Computing Environments for Control Engineering

Anita Bilqees Saifuddin

Newnham College
Cambridge

Submitted for the Degree of Doctor of Philosophy
Department of Engineering
University of Cambridge

March 1996

To my parents

Declaration

Except where otherwise stated, this thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration. Any reference to the work of other researchers is clearly indicated in the text.

None of the material contained in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institute of learning.

Anita B Saifuddin
Cambridge
29 March 1996

Acknowledgements

I am extremely grateful to my supervisor Dr J.M. Maciejowski for sharing his invaluable expertise and for his constant support, guidance and encouragement throughout the course of my research. His comments on draft manuscripts of the thesis have been very useful. I am also very grateful to Dr M. Szymkat for his technical assistance and invaluable discussions which extensively contributed to my research. I am grateful to him for inviting me to the Institute of Mining and Metallurgy in Krakow, and to all the other members of his group for their support and hospitality.

I extend my sincere thanks to all members of the control group and especially to John Woollatt for his help and encouragement. I am also grateful to Ken Wallace for allowing me to spend some time in the Engineering Design Centre and for making me feel welcome, and to Dr N. Ball with whom I had several informal discussions during my stay.

Special thanks to Ellie and Karen for proof reading draft manuscripts.

I am greatly indebted to all my friends who made my stay at Cambridge an unforgettable experience. Thanks to Jenny, Mary and Susy for their friendship which has meant so much to me.

My most heartfelt thanks go to Dr Lewicka and her family with whom I spent memorable five weeks in Krakow. Thanks to the whole family for the wonderful time I had with them and for making my stay very comfortable and stimulating for work.

No words can express my gratitude towards my parents to whom I dedicate this thesis for their continuing encouragement and support and for bearing my long absence from home.

This work has been supported by Cambridge Commonwealth Trust Scholarship and Overseas Research Student Award. I am also grateful to the Department of Engineering, Newnham College and Cambridge Philosophical Society for travel grants and research studentships.

Abstract

Present day computing environments for control engineering have come a long way in terms of numerical support and command or graphical interfaces which provide 'natural and flexible' programming support for numerical computations. However, this ease of use and support for flexible and interactive numerical programming has come at a price. Most of these environments lack the structure and organization capabilities necessary for managing model data, parameters, intermediate values and results generated during different computations. They also lack facilities for managing the workspace during any control design session and for maintaining different versions of models and results for a project. There have been several attempts at providing some kind of data management functionality, but most have not been sufficient or general enough, or have proved to be too rigid and restrictive. A review of several environments for computer-aided control engineering (CACE) shows that the common practice of using an external tool for data management is not very practical and that data management functionality should be incorporated as far as possible into the numerical programming environment. Furthermore, any such feature incorporated into the programming process should closely correspond to other programming constructs in the environment. Object-oriented methodologies are good candidates for such developments. These considerations form part of requirements analysis, on the basis of which an experimental-study for overcoming data management problems in Matlab has been conducted. The study is based on Matlab *Containers* which provide a general solution to the systematic creation and manipulation of data and functions. They are also used for the implementation of class-composition hierarchies and computational chains. An extended example illustrates how the new functionalities can be used without compromising Matlab's flexible programming style.

Contents

Glossary

ANDECS	Analysis and Design of Control Systems.
AutoCode	Module of Matrix$_x$. Trademark of Integrated Systems, Inc.
CACE	Computer-Aided Control Engineering.
CACSD	Computer-Aided Control Systems Design.
CAD	Computer-Aided Design.
CAE	Computer-Aided Engineering.
Caesy	Computer-Aided Engineering System.
CAM	Computer-Aided Manufacturing.
CASE	Computer-Aided Software Engineering.
CATPAC	Interactive Software Package for Control System Design.
CDIF	CASE Data Interchange Format.
CLADP	Cambridge Linear Analysis and Design Programs.
CSSL	Continuous System Simulation Language.
CST	Control Systems Toolbox for Matlab.
CTRL-C	Trademark of Systems Control Technology, Inc.
DAE	Differential Algebraic Equation.
DocumentIt	Module of Matrix$_x$. Trademark of Integrated Systems, Inc.
Dymola	Dynamic Modelling Language.
ECMA	European Computer Manufacturer's Association.
ECSTASY	Environment for Control System Theory and Synthesis.
EISPACK	Matrix Eigen System Routines.
FRM	Framework Reference Model.
GE-MEAD	Multi-disciplinary Expert-aided Analysis and Design.
GUI	Graphical User Interface.
IDPAC	Trademark of Lund Institute of Technology.
IMPACT	Trademark of Swiss Federal Institute of Technology.
INTRAC	Trademark of Lund Institute of Technology.

