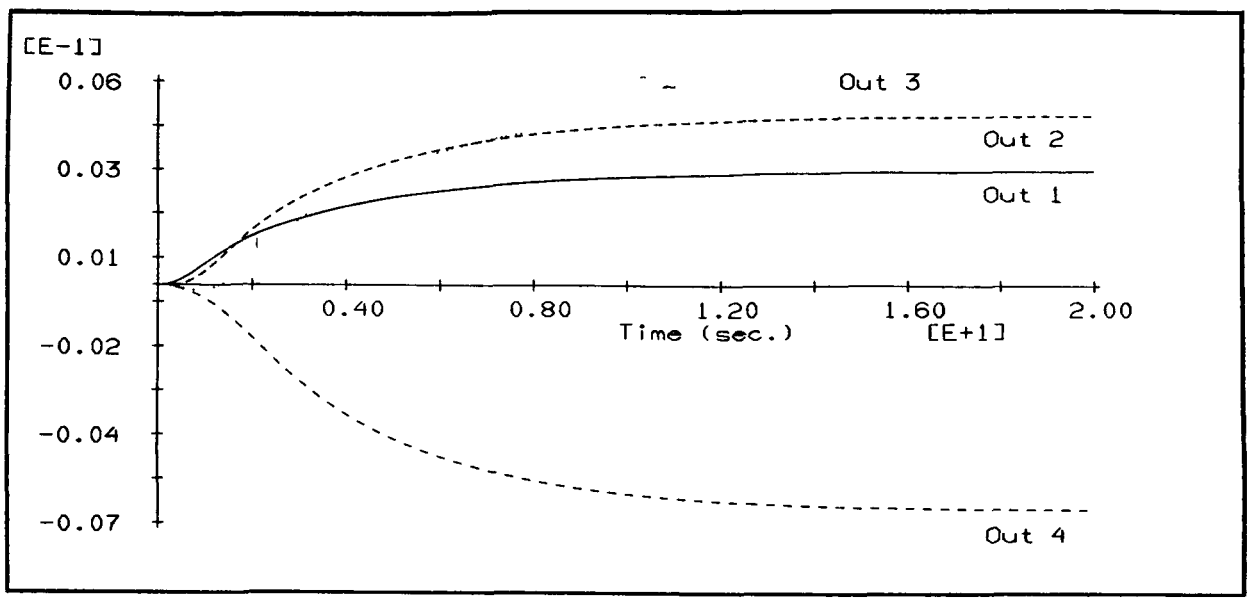
```
> Q = diag( [ 2.0 0.1 1.0 0.2 0.5 5.0 1 0 ] )
> R = diag( [ 5 0 1 0 ] )
```

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 29

Figure 29: Design Example 2 : Step response of compensated plant



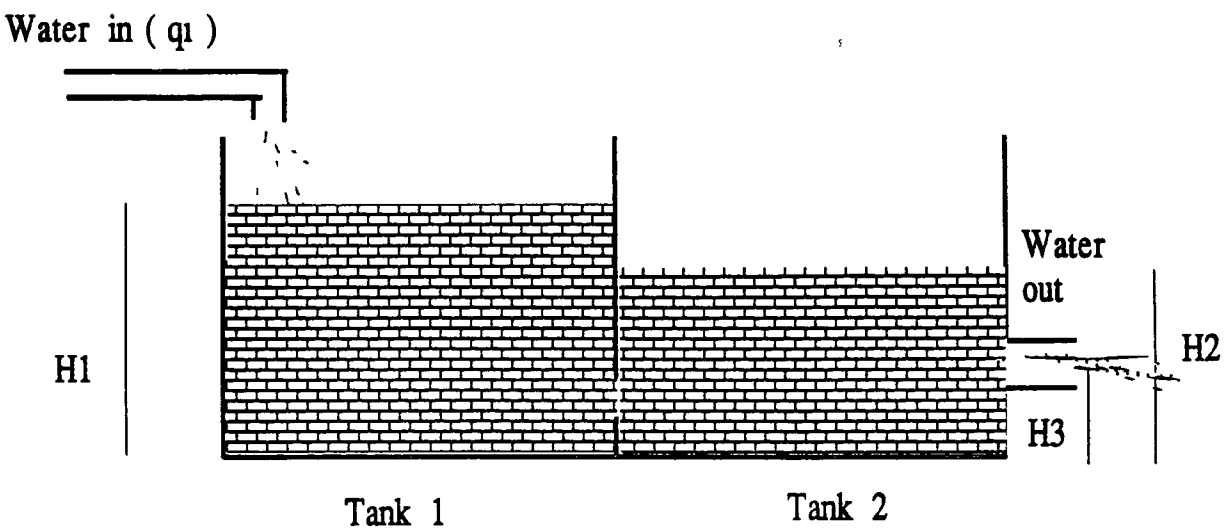
The definition of this controller has no real purpose other than to demonstrate some of the capabilities of the prototype. The pole placement design method could have been used just as easy to define the location of the poles of the closed-loop model where desired.

6.3 Full 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. Fluid control is a very widespread and important technology in industry. In most of these processes a desired amount of liquid must be available at all times for the successful completion of the operation, for example in the blending of chemicals

The basic experimental rig consists of two hold-up tanks which are coupled by an orifice The input is supplied by a variable speed pump, which supplies water to the first tank The orifice allows this water to flow into the second tank The water, which flows into tank 2, is allowed to drain out via an adjustable tap, and the entire assembly is mounted in a large tray which also forms the supply reservoir for the pump The basic design problem is to control the level of the water in the second tank by varying the speed of the pump A schematic of the coupled-tanks apparatus is shown in Figure 30.

Figure 30: Coupled-tanks Apparatus Schematic



Water is pumped from the reservoir into the first tank and is measured by a flow meter which is in the flow line between the pump and tank 1 The depth of fluid is measured using parallel track depth sensors which are stationed in tank 1 and tank 2 These devices perform as an electrical resistance which varies with the water level. The changes in resistance are detected and provide an electrical signal which is proportional to the height of the water

The apparatus is set up so that the motor drive can be driven by a voltage between zero and ten volts and the depth sensor outputs can provide outputs in the range zero to ten volts. For the design study, the depth sensors were set to supply a voltage between zero and five volts because of the A/Ds to be used to implement the controller. The A/Ds used were MetraByte's DASH-8, an 8 channel 12-bit high speed A/D board. The D/As consisted of MetraBtye's DAC-02, a 12-bit D/A board.

6.3.1 Modelling

The linear model for the apparatus [71] (derived using flow balances, and considering small variations in flowrates and water heights) is:

$$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} \frac{-k_1}{A} & \frac{-k_1}{A} \\ \frac{k_1}{A} & \frac{-(k_1+k_2)}{A} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{A} \\ 0 \end{bmatrix} q_i \tag{9}$$

For the design study H_1 = 15 0 cm., H_2 = 9.5 cm., H_3 = 3 cm., Q_i = 2000cc/min., A=98 cm^2

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 loss function and confidence factor as the order of the model increased was :

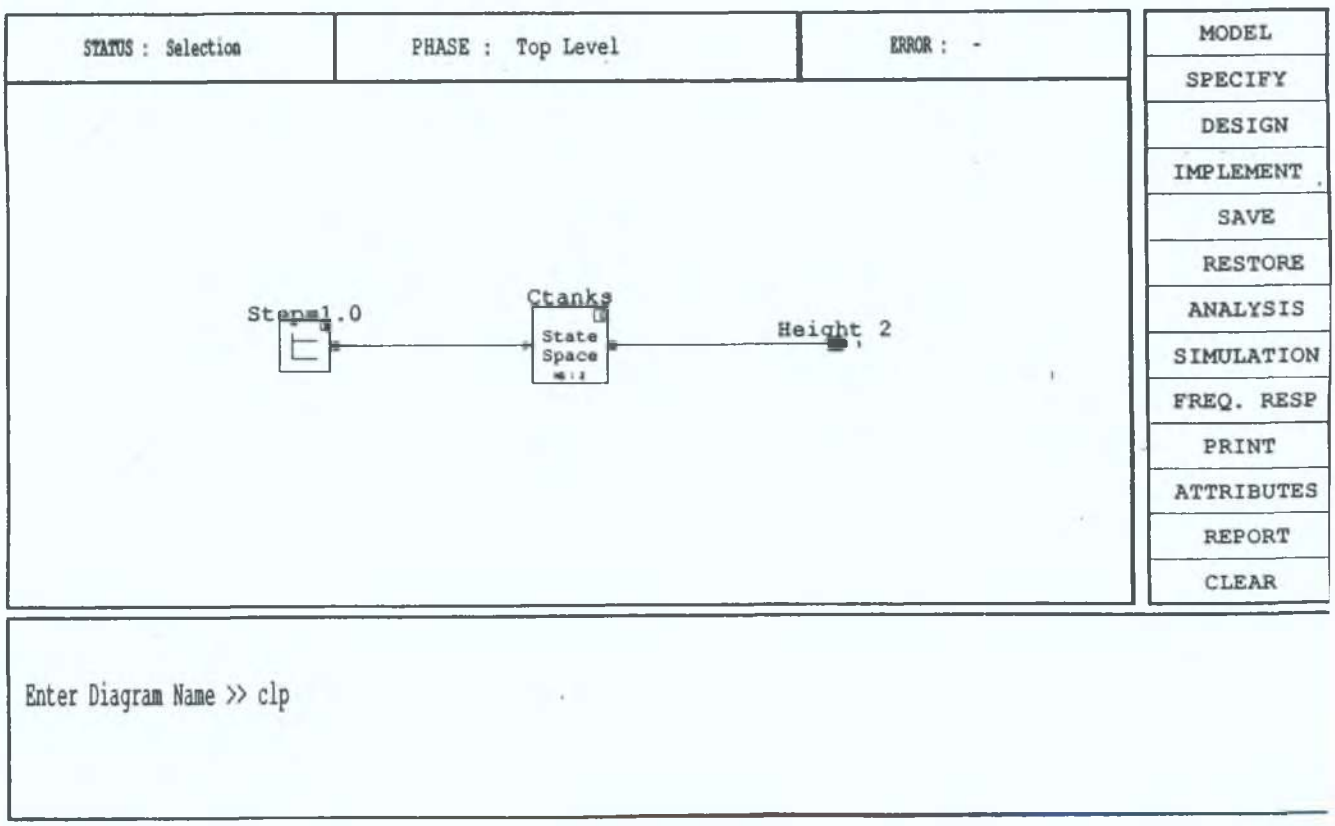
Order	V	S.F.
1	0.782	0.1041E+35
2	0.4137	0.1041E+35
3	0.4051	0.1552

This indicated that a second order model was suitable. The model identified was:

$$G(z) = \frac{1.946838E\text{-}2\ z + 4.268818E\text{-}2}{z^2 - 1.498736 + 5.197121E\text{-}2}$$

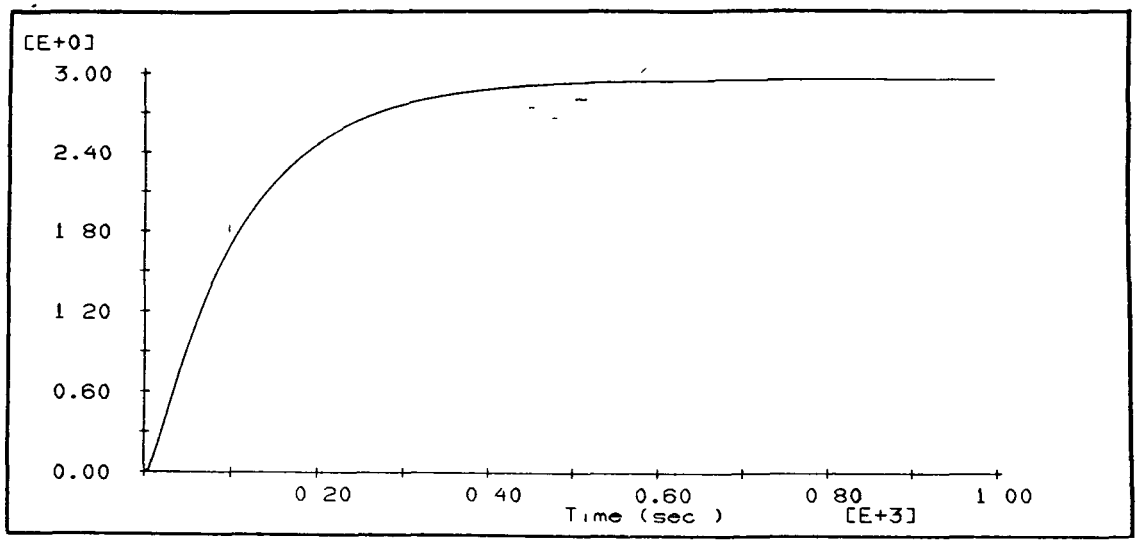
This model has poles at (0.9539167, 0.5448191299). This model was transformed into a state space model and the iconic representation of the diagram used for open loop simulation is shown in Figure 31.

Figure 31: Coupled-tanks Representation



The response of the model to a unit step response is shown in Figure 32.