ISID	Xmath Interactive System Identification Module. Trademark of Integrated Systems, Inc.
ISO	International Standards Organization.
Maple	Trademark of Waterloo Maple Software, Ontario, Inc.
Matlab	Trademark of The MathWorks, Inc.
Matrix$_x$	Trademark of Integrated Systems, Inc.
MFD	Multivariable Frequency Domain Toolbox. Trademark of Cambridge Control Ltd.
MODPAC	Trademark of Lund Institute of Technology.
MUT	μ–Analysis and Synthesis Toolbox for Matlab. Trademark of MUSYN Inc., and The MathWorks, Inc.
NCD	Nonlinear Control Design Toolbox for Matlab. Trademark of The MathWorks, Inc..
ODE	Ordinary Differential Equation.
OMG	Object Management Group.
Omola	Object-oriented Modelling Language.
OMS	Object Management System.
ORB	Object Request Broker.
PCTE	Portable Common Tool Environment.
POLPAC	Trademark of Lund Institute of Technology.
RCT	Robust Control Toolbox for Matlab. Trademark of The MathWorks, Inc.
SID	System Identification Toolbox for Matlab. Trademark of The MathWorks, Inc.
SIMNON	Trademark of Lund Institute of Technology.
Simulink	Trademark of The MathWorks, Inc.
SSDP	State-Space Design Package.
SYNPAC	Trademark of Lund Institute of Technology.
SystemBuild	Module of Matrix$_x$. Trademark of Integrated Systems, Inc.
TDT	Time Delay Toolbox for Matlab.
VLSI	Very Large Scale Integration.
Xmath	Module of Matrix$_x$. Trademark of Integrated Systems, Inc.

Chapter 1

Introduction

Over the last four decades computers have gained and they continue to gain an ever-increasing importance in control engineering. Computer programs for automatic control have traditionally been constructed for two different purposes: to aid the control engineer in modelling, analysis and design of a control system, and to implement real-time programs for digital realization of a control system. To gain an insight into the role played by computers in control engineering, it is best to look into the nature of the control design cycle. The major stages involved in the control design cycle are shown in figure 1.1.

The first stage in the control design cycle is the conceptual phase of problem formulation and specification of the control objectives. For example, the objective of a control system might be to regulate a component, to minimize energy consumption or to control the mechanical movement such as that of a robot arm, etc. After the control objectives have been formulated, a dynamic model of the process is developed. The model can have a theoretical basis (physical and chemical principles such as conservation laws and rates of reactions) or it can be developed empirically from experimental data. The goals of the modelling step are to obtain a formal representation of the plant and to acquire a basic understanding of its main properties. The next step in control design is to devise an appropriate control strategy that will meet the control objectives. Usually, a controller based on the plant model and on the desired closed loop performance is derived. This step involves selection of regulator structure and design method for deriving proper regulator parameters. Computer simulation plays an important role in modelling, assessing alternative control strategies and determining preliminary estimates of suitable controller settings. Finally, the control system hardware is selected, ordered, and installed in the plant. Then the controllers are tuned in the plant using the preliminary estimates from the design step as a starting point. Controller tuning usually involves trial and error procedures.