Figure 32: Coupled-tanks Open Loop Unit Step Response

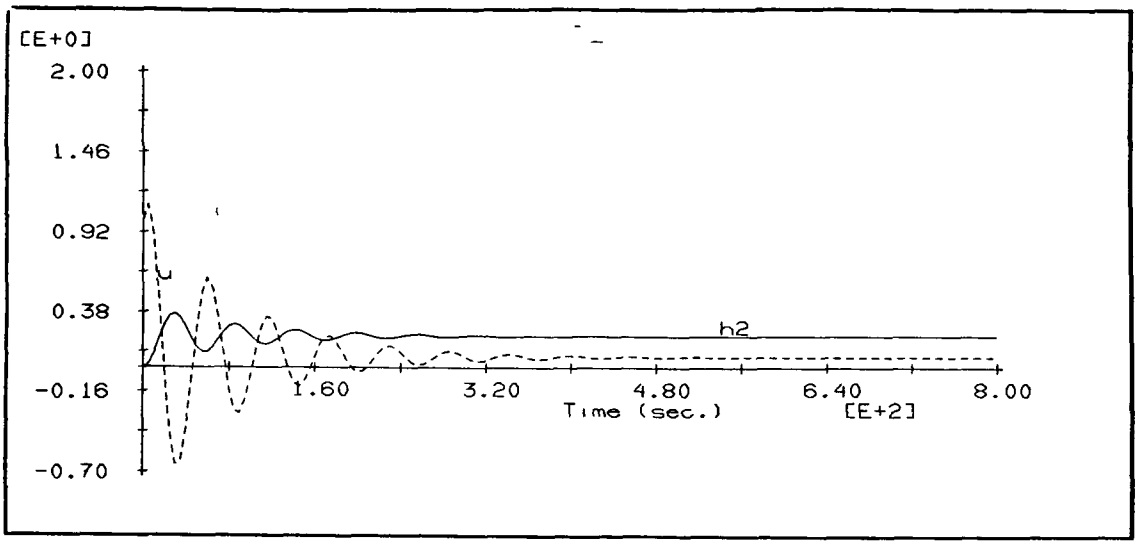


A controller was designed using pole placement. The desired poles were set to (0 6 +/- 0 2) The controller state feedback matrix computed was

```
K = [ 0 370287919734865  -0 1012640238418579 ]
```

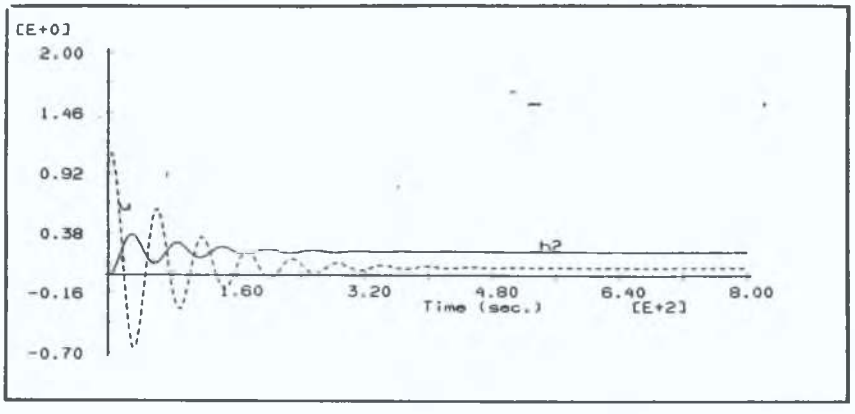
The step response of the compensated closed loop plant is shown in Figure 33

Figure 33: State Feedback Compensated Plant Step response

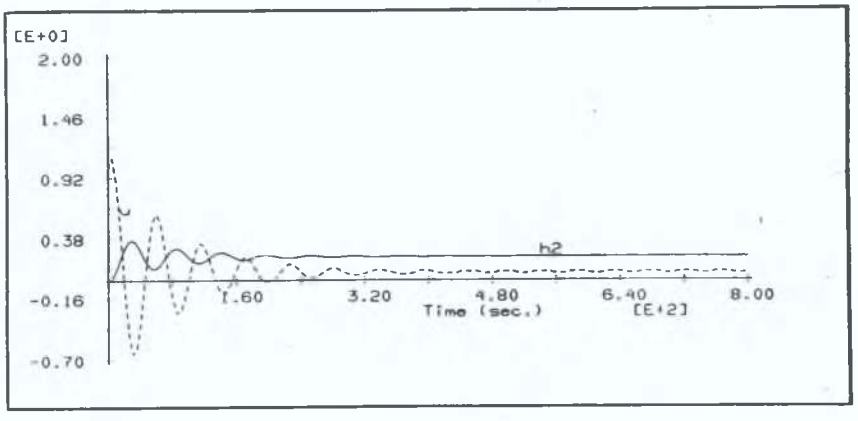


Then an implementation was defined using 12 bit D/A and 8 bit A/D with the microprocessor being defined as 4 bit, 8 bit and 32 bit in turn. The simulation of these implementations for a unit step response are shown in Figure 34.

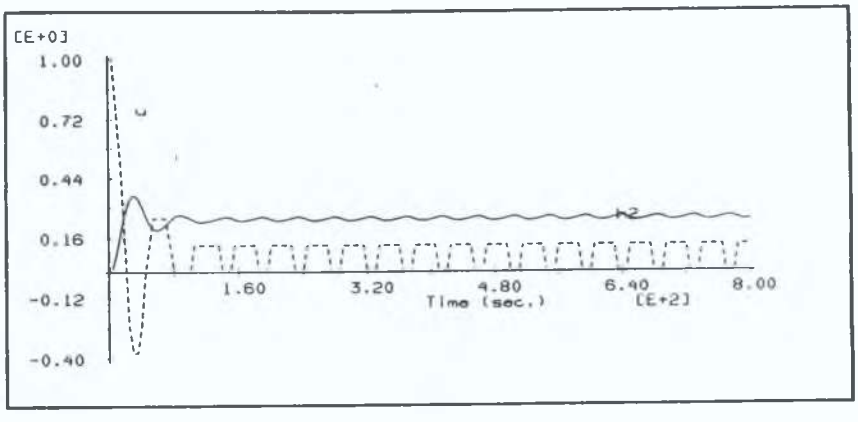
Figure 34: Simulation of 4, 8 and 32 bit based state-feedback controllers



32 bit wordlength



8 bit wordlength



4 bit wordlength

The reason for the difference in performance between the three implementations was caused by the different wordlengths used. This roundoff occurs both in the controller parameters and for the calculations made during the test runs. The differences in the coefficients of the feedback gain matrix K is shown in Table 19. Notice that the coefficients have change significantly for the 4 bit wordlength implementation.

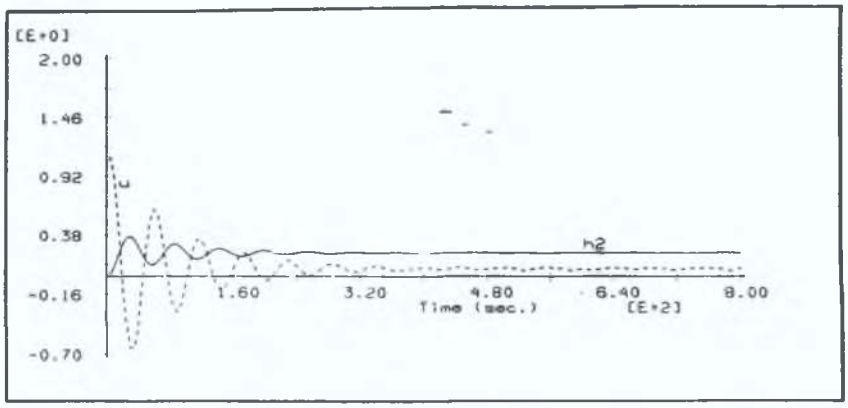
Table 19: Coefficients in 4, 8 and 32 bit controllers

Wordlength	K(1,1)	K(1,2)
4	0.250	0 000
8	0 3671875	-0 093750
32	0 370287919734865	-0 1012640238418579

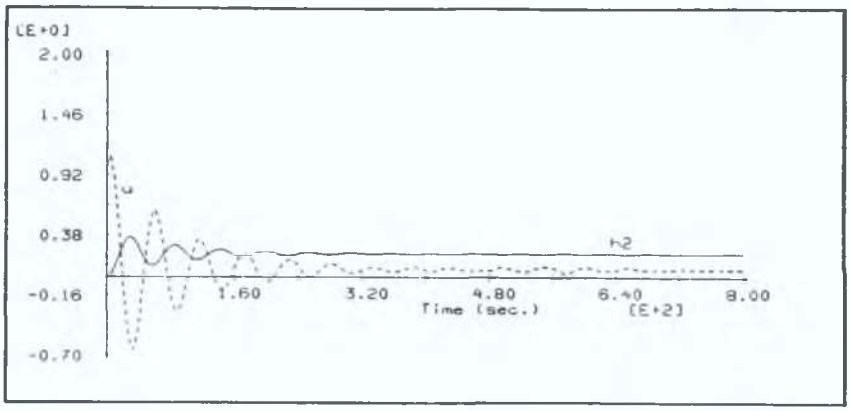
The roundoff in the feedback gain matrix is really pronounced in the 4 bit implementation This together with the roundoff involved in the computations is the reason for the changed performance. Based on this, a minimum of an 8-bit wordlength microprocessor should be used to implement this controller

The code generation facility in the prototype was used to produce an implementation of the controllers This code was compiled using the RR Software Janus Ada compiler and run on IBM PC The code produce specified 4, 16 and 32 bit fix-point mathematics The code produced to run on the PC implements these despite the fact the underlying processor is 16-bit. The recorded performance of these three controllers is shown in Figure 35

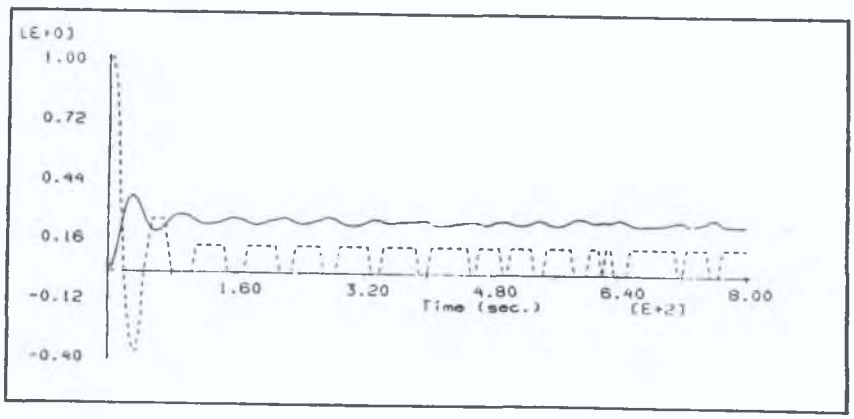
Figure 35: Actual Responses of Apparatus for 4, 8 and 32 bit based state-feedback controllers



32 bit wordlength



8 bit wordlength



4 bit wordlength

The actual response obtained is as predicated by the simulation of the implementations. The benefits of an implementation level simulation are that the final implementation can be checked out, through simulation, before applying the controller to the real plant. This

is important particularly for cases where the plant may be damaged by a poor performing controller or the hardware to implement a controller is expensive.

To further demonstrate the power of the prototype package, an output feedback controller was design The observer poles were designed to be located at (0.6, 0 6) which gave an observer feedback matrix of .

```
Ko = [ 8.1250000
       7.7890625 ]
```

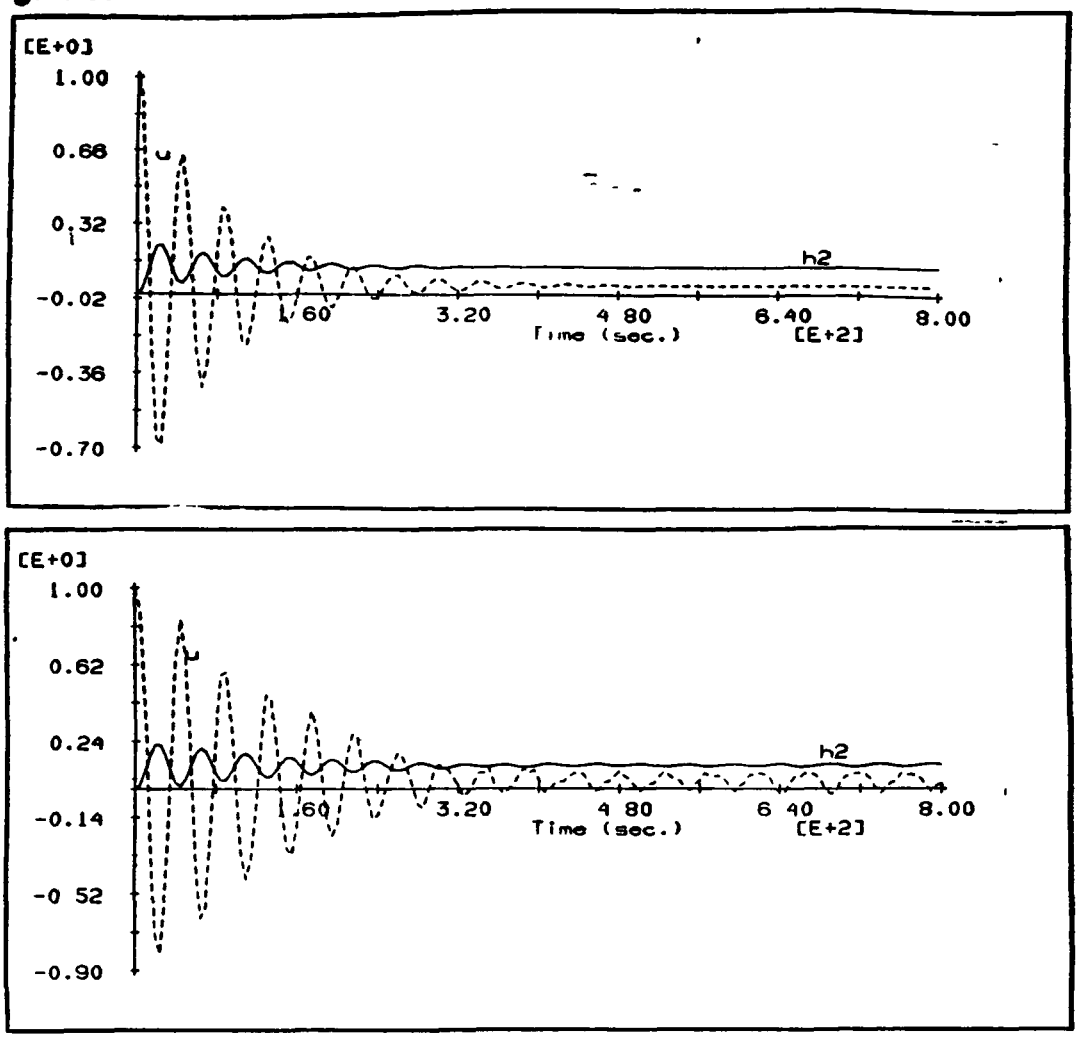
The output feedback algorithm computes

```
y = C * states

U = - K * states + v

States = A*states * B*u + ko ( Height 2 - y )
```

where A, B, C define the system, input and output matrices of the state space representation of the plant to be controlled. States contains the value of the estimated states (States is set to zero initially) K is the state feedback gain matrix and Ko is the observer feedback matrix

The simulation of this controller for 8 and 32 bit processors is shown in Figure 36.

Figure 36: Simulation of 8 and 32 bit based output feedback controllers



The actual performance seen in running these two output feedback controllers on the coupled tanks apparatus was again very similar to the simulated responses.

6.4 Summary

Some of the capabilities of the prototype have been demonstrated in the preceding 3 examples. The first two examples show some of the different types of analysis and design that can be performed using the package.

The final example demonstrates the advantages of having implementation level simulation of a controller. For this example it was seen a designer would need to use at min. an 8-bit microprocessor to implement the required state feedback controller. The output feedback controller could not be implemented on a 4 bit controller at all, as the coefficients of the observer feedback matrix are greater than the max. value available. To implement this output controller on a 4-bit microprocessor the entire problem would needed to be scaled to get the coefficients to fit into the number range.

The computation time for the coupled-tanks example was not a factor. The sample time of 5 sec. far exceeds to 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.

The code-generation facilities made it very easy to test implementations. The only drawback found was that to change an algorithm (say to use a different type of output feedback algorithm than the one used) a change in the prototype source code is needed. The code-generation facility needs to allow the user to enter the algorithm in a predefined mathematical form. Ideas on how to do this have been outlined in the previous chapter.

CHAPTER 7

CONCLUSIONS

The research outlined in this thesis derives a model for computer-aided control engineering. This model was derived by first looking at the general problem of engineering design, then looking at the particular situation of control engineering design. The model derived is similar in nature to model proposed by Rimvall [38] and Denham [28]. The main difference is that this model is considered to be ideally implemented within one package. A review of current state-of-the-art packages revealed that few attempted to encompass the entire design process from modelling to implementation. A summary of the packages reviewed is given in chapter 2 along with some current research to overcome their shortcomings.

A software architecture is then derived based on this CACE model. Functional architectural design and object-oriented architectural design was carried out to evaluate the best type of architecture. The object-oriented method was preferred because it resulted in an architecture that localises data structures and operations. This localising or modularising is much better than the functionally designed architecture which tended to leave much of the data structures global. These global structures would have made modifying the architecture difficult once it was implemented (i.e. would result in a large amount of code re-writing).

The object-oriented design focused on two objects, a *system diagram* and a *specification* object. The *system diagram* object was the key design decision for the architecture. This system diagram was implemented using a digraph representation. Details on how this digraph was implemented is given in chapter 4. The system diagram is composed of atomic components that can be interconnected. These atomic components have an *evaluation function* (EvFn) associated with them that dictates how they respond to stimuli. This idea of a *system diagram* helped to implement Maciejowski's idea of a *system data type*. By using the OOD method and *system diagram* object, this *system data type* could be consider implicitly implemented in the architecture. This implementation of the *system data type* is spread over the entire package rather than in one structure. This helped to remove the main criticism of the system data type concept i.e. that it was too inflexible.

Taking a bottom-up look at the design problem, control-oriented data-types were identified. These data types were needed for the architecture to overcome one of the key problems of many current packages - i.e. lack of appropriate data types. A list of these control-oriented data types and their required operations is given in chapter 4.

The software engineering requirements for developing a CACE package were defined. These requirements are listed in chapter 3 and they include readability, separate compilation, portability and data structures. The current popular languages in use were evaluated against this list of software requirements. Ada was selected as the most appropriate language for the implementation of a CACE package because of its encapsulation of many modern software engineering principles into its design. This included such things as packaging concepts, operator overloading, tasking and exception handling. For the future, C++ is seen as a possible alternative once commercial compilers become available for it.

A prototype package was developed that implemented this defined architecture. This prototype concentrated on developing the *system diagram* object through a menu driven interface. A control-based syntax was used for the input parser to allow the user to 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. This prototype was used for several design examples including a complete design exercise from modelling to implementation for a coupled-tanks apparatus. The simulation of the controller performance in the prototype corresponded closely to the actual performance achieved.

7.1 What was Achieved

A model for CACE was developed. An architecture, taking into account the software engineering aspects, to support this CACE model was developed using object-oriented design. This architecture is seen as a key to implementing a integrated environment for computer-aided control engineering. This architecture was implemented using Ada in a prototype package. This prototype package is the first developed using the object-oriented design method.

Software implementational issue that affect the prototype are highlighted and discussed. Tools written to aid the development of this prototype included a *grammar analyser and skeleton translator* called GAST. GAST translates an input LL(1) grammar file into an Ada programme that can parse inputs according the inputted LL(1) grammar.

Facilities to support the implementation of digital controllers have been designed into the architecture and have been included in the prototype. These facilities include simulation at the implementation level - i.e. simulation of the algorithm being performed on quantized inputs from A/Ds on a fixed word-length microprocessor, with the outputs being quantized by a D/As. Code-generation of a predefined selection of algorithms has been included in the prototype.

Also, as part of the development of the prototype, several generic packages were created. These packages included matrix, polynomial and transfer function manipulation facilities, tree, digraph and linked-list data structures creation and manipulation plus many mathematical algorithms such as pole placement and inverse nyquist array design. These packages on their own represent a significant body of useful routines or a toolbox of algorithms for control engineers. Thus, for future projects (particularly if using Ada), these routines can save a significant amount of time and effort. The idea is similiar to the use of the Fortran coded Eispack and Linpack libraries. Higher level routines can be developed from these basic packages in the fashion shown in Figure 14.

7.2 What was not Achieved

A full implementation of the architecture (called MSDI) was not performed. Only a selection of facilities in terms of specification of controller performance, design methods available and user defined algorithms for simulation and code-generation have been included. Complete facilities in these areas were considered to be beyond the scope of the current research. At various points in the thesis the facilities that are not included in the prototype are looked at and suggestions made on how they could be incorporated into the package.

7.3 Summary

The ideas investigated in this thesis have lead to the development of an architecture that can support the full control systems design process. The implementation of this architecture has been considered by looking at the software engineering aspects and by implementing the key ideas in a prototype package. Over time it is hoped that this prototype will evolve to encompass the complete MSDI architecture.

APPENDIX A

CACSD PACKAGE SURVEY

A.1 Lund Suite

These programmes represent a series of efforts at Lund Institute of Technology to develop CACSD tools. The initial projects date back to the early 1970s starting with a DEC PDP 15 with 32Kb of memory and 256 Kb disk. They were written in standard Fortran for portability.

The design goal for the *man-machine interface* (MMI) was to develop tools for the expert user rather than the novice which led to adopting a command dialogue interaction. This led the developers to command abbreviations and MACRO facilities to allowed for extendibility The MMI was called INTRAC

The data structures were derived from control theory i e. Matrices, polynomials, State Space descriptions, continuous and discrete time , etc.

Several individual packages were developed .

Package Name	Description
SIMNON	Simulation and optimisation of nonlinear continuous and discrete time systems
IDPAC	Data analysis, spectral analysis, correlation analysis and parameter estimation of linear MIMO models
POLPAC	Polynominal operations and designs
MODPAC	Transformation of models
DYMOLA	Symbolic development of models from basic equations
LISPID	Batch oriented estimation of parameters in linear stochastic systems

This suite is currently implemented on a VAX 11/780 with 2 M memory and 600 M disk Each package would need at a minimum of a PC with 1M of virtual storage and a floating point accelerator for acceptable performance

The main developers of the packages were Astrom, Wieslander, Elmqvist, Kallstrom and Ostberg Currently the packages have stabilised since the early 1980s

Comments

This suite probably represents the evolution of CACSD packages INTRAC was an excellent early attempt to move away from menu driven interfaces But its current lack of graphical inputs and inconsistencies in commands between packages and within packages is a problem. Simnon is a powerful tool but models need to be developed in a textual manner rather than in the more natural graphical form and batch oriented simulation for larger systems is not possible. The data-base facilities are poor.

A.2 MATLAB and its Children

MATLAB was released by Cleve Moler in 1980 and has grown to be among the most important developments in CACSD. It is a milestone in the history of interactive programmes with an easy-to-use, command-driven interface to Linpack and Eispack software and was called "the best piece of software available for 75 bucks" [6] Several packages have grown out of this initial package incorporating such things as control algorithms and graphical output. These include :

- PC-MATLAB and its associated toolboxes.
- PRO-MATLAB and its associated toolboxes
- CTRL-C
- MATRIXx
- IMPACT

PC and PRO- MATLAB come with optional toolboxes for control, identification and multivariable frequency design that cater for specific control problems The integrity of these toolboxes from a numerical viewpoint is due in large measure to the inclusion of people such as Alan J Laub and John N Little on the development teams.The systems are extendible with the use of M-files and script files The use of the comments in these files as part of the on-line help is very useful.

CTRL-C was developed from MATLAB with the goal to allow .

a. Easy matrix manipulation - basically from MATLAB
b. Uniform file handling - adding in UNIX-like notation
c. Direct manipulation of data - system transparent to user , nothing hidden
d. Extensibility - basically MATLAB Macros

All data is stored in a large stack. It includes both 2-D and 3-D graphics

In MATRIXx ,developed by Integrated Systems Inc , the main feature different from the MATLAB family is the facility provided by SYSTEM_BUILD that allows graphical development of systems by block diagrams plus the ability to use windowing Nonlinear components such as saturators, etc, can easily be included in the design Facilities for using Fortran code to define a subsystem are also included

IMPACT was developed by Rimvall at ETH Zurich [36] to help overcome the shortcomings of MATLAB in the areas of data structures available and to provide a powerful command language. It differs from the others also in that it is developed in Ada instead of Fortran It also goes a long way in removing the inconsistencies of syntax of MATLAB by using overloaded operators. Rimvall included a QUERY mode that would help novices to use the IMPACT system without being overwhelmed by its complexity

All of these systems can be used on a variety of systems from PCs to workstations The more advanced such as IMPACT and MATRIXx would need a workstation to derive the full benefits from their inbuilt facilities for graphics, user-interface, computations, etc

Comments

One major shortcoming of MATLAB and its children is the lack of appropriate data structures with only the complex matrix as the single basic data structure. An exception is IMPACT which supports a set of Control-Oriented data structures.

Another critical aspect common to these packages is the absence of an efficient database management system to help the user with the organisation of data and also to provide easier ways to interface different systems and software packages

Also the lack of operator overloading (except in IMPACT) leads to inconsistent use of symbols and operations (e.g. to multiply two polynomials, the command CONV is used into of and an in-fixed * operator)

Finally the lack of run-time performance of user-defined functions as the functions and macros are interpretive means there is often a large overhead associated with using them A method of "Compiling" them into the system to improve efficiency would be very useful.

A.3 DELIGHT

DELIGHT is an interactive CAD infrastructure originally developed for optimisation studies by Mayne (Imperial College) and Polak (Berkeley). It has operating system-like capabilities with string manipulation, I/O handling and file management It includes a graphics package and matrix package

Its command language is based on the language RATTLE Macros and aliasing of commands can be defined and used The command language is quite versatile but relatively unstructured It provides an editor to browse through a session's activities with a history/undo facility

The original control system definition utility used a Polish list method of system interconnection but this is being updated to a menu driven graphical input facility that would allow block diagrams to be edited and simulated

Most of the design algorithms available on the system are biased towards optimisation

Comments

The most impressive part of DELIGHT is the macro and aliasing facilities The macro features need to be made more robust to make them more consistent with the rest of the system. The upgrade to graphical input of systems is a step in the right direction There is no true database management system with the internal workspace being dumped into binary files (which makes them not portable) for saving the session.

A.4 CLADP

CLADP was developed in Cambridge University mainly by Maciejowski and Edmunds in the early 80s building on many of the ideas and theories of A G.J MacFarlane.

The package is written entirely in Fortran The User-Interface is mainly menu-driven with the driver being little more than a multi-position switch. Simple commands such as "NYQUIST" cause particular actions to be set in motion Extended command lines with arguments are not used.

A macro facility is implemented using files. Macros of CLADP commands are stored in files and executed by calling them from the keyboard. Macros can be called from within other macros. This is a simple but relatively effective for the command interface used on CLADP.

The main emphasis of the package is on frequency design and analysis techniques. The indicators of design information are usually present in frequency domain concepts such as stability (Nyquist/Bode), gain (Bode), etc. . The design algorithms available are very extensive in frequency domain. State space methods are available but no way as extensive.Matrix facilities are available that would allow the frequency domain and the state space methods to be extended and enlarged. Limited simulation facilities are implemented for the time domain. The simulation assumes a linear model is used (as assumed in all of CLADP). TSIM is usually used in conjunction with CLADP to increase the simulation capabilities. TSIM is a very powerful simulator widely used for both linear and non-linear simulation.

Comments

The user-interface of CLADP is generally poor compared to many other packages with much less flexibility for the engineer. The underlying numerical algorithms are very extensive and robust and particularly in the frequency domain are much more sophisticated than most other packages. Simulation is poor unless TSIM is used in conjunction with it.

A.5 Expert System Packages

CACE-III, the expert system designed by James et al. [1] at the Rensselaer Polytechnic Institute automatically designs a compensator for a given SISO plant, and provides a designer with little scope for the manipulating the design. Nolan [15], in contrast to this, has aimed at building an expert system which functions as a designer's assistant. He considers that the expert system should be able to carry out block diagram analysis of a given system, and provide the designer with assistance in selecting the type of compensation required. His type of expert system would link with a conventional control system CAD package which would provide the appropriate analysis and synthesis software required.

Birdwell [52], with his CASCADE package, is also working on developing an expert design package. His packages focuses on the LQG problem.

Comments

Most of these packages are focused on a very narrow section of the control system design problem. This is an area undergoing much research today but it will be some time before a commercial package addressing the full design cycle will be available.

A.6 Current Developments

The Environment for Control System Theory, Analysis and Synthesis, or ESCTACY, is a project funded by the SERC's Control and Instrumentation subcommittee to develop a new software infrastructure for CACSD. Its primary goals are :

a. To provide a common software base to assist the ready transfer of CAD tools between various academic groups.

b. To act as a means of transferring the control system design tools already developed in the academic world into industrial use in a consistent framework.

c. To provide a common software base for the development and testing of new design algorithms and facilities.

A report of the progress of this project is given in [35]. Basically it aims to provide a common operating environment and database facilities for control system design packages

CES, under development in the University of Wales, by Barker, Chen and Townsend [37] aims to provide a graphical environment for CACE. In provides for block-diagram and signal-flowgraph entry. It then can convert between the two forms. The internal representation used for both system models is a signal-flowgraph representation as it is more efficient.

Table 20 summaries the currently available packages that can be used at various stages of the control engineering design process

Table 20: CAD packages to support Design Process

Phase	Function	Package Names
Modelling	Symbolic Manipulation (Dynamic Equations)	Macsyma
	Schematic Capture	Matrix-x, Simbol
	Identification	IDPAC, CTRL-C, MATLAB[1]
	Non-Linear Dynamic Model	Tutsim, MATRIX-x, Simbol
	Linearisation	MATLAB[1], CTRL-C, MATRIX-x, Lund Suite, CLADP, Impact
	Model Reduction	MATLAB[1], CTRL-C, MATRIX-x, Lund Suite, CLADP, Impact
Specification	SISO design	CACE-III
Design	SISO / MIMO	MATLAB[1], CTRL-C, MATRIX-x, Lund Suite, CLADP, Impact
	Simulation - Linear	MATLAB[1], CTRL-C, MATRIX-x, Lund Suite, CLADP, Impact
	Simulation - Non-Linear	TSim2, Simbol, ACSL, PMSP, Simnon
	Non-Linear Techniques	CLADP, Lund Site
	Analysis	MATLAB[1], CTRL-C, MATRIX-x, Lund Suite, CLADP, Impact
Implementation	Simulation/Code Generation	MATRIX-x, Mascolt, RTS
	Documentation	Delight, CACE-III

[1] assuming appropriate toolbox

APPENDIX B

LANGUAGE EVALUATION

B.1 FORTRAN 77

Maintainability/Readability

Limited Data Abstraction Capabilities - Related to the limited data type problem which means that objects cannot be modelled in the most natural way but only in a quasi-numerical way.

No separate compilation facilities.

Modularity/Hierarchical

Not Modular - Does not lend itself easily to modular design and usually produces flat-designs with most data of the data being global, particularly with only pass by reference mechanisms.

Does not allow new operators to be defined to extend language constructs.

Not recursive

Data Structures/Types

Not Strongly Typed - Usually allows implicit declarations which lead to bugs caused by spelling mistakes which the compiler does not pick-up plus related use of loose control of mix-typing.

Limited Data Types - Only allows use of predefined types (real , integer, boolean, character) and the programmer cannot create new ones.

No List Structures - Which means that dynamic structures are very difficult to build and must be customised by the programmer.

Efficiency

Efficient - Fortran compilers tend to produce the most optimised code due to their limited constructs and their length of time in active use (since the sixties).

Concurrent Facilities

Not Concurrent - Only single thread of control catered for, causing problems in modelling solutions that are naturally concurrent.

Portability

Compilers - Most computer systems have available Fortran Compilers.
Porting - Different flavours for different computers, if Fortran 77 strictly used without the usually advanced features made available in various compilers, porting can be relatively easy.

Error Handling

No built-in facilities. This is usually handled by calls to the operating system.

I/O Facilities

Advanced I/O facilities are usually implementation dependent particularly for direct access files.

Familiarity

Very large body of programmes developed using Fortran, particularly numerical libraries such as Eispack and Linpack. Many well documented design methods with real design cases available.

GENERAL COMMENT

Fortran is often the first language that an engineer learns due to its ease of use in small programmes It has been used in many engineering packages over the years particularly in numerical software where designers were often concerned with efficiency (i.e Linpack / Eispack). These advantages in efficiency are not as important today with the improvements in hardware and compiler technology. The drawbacks make it unsuitable for large scale systems (though it still is used today because of the inbuilt inertia in many engineers to move to a "new" language)

B.2 PASCAL

Maintainability/Readability

No separate compilation facilities in standard Pascal (though available in many implementations)

Data Abstraction - Ability to create new types aids the maintainability and readability greatly

Modularity/Hierarchical

Procedural - Facilitates the breaking up of a design into modules, though generally tending towards a functional decomposition of a problem Most implementations of the language allow "external" procedures.

Recursive

Data Structures/Types

Typing - Enforces explicit declaration of all parameters used

Linked lists available

Poor string processing facilities

Efficiency

Efficient - Comparable to Fortran though usually not as highly optimised

Concurrent Facilities

Not Concurrent - Though lately "Concurrent Pascal" has been introduced to overcome this problem.

Portability

Again tends to be implementation dependent though not as much as Fortran.

Error Handling

None Available in the language.

I/O Facilities

Only Sequential File operations.

Familiarity

Many text books available with design methods and case studies of projects using Pascal.

GENERAL COMMENT

Pascal is good at teaching people structured design , containing some of the modern ideas on software engineering though not enough to make it truly useful. It was developed as a teaching aid, and in general this is the scope of its use , or in the development of small type applications (though its slightly difficult way of handling strings and arrays can cause difficulties in these areas compared to C). Many current implementations such as Turbo Pascal support additional facilities and provide such a good design environment that Pascal has become a popular modern language for small to medium size (> 10 K lines) engineering projects.

B.3 C and C ++

Maintainability/Readability

Very terse - Bad for Maintainability / Readability

Modularity/Hierarchical

Procedural - Allows for procedures and functions plus passing of variables by reference or value.

Not Extensible - Does not allow new operators to be created or operator overloading.

Independent Compilation of functions.

C++ has generic template facilities and does allow the creation of new operators and operator overloading.

Recursive

Data Structures/Types

Pointers - A very powerful feature , though often seems to be abused to the point of blurring structure and can cause subtle run-time problems if not carefully analysed/controlled.

Typing Optional - Allows for "hacking"

Coercion - Permits a variety of different types to coexist within a single expression implicitly doing the conversions.

C++ has inheritance which allows particular instantatioñš of a "generic" type to be made like say instant a MOSFET from a type FET.

Efficiency

Efficient - Though this can be implementation dependent

C++ today only has translators to standard C thus tends to be inefficient

Low Features - Allows a programme direct access to hardware if needed.

Concurrent Facilities

"Standard" C not concurrent.

C ++ concurrent

Portability

Very Portable - Standard C is powerful enough to cater for the needs of most apphcations without the need of implementation dependent features

Few C++ compilers/translators available.

Low Level Features - Need to be carefully used as can lead to porting problems

Error Handling

Not Available in C

Support in C++.

I/O Facilities

Standard Terminal I/O functionality.

File I/O - Sequential supported , others are implementation dependent

Familiarity

Some good textbooks available today for C, though not as many as for Fortran or Pascal C++ textbooks are very scarce

GENERAL COMMENT

C is a very powerful language that is often badly implemented (i e allowing too many implicit operations). It is probably the most popular "new" language in the engineering community This, more often than not, is because it facilitates "hacking". Its low level features are very powerful allowing a user direct access to hardware if required. C, in some respects, could be considered to be a "Portable Assembler" C++ overcomes many of the drawbacks of C but is mainly in an experimental stage, with usually only translators to C constructs available thus not maintaining the efficiency of C.

B.4 Ada

Maintainability/Readability

Very Verbose - leads to very readable programmes.

Supports Separate compilation - allows separation between specification and implementation.

Modern Software Concepts - Designed to incorporate and support many of these ideas

Modularity/Hierarchical

Modular - Programme built up through library units (i.e. packages, procedures, functions) which can all be compiled individually.

Data Abstraction - New types can be created to mirror the real life problem being modelled This includes restricting integers and real types to specific ranges.

Generics - Allow a programme to build up design solutions from software templates (e.g a Sort routine for anything - reals, integers, strings or arrays) thus allowing greater software reusability and brings the idea of software ICs closer.

Recursive

Data Structures/Types

Strongly Typed - All variables *must* be declared Also the compiler checks across module interfaces to ensure they are consistent.

Abstraction - Can create new types.

Attributes - Allow structures to be robust

Access Types - allow dynamical structures to be built up, though usually garbage collection (i e. releasing memory allocated automatically when no longer "pointed" to.)currently not implemented in current compilers

Low-Level Features - Allows direct access to hardware features.

Does not provide for inheritance

Does not have a procedure type - can lead to problems in system-type work

Efficiency

Strong typing and range check could lead to inefficient run-time performance in applications when compared to other languages.

Optional Turn-off of Error Checking - Lead to faster apphcations

Long Compilation times - due to cross-module interface checking

Concurrent Facilities

Tasking facilities available for concurrent programmes

Portability

Compilers - Currently limited no. of available compilers, though growing.

Porting - Rigidly defined in a standard and all compilers must be validated by the US Dept. of Defence (DoD).

Error Handling

Exception Handling - Error mechanisms implemented to control programme execution after the detection of an error condition.

User Definable - User can define his own errors and handling routines.

I/O Facilities

File Handling - Sequential and random access facilities.

Terminal I/O - Basic because idea is that user defines and builds up his own I/O packages.

Low-Level - An implementation defined package for primitive I/O.

Familiarity

Lack of Experience - It is a new language with few design examples to use as a guide in developing a programme in it and few performance details to see its implementation limitations.

Limited number of textbooks.

GENERAL COMMENT

Ada is designed to be used for large programmes. It was designed by an initiative of the DoD to help in the software crisis. Some of its constructs cause it to produce not as optimised code as Fortran compilers. The language encompasses a design environment where programmes are compiled into a library thus aiding in project management aspects. Its generic facilities are extremely important (though not as powerful as those implemented in C++). It lends itself to real-time environments with its low-level features and tasking constructs.

B.5 LISP / PROLOGUE

Maintainability/Readability

Poor, due to syntax mainly (e.g. Lisp's Setq (+(A,B))

Modularity/Hierarchical

Lisp very modular - declarative language

Prologue - Written in sentence - can be difficult to modularize

Different design philosophy required compared to other languages.

Data Structures/Types

Lisp - Based only on lists

Prologue - Allows multiple types to be built

Usually limited in precision thus not suitable for numerical work.

Efficiency

Both languages are highly computation intensive due to their list processing and backtracking mechanisms.

Very poor for numerical applications - slow.

Often Interpreted not compiled.

Inferencing Engines - Powerful inferencing mechanisms (though Prologue is often only implemented with backward chaining paradigm)

Concurrent Facilities

Not available

Portability

Compilers - Many different compilers available for different computer systems

Porting - Very difficult due to the many different flavours or dialects of each language.

Error Handling

Not available

I/O Facilities

Poor on their own

Familiarity

Lisp is an "old" language with many textbooks and application studies Prologue is newer and is not as well documented and "teased" out in design studies

GENERAL COMMENT

Lisp and Prologue are really AI languages (though people have used them for general applications) which allow the encapsulation of knowledge and deduction of facts from this knowledge through their inferencing mechanisms They tend to be very inefficient, in that current hardware topologies are not oriented to support their type of operation easily They are often used in the implementation of expert systems.

APPENDIX C

FUTURE TRENDS IN DESIGN COMPLEXITY AND TECHNOLOGY

C.1 Complexity Trends

The Control Engineering design problem, like nearly all facets of engineering, is growing in size and complexity. The 1960s and 1970s saw industrial applications mainly focus on analog implementations of the 3-term controller. The conventional control system for a mechanism or industrial process incorporated (many still do incorporate) many small units of pneumatic and/or analogue electronic controllers. Each of these usually was dedicated to a particular task, with little communication with other units.

The micro-processor has revolutionised the architecture of control systems. Microprocessors now often substitute for the conventional electronic controllers. Such units may control a single loop or a number of loops simultaneously. The control function itself is implemented in software while the hardware functions of these controllers are mainly restricted to the conversions between analogue and digital signals.

During the 80s there has been a gradual increase in the acceptance of more complex control strategies such as optimal design. These methods require the power of both sophisticated CAD systems to aid the designer to effectively create the design and powerful, reliable microprocessor based systems to implement them. The reduction in the price of microprocessor hardware and their increased reliability plus the need for increased performance in more complex design problems are the reasons for their gradual acceptance in industry. The leaders in the use of this type of technology have been, as in many other areas, the military and space agencies. Many current aircraft have sophisticated control systems which the pilot would be unable to fly the plane without [53]. Robotics is another area where new techniques are being investigated as a means of increasing performance.

The complexity that these systems demand is ever increasing, e.g. tighter specifications of performance, optimisation of controller and plant parameters in the face of multiple design objectives. Instead of single independent loops, systems today are tending towards tightly coupled loops to maximise performance. Also the needs for reliability, particularly in safety-related areas, such as auto-pilot systems, demand performance far greater than those of the 1970s. These considerations must be factored into the design process to ensure that a satisfactory controller meeting the design objectives can be achieved.

C.2 Technology Trends

Technology has rapidly changed in the last decade and will continue for the foreseeable future. We have moved from very expensive mainframe computers to the inexpensive PCs and workstations of today. Already research and design teams are working on such products as application oriented workstations (for AI, CAD, etc.) plus systems that combine parallel processing and graphic technology into a workstation (e.g. Stellar). RISC will attempt to

dislodge CISC technology in microprocessors. The areas of development in which we are likely to see major technological advances are :

- *Miniaturisation of electronics* - One of the greatest forces in change in the future as it makes designs both faster and cheaper.
- *Input technology / Human Interfaces* - Both the specific input devices and the screen "look" are areas in which developments are still forthcoming The availability of every decreasing in-price hardware has opened up new possibilities in input devices. Artificial intelligence and graphics should prompt developments in the Human Interface area
- *Handling of Image Information* - The incorporation of non-coded information into databases is an important direction in computing Specifically, the ability to capture images of the physical surroundings efficiently and reliably in computerised form is being actively researched The increasing availability of vision technology and the cheapness of storage open up opportunities in this area.
- *Parallelism* - The increasing use of parallel architecture is being pushed to increase price/performance, as well as to achieve the greatest performance from a computer
- *System Software* - The key advances have been and will continue to be , the crystallising of interfaces (both human and inter-program), the use of higher levels of abstraction and greater subsystem functionality. The availability of cheap hardware is permitting, if not driving, the trend toward increasingly complex applications

APPENDIX D

LOW LEVEL OBJECT CODE EXAMPLES

Figure 37: Generic Matrix Package

```
--
-- PACKAGE MATRICE  ( Specification )
--
--
-- Version    1-002
--
-- Facility    User Numerical Computation Library
--
--
-- Abstract : This generic package provides all the elementary
--     operations required for matrix computations  The exceptions
--      defined in package MATRIX_EXCEPTION are imported for use in
--     the body of this package.
--
--
-- Author      John Hickey      Creation Date · 01/02/87
--

with TEXT_IO, use TEXT_IO,
--

generic

 type ELEMENT is private;
 --+
 --+ ELEMENT defines the type the individual components used to
 --+ construct the matrix
 --+

 ELEMENT_ZERO   ELEMENT;
 --+
 --+ ELEMENT_ZERO defines the null element value for the generic
 --+ type.
 --+

 ELEMENT_ONE    ELEMENT,
 --+
 --+ ELEMENT_ONE defines the Identity element value for the generic
 --+ type i e  multiply an element by this value and the result is the
 --+ same element.
 --+
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
 DEFAULT_FORE : TEXT_IO.FIELD;
 DEFAULT_AFT : TEXT_IO.FIELD;
 DEFAULT_EXP : TEXT_IO.FIELD;
 --+
 --+ Define the default format for the Get and Put Element routines.
 --+

 with function "+"( LEFT, RIGHT : in ELEMENT ) return ELEMENT is <>;
 with function "-"( LEFT, RIGHT : in ELEMENT ) return ELEMENT is <>;
 with function "*"( LEFT, RIGHT : in ELEMENT ) return ELEMENT is <>;
 with function "/"( LEFT, RIGHT : in ELEMENT ) return ELEMENT is <>;
 with function ">"( LEFT, RIGHT : in ELEMENT ) return BOOLEAN is <>;
 with function "**"( LEFT : in ELEMENT; RIGHT : in INTEGER ) return ELEMENT is <>;
 with function "ABS"( X : in ELEMENT ) return ELEMENT is <>;
 with function COS( X : in ELEMENT ) return ELEMENT is <>;
 with function SIN( X : in ELEMENT ) return ELEMENT is <>;
 with function TAN( X : in ELEMENT ) return ELEMENT is <>;
 with function   "-"( X : in ELEMENT ) return ELEMENT is <>;
 --+
 --+  These procedure define the addition, subtraction, multiplication
 --+ and division, relational, uniary and absolute value properties for
 --+ ELEMENT types needed for the package.
 --+

 with procedure GET( X : out ELEMENT; WIDTH : in TEXT_IO.FIELD := 0 ) is <>;
 with procedure GET( FILE : in TEXT_IO.FILE_TYPE; X : out ELEMENT; WIDTH : in TEXT_IO.FIELD

 with procedure PUT( X : in ELEMENT; FORE : in TEXT_IO.FIELD := 2; AFT : in TEXT_IO.FIELD :
     EXP : in TEXT_IO.FIELD := 3 ) is <>;
 with procedure PUT( FILE : in TEXT_IO.FILE_TYPE; X : in ELEMENT; FORE : in TEXT_IO.FIELD :
    AFT : in TEXT_IO.FIELD := DEFAULT_AFT; EXP : in TEXT_IO.FIELD := DEFAULT_EXP ) is <>;
 --+
 --+ Procedures Get and Put define I/O operations for ELEMENT to the
 --+ terminal and a file.
 --+
package GEN_MATRICE is


 type MATRIX is array ( INTEGER range <>, INTEGER range <> ) of ELEMENT;
 --+
 --+  MATRIX defines the type supported by this package. It consists
 --+ of a two dimensional array of the generic type ELEMENT.
 --+

 --
 -- Input/Output Routines
 --
 procedure GET( A : out MATRIX         );
 procedure GET( FILE : in TEXT_IO.FILE_TYPE; A : out MATRIX );

 procedure PUT( A : in MATRIX         );
 procedure PUT( FILE : in TEXT_IO.FILE_TYPE; A : in MATRIX );


 --
 -- Addition and Subtraction Routines
 -- ERROR : Non_Conformable raised if not mathematically defined
 --  operation for Left/Right parameters used.
 --
 function "+"( LEFT,RIGHT : in MATRIX   ) return MATRIX;
 function "-"( LEFT,RIGHT : in MATRIX   ) return MATRIX;
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
-- Unıary Functıon
functıon "-"( X : ın MATRIX   ) return MATRIX;

--
-- Scalar and Matrıx Multıplıcatıon
-- ERROR · Non_Conformable raıse ıf not mathematıcally defıned
--  operatıon for Left/Rıght parameters used.
--
functıon "*"( LEFT,RIGHT · ın MATRIX ) return MATRIX;
functıon "*"( LEFT,RIGHT   ın MATRIX ) return ELEMENT,
functıon "*"( LEFT . ın ELEMENT, RIGHT : ın MATRIX  ) return MATRIX;
functıon "*"( LEFT · ın MATRIX;  RIGHT   ın ELEMENT ) return MATRIX,

--  Matrıx Exponent
functıon "**"( LEFT   ın MATRIX, RIGHT   ın NATURAL ) return MATRIX,

--
-- Transpose of Matrıx
--
functıon TRANS ( A   ın MATRIX   ) return MATRIX;

--
--  Matrıx Inversıon
--  ERROR   DIMENSION_ERROR raıse ıf not a square matrıx
--     SINGULAR raıse when the matrıx ıs sıngular ı e  no ınverse
--
functıon INV( A   ın MATRIX ) return MATRIX,

-- Matrıx Trıgnometrıc functıons
functıon COS( A   ın MATRIX ) return MATRIX;
functıon SIN( A   ın MATRIX ) return MATRIX,
functıon TAN( A   ın MATRIX ) return MATRIX;

--
-- Multıplıcatıon by Inverse of Rıght
--
functıon "/"( LEFT,RIGHT   ın MATRIX ) return MATRIX,
functıon "/"( LEFT   ın MATRIX; RIGHT   ın ELEMENT ) return MATRIX,

--
-- Matrıx Utılıtıes
--
functıon IDENTITY ( SIZE   ın NATURAL ) return MATRIX,    -- Form Identıty matrıx
functıon COL_NORM ( A      ın MATRIX  ) return ELEMENT;  -- Compute the column norm of A
functıon ROW_NORM ( A      ın MATRIX  ) return ELEMENT,   -- Compute the row norm of A
functıon TRACE    ( A      ın MATRIX  ) return ELEMENT,  -- Computes the trce of the matrı
functıon IS_ZERO ( A · ın MATRIX ) return BOOLEAN,  -- Returns True ıf Matrıx ıs all zeros

end GEN_MATRICE,
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
--
with INTEGER_TEXT_IO, TEXT_IO, MATRIX_EXCEPTION;
use  INTEGER_TEXT_IO, TEXT_IO, MATRIX_EXCEPTION,
--

package body GEN_MATRICE is


------------------------------------------------------
---    Specification for the linear solver package   ----
---                                                  ----
---                   L  U  _  B  A  C  K            ----
---                   ~~~~~~~~~~~~~~~~~~~            ----
------------------------------------------------------

package LU_BACK is

 type COL_VECTOR is array ( INTEGER range <> ) of ELEMENT;
 type VECINT is array ( INTEGER range <> ) of INTEGER,
 --+
 --+ These types are needed for the Inverse routine using
 --+ package LU_BACK
 --+

 procedure DEC (  n    in INTEGER,
                       ndim    in INTEGER,
                          a    in out MATRIX,
                         ip    in out VECINT,
                        ier    in out INTEGER ),

 procedure SOL (  n : in INTEGER,
                       ndim    in INTEGER,
                          a    in MATRIX,
                          b    in out COL_VECTOR,
                         ip    in VECINT  ),


end LU_BACK,
------------------------------------------------------------
----       BODY  of   lu_back   is as follows           ----
------------------------------------------------------------

package body LU_BACK is

 one    constant ELEMENT  = ELEMENT_ONE,
 zero   constant ELEMENT  = ELEMENT_ZERO;


procedure DEC ( n     in INTEGER,  ndim     in INTEGER,
  a     in out MATRIX, ip     in out VECINT,
  ier     in out INTEGER       ) is

      tstar     ELEMENT,
      m, nm1, k, kp1, i . INTEGER,

begin
 ier := 0;
 ip(n)  = 1,
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
 if n > 1 then
  nm1 := n-1;
  for k in 1..nm1 loop
     kp1 .= k + 1,
   m   .= k;

   for i in kp1 .n loop
    if abs(a(i,k)) > abs(a(m,k)) then
     m := i;
    end if;
   end loop;

      ip(k) := m;
   tstar  = a(m,k);

   if m /= k then
    ip(n)  = - ip(n),
    a(m,k) := a(k,k);
    a(k,k)  = tstar,
   end if;

   if tstar = zero then
    ier := k,
    ip(n) := 0,
    return,
   end if,

   tstar  = one/tstar,

   for i in kp1..n loop
    a(i,k) := - a(i,k)*tstar;
   end loop;

   for j in kp1..n loop
    tstar := a(m,j),
       a(m,j) .= a(k,j),
    a(k,j)  = tstar;
    if tstar /= zero then
        for i in kp1  n loop
         a(i,j)  = a(i,j) + a(i,k)*tstar,
        end loop,
    end if,
   end loop;
  end loop,

 end if,
--
 k .= n,

 if a(n,n) = zero then
  ier  = k,
  ip(n)  = 0;
 end if,
--
end DEC,
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
-----------------------------------------------------------------
-----          Procedure                S   O   L          ------
-----                                   ~~~~~~~~~~         ------
-----------------------------------------------------------------
procedure SOL ( n  : in INTEGER;
  ndim  : in INTEGER,
  a  . in MATRIX;
  b : in out COL_VECTOR,
  ip  : in VECINT)    is

 tstar : ELEMENT,
 i, k, kb, km1, kp1, m, nm1 · INTEGER;

begin
 if n /= 1 then
  nm1 := n - 1,

  for k in 1  nm1 loop
      kp1 ·= k + 1,
      m ·= ip(k),
      tstar .= b(m),
      b(m) := b(k),
      b(k)  = tstar;
      for i in kp1  n loop
   b(i)  = b(i) + a(i,k)*tstar;
      end loop,
  end loop;

  for kb in 1  nm1 loop
      km1 := n - kb,
      k  = km1 + 1,
      b(k) := b(k)/a(k,k),
      tstar  = - b(k),
      for i in 1  km1 loop
   b(i)  = b(i) + a(i,k)*tstar,
      end loop,
  end loop;

    end if;
    b(1)  = b(1)/a(1,1),

 return;

 end SOL,

end LU_BACK,
--
-- This procedure allows elements of an array to inputed
--
procedure GET ( A   out MATRIX ) is
begin
 for I in A'RANGE(1) loop
  put("Enter Row "),put(I),put("  "),
  for J in A'RANGE(2) loop
   new_line,
   put(" > "),GET ( A(I,J) ),
  end loop,
 end loop;
end GET;
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
--
-- This procedure gets the matrix from a file
--
procedure GET ( FILE . in FILE_TYPE, A : out MATRIX ) is
begin
 for I in A'RANGE(1) loop
  skip_line( FILE );
  for J in A'RANGE(2) loop
   GET ( FILE , A(I,J) );
  end loop;
 end loop,
end GET,

--
-- This procedure Outputs the contents of a matrix
--
procedure PUT ( A in MATRIX ) is
begin
 for I in A'RANGE(1) loop
  new_line,
  for J in A'RANGE(2) loop
   PUT("    ");PUT( A(I,J) ), PUT(" "),
  end loop;
 end loop;
 new_line,
end PUT,

--
-- This procedure Outputs the contents of a matrix
--
procedure PUT ( FILE in FILE_TYPE; A . in MATRIX ) is
begin
 for I in A'RANGE(1) loop
  new_line( FILE ),
  for J in A'RANGE(2) loop
   PUT( FILE, A(I,J) ),
  end loop,
 end loop,
end PUT,
--
-- This function returns the sum of two matrices.
-- ERROR   Non_Conformable raised if not mathematically defined
--  operation for Left/Right parameters used
--
function "+"( LEFT, RIGHT   in MATRIX ) return MATRIX is
 C    MATRIX ( LEFT'RANGE(1),LEFT'RANGE(2) ),
begin
 if ((LEFT'LENGTH(1)/=RIGHT'LENGTH(1)) or (LEFT'LENGTH(2)/=RIGHT'LENGTH(2))) then
  raise NON_CONFORMABLE,  -- dimensional error exception ---
 else
  for ROW in LEFT'RANGE(1) loop
   for COL in LEFT'RANGE(2) loop
    C(ROW,COL)  = LEFT(ROW,COL) + RIGHT(ROW,COL),
   end loop,
  end loop;
  return C;
 end if,
end "+",
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
--
-- This function returns the difference of two matrices.
-- ERROR . Non_Conformable raised if not mathematically defined
--  operation for Left/Right parameters used.
--
function "-"( LEFT, RIGHT   in MATRIX ) return MATRIX is
 C  . MATRIX ( LEFT'RANGE(1),RIGHT'RANGE(2) ),
begin
 if ((LEFT'LENGTH(1)/=RIGHT'LENGTH(1)) or (LEFT'LENGTH(2)/=RIGHT'LENGTH(2))) then
  raise NON_CONFORMABLE,  -- dimensional error exception ---
 else
  for ROW in LEFT'RANGE(1) loop
   for COL in LEFT'RANGE(2) loop
    C(ROW,COL)  = LEFT(ROW,COL) - RIGHT(ROW,COL),
   end loop,
  end loop;
  return C,
 end if,
end "-";

-- Unary Sub  function
function "-"( X   in MATRIX ) return MATRIX is
 ANS    MATRIX ( X'range(1), X'range(2) ),
begin
 for ROW in X'range(1) loop
  for COL in X'range(2) loop
   ANS(ROW,COL)  = - X( ROW,COL ),
  end loop,
 end loop,
 return ANS;
end "-";

--
-- This function returns the product of two matrices
-- ERROR   Non_Conformable raised if not mathematically defined
--  operation for Left/Right parameters used
--
function "*"( LEFT, RIGHT   in MATRIX ) return MATRIX is
  C    MATRIX ( LEFT'RANGE(1),RIGHT'RANGE(2) );
  INDEX     INTEGER,
  TOTAL     ELEMENT  := ELEMENT_ZERO,
begin
 if ( LEFT'LENGTH(2) /= RIGHT'LENGTH(1) ) then
  raise NON_CONFORMABLE;  -- dimensional error exception ---
 else
  for ROW in LEFT'RANGE(1) loop
   for COL in RIGHT'RANGE(2) loop
    TOTAL  = ELEMENT_ZERO,
    for INDEX in LEFT'RANGE(2) loop
     TOTAL  = TOTAL + LEFT(ROW,INDEX)*RIGHT(INDEX,COL),
    end loop,
    C(ROW,COL) .= TOTAL;
   end loop,
  end loop,
  return C,
 end if,
end "*";
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
--
-- This function returns the product of two matrices as an Element type.
-- ERROR   Non_Conformable raised if not mathematically defined
--  operation for Left/Right parameters used.
--
function "*"( LEFT, RIGHT . in MATRIX ) return ELEMENT is
 A   MATRIX(1. 1,1 .1),
begin
 if ( LEFT'LENGTH(1) /= 1 and RIGHT'LENGTH(2) /= 1 ) then
  raise NON_CONFORMABLE,  -- dimensional error exception ---
 else
  A := LEFT * RIGHT,
  return A(1,1),
 end if;
end "*",

--
-- Scalar Multiplication of a Matrix on left.
--
function "*"(LEFT   in ELEMENT, RIGHT   in MATRIX) return MATRIX is
 LEFTRIGHT    MATRIX (RIGHT'RANGE(1),RIGHT'RANGE(2)),
begin
 for ROW in RIGHT'RANGE(1) loop
  for COL in RIGHT'RANGE(2) loop
   LEFTRIGHT(ROW,COL)  = LEFT*RIGHT(ROW,COL);
  end loop,
 end loop,
 return LEFTRIGHT,
end "*",

--
-- Scalar Multiplication of a Matrix on right
--
function "*"(LEFT   in MATRIX, RIGHT   in ELEMENT ) return MATRIX is
 LEFTRIGHT    MATRIX ( LEFT'RANGE(1), LEFT'RANGE(2) ),
begin
 for ROW in LEFT'RANGE(1) loop
  for COL in LEFT'RANGE(2) loop
   LEFTRIGHT(ROW,COL)  = RIGHT*LEFT(ROW,COL);
  end loop;
 end loop,
 return LEFTRIGHT,
end "*",
--
--  Transposition of Matrix
--
function TRANS ( A   in MATRIX ) return MATRIX is
 C : MATRIX (A'RANGE(2),A'RANGE(1)),
begin
 for ROW in C'RANGE(1) loop
  for COL in C'RANGE(2) loop
   C(ROW,COL)  = A(COL,ROW),
  end loop,
 end loop,
 return C,
end TRANS,
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
--
--  This is a function to compute the Inverse of a Matrix using
--  LU decomposiiton.
--
--  ERROR : DIMENSION_ERROR raised if not a square matrix
--     SINGULAR raised when the matrix is singular i e. no inverse
--
function INV( A : in MATRIX ) return MATRIX is

 use LU_BACK;

 TEMP  : MATRIX( A'RANGE(1),A'RANGE(2) );
 ANS   MATRIX( A'RANGE(1), A'RANGE(2) );
 B    COL_VECTOR( A'RANGE(1) ),
 IP   . VECINT( A'RANGE(1) ),
 IER   . INTEGER,
 N   INTEGER    = A'LAST(1),

begin
 if A'length(1) /= A'length(2) then
  raise DIMENSION_ERROR,
 else
   for COL in A'range(1) loop

   TEMP .= A;

   for I in A'range(1) loop
    if I = COL then
     B(I) := ELEMENT_ONE,
    else
     B(I)  = ELEMENT_ZERO, -- Columns of I matrix
    end if,
   end loop,

   LU_BACK.DEC( N,N,TEMP,IP,IER),
   LU_BACK.SOL( N,N,TEMP,B,IP  ),

   for I in A'range(1) loop
    ANS(I,COL)  = B(I),
   end loop,
  end loop,

  return ANS,

 end if,

 exception
  when NUMERIC_ERROR => raise SINGULAR;

end INV,
-- Matrix Trigonometric Functions
function COS ( A   in MATRIX ) return MATRIX is
 ANS   MATRIX( A'range(1), A'range(2) ),
begin
 for I in A'range(1) loop
  for J in A'Range(2) loop
   ANS(I,J) .= COS( A(I,J) ),
  end loop;
 end loop,

 return ANS;
end COS,
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
function SIN ( A : in MATRIX ) return MATRIX is
 ANS . MATRIX( A'range(1), A'range(2) ),
begin
 for I in A'range(1) loop
  for J in A'Range(2) loop
   ANS(I,J)  = SIN( A(I,J) );
  end loop;
 end loop,

 return ANS,
end SIN,

function TAN ( A · in MATRIX ) return MATRIX is
 ANS   MATRIX( A'range(1), A'range(2) ),
begin
 for I in A'range(1) loop
  for J in A'Range(2) loop
   ANS(I,J)  = TAN( A(I,J) ),
  end loop,
 end loop,

 return ANS;
end TAN,
--
-- Multiplication by Inverse of Right.
-- ERROR · NON_CONFORMABLE raised when matrices cannot be multiplied
--  SINGULAR raised when Right parameter is singular
--
function "/"( LEFT, RIGHT   in MATRIX ) return MATRIX is

 C     MATRIX ( LEFT'range(1), RIGHT'range(2) ),
 INV_RIGHT   MATRIX ( RIGHT'range(1), RIGHT'range(2) );
begin

 if ( RIGHT'length(1) = 1 and RIGHT'length(2) = 1 ) and  -- Scalar Division
   ( LEFT'length(2) = 1  ) then
  for ROW in LEFT'range(1) loop
   C(ROW,1) .= LEFT(ROW,1) / RIGHT(1,1),
  end loop;
  return C,
 else
  if ( LEFT'length(2) /= RIGHT'length(1) ) then
   raise NON_CONFORMABLE,  -- dimensional error exception ---
  else
   INV_RIGHT := INV ( RIGHT ),
   return ( LEFT * INV_RIGHT ),
  end if,
 end if,

end "/",


function "/"( LEFT   in MATRIX, RIGHT   in ELEMENT ) return MATRIX is
 ANS   . MATRIX ( LEFT'range(1), LEFT'range(2) ),
begin

 for I in LEFT'range(1) loop
  for J in LEFT'range(2) loop
   ANS(I,J) ·= LEFT(I,J) / RIGHT,
  end loop;
 end loop;
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
 return ANS;

end "/";
function "**"( LEFT : in MATRIX, RIGHT . in NATURAL ) return MATRIX is
--=====================================================================
-- This function computes the matrix exponent  LEFT must be a square
-- matrix. If not Dimension_Error is raised.
--=====================================================================
 TEMP . MATRIX( LEFT'range(1), LEFT'range(1) ) := IDENTITY( LEFT'length(1) );
begin
 if LEFT'length(1) /= LEFT'length(2) then
  raise DIMENSION_ERROR,
 else
  if RIGHT = 0 then
   null,
  else
   for POWER in 1 .RIGHT loop
    TEMP .= TEMP * LEFT,
   end loop;
  end if,

  return TEMP;
 end if;

end "**",


function IDENTITY ( SIZE   in NATURAL ) return MATRIX is
--=====================================================================
-- Function Identity returns an Identity matrix of dimension SIZE.
--=====================================================================
 IDENT   MATRIX( 1  SIZE, 1 .SIZE)  = ( others => ( others => ELEMENT_ZERO ) ),
begin
 for I in 1 .SIZE loop   -- Set Diagonal Elements
  IDENT(I,I)  = ELEMENT_ONE, -- to be Identity ELement.
 end loop,

 return IDENT,
end IDENTITY,


function COL_NORM ( A   in MATRIX ) return ELEMENT is
--=====================================================================
-- Function COL_NORM computes the column norm of the matrix A, i.e  the
-- max  value of the sum of the absolute values of an individual column
--=====================================================================
 MAX_COL_NORM   ELEMENT   = ELEMENT_ZERO;
 COL_NORM · ELEMENT,
begin
 for J in A'range(2) loop
  COL_NORM  = ELEMENT_ZERO,
  for I in A'range(1) loop -- Add up column elements Absolute values.
   COL_NORM  = COL_NORM + ABS( A(I,J ) );
  end loop,

  if COL_NORM > MAX_COL_NORM then  -- See if current column norm is largest so far
   MAX_COL_NORM  = COL_NORM; -- If so record it.
  end if,
 end loop;

 return MAX_COL_NORM;

end COL_NORM;
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
function ROW_NORM ( A   in MATRIX ) return ELEMENT is
--=========================================================================
-- Function ROW_NORM computes the row norm of the matrix A, i.e  the
-- max  value of the sum of the absolute values of an individual row.
--=========================================================================
 MAX_ROW_NORM : ELEMENT  := ELEMENT_ZERO;
 ROW_NORM   ELEMENT;
begin
 for I in A'range(1) loop
  ROW_NORM := ELEMENT_ZERO;
  for J in A'range(2) loop -- Add up row elements Absolute values
   ROW_NORM := ROW_NORM + ABS( A(I,J ) );
  end loop;

  if ROW_NORM > MAX_ROW_NORM then  -- See if current row norm is largest so far
   MAX_ROW_NORM  = ROW_NORM; -- If so record it
  end if;
 end loop;

 return MAX_ROW_NORM;

end ROW_NORM;

function TRACE ( A   in MATRIX ) return ELEMENT is
--=========================================================================
-- Returns the trace of matrix , i e. sum of the principal diagonal
-- values
--=========================================================================
 TRACE_VALUE   ELEMENT  = ELEMENT_ZERO;
begin
 for I in A'range(1) loop
  TRACE_VALUE := TRACE_VALUE + A(I,I);
 end loop;

 return TRACE_VALUE;

 exception
  when CONSTRAINT_ERROR =>
   raise DIMENSION_ERROR;

end TRACE;


function IS_ZERO( A : in MATRIX ) return BOOLEAN is
--=========================================================================
-- This procedure checks if each element in A is ELEMENT_ZERO and
-- returns True if it is, and false if not
--=========================================================================
begin
 for I in A'range(1) loop
  for J in A'range(2) loop
   if A(I,J) /= ELEMENT_ZERO then
    return FALSE;
   end if;
  end loop;
 end loop;

 return TRUE;

end IS_ZERO;
```

Figure 37 Cont'd. on next page

Figure 37 (Cont.): Generic Matrix Package

```
end GEN_MATRICE;
--+
```

Figure 38: Generic Complex Package

```
--
-- FACILITY
--   GEN_COMPLEX    ( Specification )
--
-- ABSTRACT
--
-- This generic package implements the type Complex and its required functions
-- Also overloaded Text_IO functions are provided.
--
--
-- AUTHOR
--   John Hickey
--
--
with TEXT_IO,
--

generic


 type ELEMENT is digits <>,
  --+ Defines the type to be used as the basis of the complex no.

 type ELEMENT_MATRIX is array ( INTEGER range <>, INTEGER range <> ) of ELEMENT,
  --+ Defines a matrix for Element values


 with procedure GET( X   out ELEMENT, WIDTH   TEXT_IO.FIELD := 0 ) is <>,
 with procedure GET( FILE   in TEXT_IO.FILE_TYPE, X   out ELEMENT, WIDTH   TEXT_IO FIELD  =
 with procedure PUT( X   in ELEMENT, FORE   in TEXT_IO FIELD  = 2, AFT : TEXT_IO FIELD  = 5
     EXP   TEXT_IO FIELD := 3 ) is <>,
 with procedure PUT( FILE   TEXT_IO FILE_TYPE, X   in ELEMENT, FORE   in TEXT_IO FIELD  = 2
     EXP   TEXT_IO FIELD  = 3 ) is <>,
 --+
 --+ Procedures Get and Put define I/O operations for ELEMENT to the
 --+ terminal and a file

 with function SQRT ( X   in ELEMENT ) return ELEMENT is <>;
 with function ATAN ( X   in ELEMENT ) return ELEMENT is <>;
 --+ Mathematical functions of Element used within the package

-----------------------
package GEN_COMPLEX is
-----------------------


 type COMPLEX is private,
  --+
  --+  Complex Number Data type
  --+
```

Figure 38 Cont'd. on next page

Figure 38 (Cont.): Generic Complex Package

```
type COMPLEX_MATRIX is array ( INTEGER range <>, INTEGER range <> ) of COMPLEX;
 --+
 --+ Complex Matrix
 --+

type COMPLEX_VECTOR is array ( INTEGER range <> ) of COMPLEX;
 --+
 --+ Complex Vector
 --+

--
--  Addition Property
--
function "+"( LEFT, RIGHT : COMPLEX ) return COMPLEX;
function "+"( LEFT : ELEMENT; RIGHT : COMPLEX ) return COMPLEX;
function "+"( LEFT : COMPLEX; RIGHT : ELEMENT ) return COMPLEX;

--
--  Subtraction Property
--
function "-"( LEFT, RIGHT : COMPLEX ) return COMPLEX;
function "-"( LEFT : ELEMENT; RIGHT : COMPLEX ) return COMPLEX;
function "-"( LEFT : COMPLEX; RIGHT : ELEMENT ) return COMPLEX;

--
--  Multiplication Property
--
function "*"( LEFT, RIGHT : COMPLEX) return COMPLEX;
function "*"( LEFT : ELEMENT; RIGHT : COMPLEX ) return COMPLEX;
function "*"( LEFT : COMPLEX; RIGHT : ELEMENT ) return COMPLEX;

--
--  Division Property
--
function "/"( LEFT, RIGHT : COMPLEX ) return COMPLEX;
function "/"( LEFT : ELEMENT; RIGHT : COMPLEX ) return COMPLEX;
function "/"( LEFT : COMPLEX; RIGHT : ELEMENT ) return COMPLEX;

--
--  Complex Conjugate, Extract Real/Imaginary Part, Length
--
function CONJ ( NUM : COMPLEX ) return COMPLEX;
function RE ( NUM : COMPLEX ) return ELEMENT;
function RE ( NUM : COMPLEX_MATRIX ) return ELEMENT_MATRIX;
function IM ( NUM : COMPLEX ) return ELEMENT;
function IM ( NUM : COMPLEX_MATRIX ) return ELEMENT_MATRIX;
function NORM ( NUM : COMPLEX ) return ELEMENT;

function MAGNITUDE( NUM : in COMPLEX ) return ELEMENT;
function PHASE ( NUM : in COMPLEX ) return ELEMENT;

function MAGNITUDE( NUM : in COMPLEX_MATRIX ) return ELEMENT_MATRIX;
function PHASE ( NUM : in COMPLEX_MATRIX ) return ELEMENT_MATRIX;

function TRANS ( A : in COMPLEX_MATRIX ) return COMPLEX_MATRIX;

--
--  Form Complex type from Real and Imaginary parts
--
function FORM_COMPLEX ( RE , IMG : in ELEMENT ) return COMPLEX;
function FORM_COMPLEX ( RE : in ELEMENT ) return COMPLEX;
```

Figure 38 Cont'd. on next page

Figure 38 (Cont.): Generic Complex Package

```
 --
 --  Input_Output Functions
 --
 procedure GET( NUM : out COMPLEX );
 procedure PUT( NUM : in  COMPLEX );
 procedure PUT_LINE ( NUM : in  COMPLEX );

 procedure GET( MAT : out COMPLEX_MATRIX );
 procedure PUT( MAT : in  COMPLEX_MATRIX );

 procedure GET( MAT : out COMPLEX_VECTOR );
 procedure PUT( MAT : in  COMPLEX_VECTOR );

--
--  Define Private Type Complex for Complex Number
--      Representation
--

 private

  type COMPLEX is
   record
    RE : ELEMENT := 0.0;
    IM   ELEMENT  = 0 0;
   end record;

end GEN_COMPLEX;
-- Package Complex body which implements the functions for type
-- complex
--
with TEXT_IO, INTEGER_TEXT_IO;
use  TEXT_IO, INTEGER_TEXT_IO;
--
package body GEN_COMPLEX is

--
--  Addition Functions
--
 function "+"(LEFT, RIGHT   COMPLEX ) return COMPLEX is
  ANS   COMPLEX ;
 begin
  ANS RE  = LEFT RE + RIGHT.RE;
  ANS IM  = LEFT IM + RIGHT IM;
  return ANS;
 end "+";

--
 function "+"(LEFT   ELEMENT; RIGHT   COMPLEX ) return COMPLEX is
  ANS   COMPLEX ;
 begin
  ANS RE  = LEFT + RIGHT RE;
  ANS.IM  = RIGHT IM;
  return ANS,
 end "+",

--
 function "+"(LEFT   COMPLEX; RIGHT   ELEMENT) return COMPLEX is
  ANS   COMPLEX ;
 begin
  ANS RE := LEFT RE + RIGHT,
  ANS.IM := LEFT.IM,
  return ANS,
 end "+";
```

Figure 38 Cont'd. on next page

Figure 38 (Cont.): Generic Complex Package

```
--
--  Subtraction Functions
--
 function "-"(LEFT, RIGHT   COMPLEX ) return COMPLEX is
  ANS   COMPLEX ,
 begin
  ANS.RE := LEFT RE - RIGHT RE,
  ANS.IM .= LEFT IM - RIGHT.IM;
  return ANS;
 end "-",

--
 function "-"(LEFT   ELEMENT; RIGHT . COMPLEX ) return COMPLEX is
  ANS   COMPLEX ,
 begin
  ANS.RE := LEFT - RIGHT.RE,
  ANS IM  = -RIGHT.IM,
  return ANS;
 end "-",

--
 function "-"(LEFT · COMPLEX, RIGHT   ELEMENT) return COMPLEX is
  ANS · COMPLEX ,
 begin
  ANS.RE := LEFT RE - RIGHT,
  ANS.IM .= LEFT IM,
  return ANS;
 end "-",
--
--  Multiplication Functions
--
 function "*"(LEFT, RIGHT   COMPLEX ) return COMPLEX is
  ANS   COMPLEX ,
 begin
  ANS RE  = (LEFT.RE*RIGHT.RE)-(LEFT.IM*RIGHT IM);
  ANS.IM  = (LEFT IM*RIGHT RE)+(LEFT RE*RIGHT IM),
  return ANS,
 end "*",

--
 function "*"(LEFT · ELEMENT, RIGHT · COMPLEX ) return COMPLEX is
  ANS . COMPLEX ,
 begin
  ANS.RE  = LEFT*RIGHT RE,
  ANS.IM  = LEFT*RIGHT IM;
  return ANS,
 end "*",

--
 function "*"(LEFT   COMPLEX, RIGHT   ELEMENT) return COMPLEX is
  ANS   COMPLEX ,
 begin
  ANS RE  = LEFT RE*RIGHT;
  ANS IM  = LEFT IM*RIGHT,
  return ANS,
 end "*";
```

Figure 38 Cont'd. on next page

Figure 38 (Cont.): Generic Complex Package

```
--
--  Division Functions
--
 function "/"(LEFT, RIGHT : COMPLEX ) return COMPLEX is
  ANS . COMPLEX ;
 begin
  if RIGHT IM = 0.0 then     -- If only Real part
   ANS := LEFT / ELEMENT( RIGHT.RE ),
  else
   ANS.RE  = ((LEFT RE*RIGHT RE)+(LEFT.IM*RIGHT.IM))/
      (RIGHT.RE**2+RIGHT IM**2);
   ANS IM  = ((LEFT IM*RIGHT.RE)-(LEFT RE*RIGHT.IM))/
      (RIGHT.RE**2+RIGHT.IM**2),
  end if;
  return ANS,
 end "/",

--
 function "/"(LEFT   ELEMENT; RIGHT   COMPLEX ) return COMPLEX is
  ANS   COMPLEX ;
 begin
  if RIGHT IM = 0 0 then     -- If only Real part
   ANS.RE  = LEFT / ELEMENT( RIGHT RE );
   ANS IM  = 0 0,
  else
   ANS.RE  = (LEFT*RIGHT RE)/(RIGHT.RE**2+RIGHT.IM**2);
   ANS IM  = (LEFT*(-RIGHT.IM))/(RIGHT RE**2+RIGHT IM**2);
  end if,
  return ANS,
 end "/",

--
 function "/"(LEFT   COMPLEX, RIGHT   ELEMENT) return COMPLEX is
  ANS . COMPLEX ,
 begin
  ANS RE  = (LEFT RE/RIGHT),
  ANS IM  = (LEFT IM/RIGHT),
  return ANS,
 end "/";
--
--  Complex Conjugate Function
--
 function CONJ (NUM   COMPLEX) return COMPLEX is
 begin
  return (RE=>NUM RE, IM=>(-NUM IM)) ,
 end CONJ,


--
--  Functions to return Real and Imaginary parts of Complex
--   Number
--
 function RE (NUM   COMPLEX) return ELEMENT is
 begin
  return NUM RE,
 end RE;
```

Figure 38 Cont'd. on next page

Figure 38 (Cont.): Generic Complex Package

```
function RE (NUM : COMPLEX_MATRIX) return ELEMENT_MATRIX is
 TEMP : ELEMENT_MATRIX( NUM'range(1), NUM'range(2) ),
begin
 for I in NUM'range(1) loop
  for J in NUM'range(2) loop
   TEMP(I,J)  = RE( NUM( I,J) );
  end loop,
 end loop;

 return TEMP;
end RE;


function IM (NUM · COMPLEX) return ELEMENT is
begin
 return NUM IM;
end IM;

function IM (NUM   COMPLEX_MATRIX) return ELEMENT_MATRIX is
 TEMP   ELEMENT_MATRIX( NUM'range(1), NUM'range(2) ),
begin
 for I in NUM'range(1) loop
  for J in NUM'range(2) loop
   TEMP(I,J)  = IM( NUM( I,J) ),
  end loop;
 end loop,

 return TEMP,
end IM,


--
--  Function to compute the NORM of a complex number
--
 function NORM (NUM   COMPLEX) return ELEMENT is
 begin
  return SQRT(NUM RE**2 + NUM IM**2);
 end NORM,
function MAGNITUDE( NUM   in COMPLEX ) return ELEMENT is
begin
 return SQRT(NUM RE**2 + NUM IM**2),
end MAGNITUDE,


function PHASE ( NUM   in COMPLEX ) return ELEMENT is
 PIE  · constant ELEMENT  = 22 0/7.0,
 ANGLE   ELEMENT,
begin
 ANGLE  = ( ATAN ( NUM IM / NUM RE ) * 180 0/PIE );

 -- Now account for Complex Location as ATAN only returns a number from -90 to 90
 if NUM RE < 0 0 and NUM IM >= 0 0 then
  ANGLE ·= ANGLE + 180 0,
 elsif NUM RE < 0 0 and NUM IM < 0 0 then
  ANGLE  = ANGLE - 180 0,
 end if;

 return ANGLE;

 exception
```

Figure 38 Cont'd. on next page

Figure 38 (Cont.): Generic Complex Package

```
  when NUMERIC_ERROR => -- i.e. ATan ( Infinite ) = 90 deg
   if NUM.IM < 0.0 then
    return -90.0;
   else
    return 90.0;
   end if;
end PHASE;


function MAGNITUDE( NUM : in COMPLEX_MATRIX ) return ELEMENT_MATRIX is
 TEMP : ELEMENT_MATRIX( NUM'range(1), NUM'range(2) );
begin
 for I in NUM'range(1) loop
  for J in NUM'range(2) loop
   TEMP(I,J) := MAGNITUDE( NUM(I,J) );
  end loop;
 end loop;
 return TEMP;
end MAGNITUDE;


function PHASE ( NUM : in COMPLEX_MATRIX ) return ELEMENT_MATRIX is
 TEMP : ELEMENT_MATRIX( NUM'range(1), NUM'range(2) );
begin
 for I in NUM'range(1) loop
  for J in NUM'range(2) loop
   TEMP(I,J) := PHASE( NUM(I,J) );
  end loop;
 end loop;
 return TEMP;
end PHASE;
function TRANS ( A : in COMPLEX_MATRIX ) return COMPLEX_MATRIX is
 TEMP : COMPLEX_MATRIX( A'range(2) , A'range(1) );
begin
 for I in A'range(2) loop
  for J in A'range(1) loop
   TEMP(I,J) := A(J,I);
  end loop;
 end loop;
 return TEMP;
end TRANS;


--
--
-- Form a Complex Data Element from Real and IMG parts
--
function FORM_COMPLEX ( RE, IMG : in ELEMENT ) return COMPLEX is
begin
 return ( RE, IMG );
end FORM_COMPLEX;


function FORM_COMPLEX ( RE : in ELEMENT ) return COMPLEX is
begin
 return ( RE, 0.0 );
end FORM_COMPLEX;
```

Figure 38 Cont'd. on next page

Figure 38 (Cont.): Generic Complex Package

```
--
--  Input/Output to terminal Routines
--
procedure GET ( NUM : out COMPLEX ) is
begin
 new_line,put ("Enter Real Part       > "),
 get(NUM.RE),
 new_line;put ("Enter Imaginary Part  > "),
 get(NUM IM);
end GET;

--
procedure PUT ( NUM   in COMPLEX ) is
begin
 put('('),put(NUM RE),put(", ");
 put(NUM IM);put(')');
end PUT;
--

procedure PUT_LINE ( NUM : in COMPLEX ) is
begin
 put('('),  put(NUM RE),
 put(", "), put(NUM IM), put(')'),
 new_line,
end PUT_LINE;
--
procedure GET ( MAT   out COMPLEX_MATRIX ) is
 TEMP · COMPLEX_MATRIX ( MAT'RANGE(1), MAT'RANGE(2) ),
begin
 new_line(2),
 for I in TEMP'RANGE(1) loop
  for J in TEMP'RANGE(2) loop
   put("Element "),put (I),put(','),put(J),
   put(" > "),get( TEMP(I,J) ),
  end loop,
 end loop,
end GET,

procedure PUT ( MAT   in COMPLEX_MATRIX ) is
begin
 new_line(2);
 for I in MAT'RANGE(1) loop
  new_line,
  for J in MAT'RANGE(2) loop
   new_line,
   put(  MAT(I,J) ),put("  "),
  end loop,
 end loop,
end PUT;
```

Figure 38 Cont'd. on next page

Figure 38 (Cont.): Generic Complex Package

```
procedure GET ( MAT : out COMPLEX_VECTOR ) is
 TEMP : COMPLEX_VECTOR ( MAT'RANGE );
begin
 new_line(2),
 for I in TEMP'RANGE loop
   put("Element "),put (I);
   put(" > "),get( TEMP(I) ),
 end loop,
end GET,

procedure PUT ( MAT . in COMPLEX_VECTOR ) is
begin
 new_line(2),
 for I in MAT'RANGE loop
  new_line;
  put(  MAT(I) ),
 end loop,

end PUT;

end GEN_COMPLEX,
```

APPENDIX E

MSDI TRANSLATIONAL GRAMMAR

This appendix details the grammar used for the prototype Version of MSDI to parse an input sentence tokenized by the scanner utility. The Translation Grammar is LL(1).

EMPTY corresponds to the end of sentence, i.e. EMPTY TOKEN from Scanner (Token_ Class = Empty).

Deliminators = [' ', ASCII.HT]

Delim_Tokens = [';', ',', '=', '+, '-', '*', '/', "", '(', ')', '[', ']', '^', '{', '}', ""]

Keywords = Discretize, Exit, Simulate, Show, Inv, Cos, Sin, Tan, GED, Eigen, Diag, Ident.

```
Exec_Stmt          ->  Assign_Stmt                                  [ Ident_type ]
                   ->  Keyword_Stmt                                 [ Keyword_Type ]
                   ->  "null"                                       [ Empty  ]

Assign_Stmt        ->  Identifier Assign_Type "=" Express           [ Ident_Type ]

Assign_Type        ->  "(" Variable ")"                             [ ( ]
                   ->  "null"                                       [ = ]

Keyword_Stmt       ->  "Discretize" Discrete_Stmt                   ["Discretize"]
                   ->  "Show"  Express                              ["Show" ]
                   ->  "Simulate" Sim_Stmt                          ["Simulate"]
                   ->  "Exit"                                       ["Exit"]
                   ->  "Ged"                                        ["Ged" ]

Discrete_Stmt      ->  "(" Identifier Identifier_List ")"
                       Discrete_Out                  [ "(" ]

Discrete_Out       ->  "/" "Output" "=" Identifier Identifier_List      [ "/" ]
                   ->  "null"                                           [ Empty ]

Sim_Stmt           ->  Dynamic_Part Feedforward_Part                [ Express ]

Dynamic_Part       ->  Express Express_List Express_List            [ Express]

Feedforward_Part   ->  Express_List                                 [ ","   ]
                   ->  "null"                                       [ Empty ]

Express_List       ->  "," Express                                  [ "," ]

Identifier_List    ->  "," Identifier                               [ "," ]

Express            ->  Term Expresses                               [ Head(Factor) ]

Expresses          ->  ASOP Term Expresses                          [ -,+ ]
                   ->  "null"                                       [ Empty, ",",) ]
```

```
Term              ->  Factor Hi_Preced_Op Terms                [ Head(Factor) ]

Terms             ->  MDOP Factor Hi_Preced_Op Terms           [ /,*    ]
                  ->  "null"                                   [-,+,Empty,",",) ]

Hi-Preced_Op      ->  "^" Factor Hi_Preced_Op                  [ ^ ]
                  ->  "'"                                      [ ' ]
                  ->  "null"                        [ *, /,-,+, Empty, "," , ) ]

Factor            ->  ASOP Unsigned_Factor                     [ +, - ]
                  ->  Unsigned_Factor                          [Head(Un_Fac) ]

Unsigned_Factor   ->  ( Express )                         [ ( ]
                  ->  Operand                             [ Num_Type, Ident ]
                  ->  Matrix_Def                          [ [ ]
                  ->  Poly_Def                            [ { ]
                  ->  Function_OP                         [ Function_names ]

Function_OP       ->  "Cos" "(" Express ")"               [ Cos ]
                  ->  "Sin" "(" Express ")"               [ Sin ]
                  ->  "Tan" "(" Express ")"               [ Tan ]
                  ->  "INV" "(" Express ")"               [ INV ]
                  ->  "Diag" "(" Express ")"              [ Diag ]
                  ->  "Ident" "(" Express ")"             [ Ident ]

ASOP              -> +                                    [ + ]
                  -> -                                    [ - ]

MDOP              ->  *                                   [ * ]
                  ->  /                                   [ / ]

Matrix_Def        ->  "[" Row Next_Row "]"                [  [   ]

Row               ->  Factor List_of_Elements             [ Head(Factor) ]

Next_Row          ->  Row Next_Row                        [ Head(Factor) ]
                  ->  EOL Next_Row                        [ Empty        ]
                  ->  "null"                              [ ]            ]

List_of_Elements  ->  Factor List_of_Elements             [ Head(Factor) ]
                  ->  ","                                 [ , ]
                  ->  "null"                              [ Empty, ] ]

Poly_Def          ->  "{" Poly_Term Poly_Terms "}"        [ {  ]

Poly_Term         ->  Variable Power                      [ Variable ]
                  ->  Factor Poly_Power                   [ Head(Factor) ]

Poly_Power        ->  "*" Variable POWER                  [ * ]
                  ->  "null"                              [ +, -, }  ]

Power             ->  "^" Integer                         [ ^ ]
                  ->  "null"                              [ +, -, }  ]

Poly_Terms        ->  ASOP Poly_Term Poly_Terms           [ +, - ]
                  ->  "null"                              [ }    ]
```

```
Head( Unsigned_Factor ) =  (, Ident, Num_Type, [ , INV, Cos, Sin,
                           Tan, Eigen, {

Head( Factor ) = Head( ASOP ) + Head( Unsigned_Factor )
               = + , - , (, Ident, Num_Type, [ , Function_Names

Function_Names = INV, Cos, Sin, Tan, Eigen

Variable       = an Identifier that is set by Assign_Type first, then
                 compared to this Variable_type for reference later in
                 parse.

Note
_____

Procedure for adding a Function .

        (1) Add the new function name to the Keywords in TG_ ada

        (2) Add new keyword to Perform ( Monadic Operations ) in TG ada
              - add to perform procedure options
              - add action routine to perform function.

        (3) Add the new function_name to Function_Op

        (4) Update Documentation with new function
```

APPENDIX F

REACTOR EXAMPLE MODEL ELEMENTS

For Reactor State Space Model .

A = [

```
 9.764E-01 -1.672E-03  1.522E-03 -1.569E-03  1 764E-03 -2 477E-03  1 519E-03
 9.077E-03  9 814E-01 -2.949E-03  2 578E-03 -2 755E-03  3 803E-03 -2.326E-03
 5 440E-03  1 828E-02  9 850E-01 -3 969E-03  3.599E-03 -4.732E-03  2 869E-03
-1.686E-03  5 116E-03  2.303E-02  9 858E-01 -4 281E-03  4 762E-03 -2 811E-03
-3.771E-03 -7 369E-03  6.826E-03  2.485E-02  9.827E-01 -3.261E-03  1 659E-03
-8.951E-04 -8 536E-03 -5.776E-03  1.709E-02  1 974E-02  9 781E-01 -1.125E-03
 1.465E-04 -7 578E-03 -8.367E-03  1.415E-02  1.995E-02  7.886E-03  9.726E-01  ]
```

```
B = [  2.56193279E-02 1.76716092E-04
       1.50561591E-02 6 39012050E-04
      -1 10366099E-03 9 44330516E-04
      -5 43357878E-03 9.06964715E-04
       2 63209210E-03 1 00908396E-03
       6 35339621E-03 1.25555413E-03
       6 41673940E-03 1 35241377E-03   ]
```

C = [

```
-3.755E-01  8 200E-01  4.8340E-01 -1 698E-01  1 229E-01 -1.427E-01  8 40E-02
 2 118E-01 -2 439E-01  5.7660E-01  7 193E-01 -2.745E-01  2.758E-01 -1 585E-01
-8 520E-02  8 600E-02 -1.1990E-01  3.097E-01  9 534E-01 -4 021E-01  2 112E-01
-8 460E-02 -1 912E-01 -2.8040E-01 -3 298E-01 -3 012E-01 -1 621E-01  0 000E+00  ]
```

REFERENCES

1. James, J., "Considerations concerning the construction of an expert system for Control System Design"; Ph.D. Thesis, Rensselaer Polytechnic, 1986.
2. Taylor, J. and Frederick, D., "An Expert System Architecture for Computer-Aided Control Engineering"; Proc. of IEEE, Vol 72, no. 12, pp. 1795-1805 Dec. 1984.
3. Noton, M., "A Survey of Control Methodologies Applied to Spacecraft Control and Guidance", Proc. of Control'88, Oxford, U.K., 1988.
4. Melsa, J.L. and Jones S.K., "Computer Programs for Computational Assistance in the Study of Linear Control Theory", 2nd edition, New York, McGraw-Hill, 1973.
5. Rosenbrock, H.H., "Computer-Aided Control System Design", Academic Press, New York, 1974.
6. Moler, C., "MATLAB User's Guide", Dept. of Computer Science, University of New Mexico, Albuquerque, U.S.A, 1980.
7. Hickey. J., "Design Process Model Development Report", Research Report CTRU8803, Control Technology Research Unit, Dublin City University, Jan. 1988.
8. Jamshidi, M. and Herget C.J., (Eds), "Computer-Aided Control Systems Engineering", North Holland, 1985.
9. "SPICE User Guide", University of California at Berkeley, US, 1978.
10. "PNE/6, Multiple Input/Multiple Output Design Package", Notes for Guidance to students, Internal Document, Computer Services Dept., Ulster Polytechnic, 1980.
11. Shah, S., Floyd, M., and Lehman, L., "MATRIXx: Control Design and Model Building CAE Capability", in [8].
12. Sprang, A., "The Federated Computer-Aided Control Design System", Proc. of IEEE, Vol 72 No. 12, pp. 1724-1731, Dec. 1984.
13. -, "Structured Computer Aided Logic Design", User Guide , Valid Systems Inc.
14. Erman, L.D et al, "Hearsay-II Speech Understanding System : Integrating Knowledge to Resolve Uncertainty", Computing Surveys 12, 1980.
15. Nolan, P.J., "An Intelligent Assistant for Control System Design", Proc. of 1st Inter. Conf. on Application of AI in Engineering Problems, Uni. of Southampton, U.K., 1986.
16. Yoshikawa, H., "CAD Framework Guided by General Design Theory", CAD Systems Framework, Proc. of IFIP, Norway 1982.
17. Asimov, M., "Introduction to Design", Prentice-Hall, 1962.
18. Ross, D.T., "Structured Analysis for Requirement Definitions", 2nd Int. Conf. on Software Engineering, 1976.

19. Warman, E.A."Man in a Machine Environment", Proc. Man-Machine Communication in CAD/CAM, IFIP, 1980.

20. Begg, V., "Developing Expert CAD Systems ", Addison Wesley, Reading MA, 1984.

21. Yourdon. E., "Classics in Software Engineering", Prentice-Hall, Englewood Cliffs, N.J., 1979.

22. Bjorner, D. and Jones, C., "Formal Specification and Software Development", Prentice-Hall, Englewood Cliffs, N.J., 1982.

23. Yourdon, E. and Constantine, L., "Structured Design : Fundamentals of a Discipline of Computer Program and System Design", Prentice-Hall, 1979.

24. Deming, E., "Out of the Crisis", Cambridge University Press, Cambridge, Melbourne, Australia, 1986

25. Byrne, D. and Taguchi, S., "The Taguchi Approach to Parameter Design" ASQC Annual Conf. Transactions, 1986.

26. Nelson, G.,"A Generalisation of Dijkstra's Calculus", Digital Systems Research Centre, Research Report 16, 1987.

27. Harrison, F.L., "Advanced Project Management", McGraw-Hill, 1985.

28. Denham, M., "Design Issues for CACSD Systems", Proc. of IEEE, Vol 72 No. 12, pp. 1714-1723, Dec. 1984.

29. Pang G. and MacFarlane, A., "An Expert Systems Approach to Computer-Aided Design of Multivariable Systems", Springer-Verlag Berlin, Heidelberg, 1987.

30. Bennett, J.S and Engelmore R., "SACON A Knowledge-Based Consultant for Structural Analysis", Proc. of Sixth Inter. Joint Conf. on Artificial Intelligence, Tokyo,1979.

31. Haber, R.N. and Wilkinson, L., "Perceptual Components of Computer Displays", IEEE Computer Graphics and Applications, Vol 2, No. 3, 1982.

32. Williams, S., "The Changing Face of CACSD Tools", Computer-Aided Control System Design, Institute of Measurement and Control, Salford UK, July 1986.

33. Rimvall, M., "Man-Machine Issues in CACSD", Computer-Aided Control System Design, Institute of Measurement and Control, Salford UK, July 1986.

34. Goodfellow, S. and Munro, N., "Integra, an Input Translation Facility for Computer Aided Control System Engineering" Proc. 3rd IFAC Computer Aided Design in Control and Engineering Systems, Lyngby, Denmark, 1985.

35. Munro, N., "Ecstasy - A Control System CAD Environment", Proc. of Control'88, Oxford, UK, 1988.

36. Rimvall, M., "Man-Machine Interfaces for CACSD and implementational aspects", PhD thesis, ETH Zurich, 1987.

37. Barker, H., Chen, M., Jobling, P. and Townsend, P., "Interactive Graphics for the Computer-Aided Design of Dynamic Systems", IEEE Control Systems Magazine, pp. 19-25, June 1987.

38. "Proc. 3rd IFAC symposium on Computer-Aided Design in Control and Engineering Systems" , Lyngby, Denmark, 1985.

39. Astrom, K.J., Wittenmark, B., "Computer Controlled Systems", Prentice-Hall, Englewood Cliffs, N.J., 1984.

40. Laub, A.J., "Numerical Linear Algebra Aspects of Control Design Computations", IEEE Trans. on Auto. Control, Vol AC-30, No. 2, pp. 97-108, Feb. 1985.

41. Date, C.J., "An Introduction to Database Systems", Prentice-Hall, 1985.

42. Brooks, "The Mythical Man Month", Addison-Wesley, Reading, MA, 1975.

43. Fisher, D.A, "A Common Programming Language for the Department of Defence - Background and Technical Requirements ", Report P-1191, Institute for Defense Analysis, Arlington, Va, June, 1976.

44. Ross, D.T, Goodenough, J.B. and Irvine, C.A., "Software Engineering : Process, Principles and Goals", Computer, May, 1975.

45. Sommerville, I., "Software Engineering", Addison-Wesley, Reading, MA, 1985.

46. Jackson, M., "The Jackson Design Methodology, Infotec State of the Art Report, Structured Programming, 1978.

47. Parnas, D.L., "On the Criteria to be used in Decomposing a System into Modules", CMU-CS-71-101, Communications of the ACM, Vol. 15, No. 12, Dec., 1972.

48. Hickey, J., "Software Engineering Considerations for CACE", Research Report CTRU8807, Control Technology Research Unit, Dublin City University, Jan. 1988.

49. Hickey, J., "A CAD package for the Design of Digital Control Systems", Project Report for B.Eng, N.I.H.E, Dublin, 1984.

50. "VAX GKS Reference Manual", V3.0, Digital Equipment Corp.

51. Booch, G., "Software Engineering with Ada" , Benjamin/Cummings Publishing Company, 1986

52. Birdwell, J.D., Cockett, J.R., Heller, R., Rochelle, R., Laub, A.J., Athans, M., and Hatfield, L., "Expert Systems Techniques in a Computer-Based Control System Analysis and Design Environment", Proc. 3rd IFAC symposium on Computer-Aided Design in Control and Engineering Systems, Lyngby, Denmark, 1985.

53. Williams, D. and Friedland B.,"Modern Control Theory for Design of Autopilots for Bank-to-Turn Missiles", American Control Conference, Vol 2, 1986.

54. Hvelplund H., "Modelling System Specifications on A Finite State Machine", Proc. 3rd IFAC Computer Aided Design in Control and Engineering Systems, Lyngby, Denmark, 1985.

55. Hickey, J., "Survey of Current CACSD Packages", Research Report CTRU8804, Control Technology Research Unit, Dublin City University, Jan. 1988.

56. James, J., Taylor, J. and Frederick, D. "An Expert System Architecture for Coping with Complexity in Computer-Aided Design", Proc. of 3rd. IFAC Symposium on CAD in Control and Engineering Systems, Lyngby, Denmark, August, 1985.

57. Feur and Gehani, "Comparing and Assessing Programming Languages Ada , C and Pascal ".

58. Maciejowski, J.M, "Data Structures for Control System Design", Proc. EUROCON 84, Brighton, 1984.

59. Hickey, J., "Development of MSDI Architecture", Research Report CTRU8805, Control Technology Research Unit, Dublin City University, Jan. 1988.

60. "IEEE Control System Magazine", Vol. 2, No. 4, 1982.

61. Liu, C.L., "Elements of Discrete Mathematics", Liu New York: McGraw-Hill, 1977.

62. Hickey, J., "A Study of Parsing Methods", Research Report CTRU8808, Control Technology Research Unit, Dublin City University, Mar 1988.

63. Pyster, A., "Compiler Design and Construction", Van Nostrand Reinhold, 1980.

64. "VAX Document User Manual, Volume 1", V1.0, Digital Equipment Corp

65. Moler, C., and Van Loan C., "Nineteen Dubious Ways to Compute the Exponential of a Matrix", Siam Review, Vol 20, No. 4, pp 801-836, 1978.

66. Gustavsson, I., "Comparison of Different Methods for Identification of Industrial Processes", Automatica, vol. 8, 1972.

67. "A Layout Algorithm for Dataflow Diagrams", IEEE Trans on Software Engineering, Vol. SE-12, No. 4, April 1986

68. Laub, A.J., "Efficient Multivariable Frequency Response Computations", IEEE Trans. Aut. Control, AC-26, pp. 407-408, 1981

69. MacFarlane, A G J, and Postlethwaite, I., "The Generalised Nyquist Stability Criterion and Multivariable Root Loci", Int. J Control, 25, 1977.

70. 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

71. "Coupled-Tanks Apparatus - Modelling and Experimenting", Internal Document, School of Electronic Engineering, Dublin City University

72. "VAX Document, User Guide Volume 1", Digital Equipment Corp., 1988.

73. Hanselmann, H , "Implementation of Digital Controllers - A Survey", Automatica, Vol 23, No 1, pp 7-32, 1987

74. Dorato, P, "Theoretical Developments in Discrete-Time Control", Automatica, Vol 19, No. 4, pp 395-400, 1983.