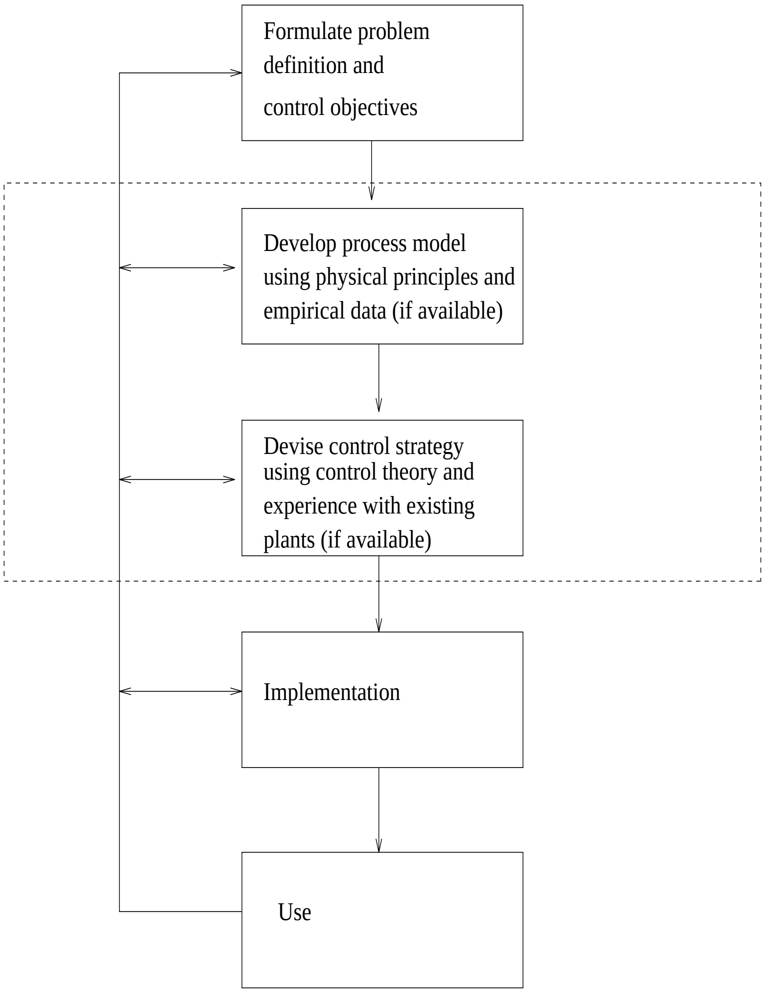

Figure 1.1: The control design cycle

Generally, intuition developed while experimenting with the physical system plays an important part in the design process. Depending on the system and the experience of the designer, some of the steps in the control cycle may be omitted. Also, the different steps in the control design cycle may follow a completely different sequence to the one shown in figure 1.1. For example, it is common for control systems to be implemented by choosing standard forms of controllers and experimentally determining the parameters of the controller by following a specified step-by-step procedure with the physical system; no mathematical models are developed.

Due to inaccuracies in the model, the tests on the physical control system are usually non-satisfactory. To improve the system, the control engineers then iterate the different stages of the design cycle. Iteration is a fundamental property of the design process.

A large number of computer programs exists to assist the control engineer in the different phases of the design cycle. In the early days of computer application to control engineering, the terms *off-line* and *on-line* programs were frequently used to classify control-related software. However, as pointed out in [85], these terms are not mutually exclusive. A real-time (on-line) data acquisition and/or identification package may be used during an otherwise off-line stage in the design cycle. On the other hand, the automatic tuner of an adaptive controller may be seen as a design component built into an on-line program. To avoid this confused terminology, more distinctive terms were adopted. One such term which gained wide-spread usage only a few years ago is computer-aided control system design (CACSD). As illustrated by the complete control design cycle in figure 1.1, the actual analytical or numerical design of a controller is only one of the many steps undertaken between the conception and the implementation of a control system. A broader definition of the term CACSD given by Jamshidi and Herget in [44] is:

"The use of digital computers as primary tool during the modelling, identification, analysis and design phases of control engineering."

This definition increases the scope of the term CACSD to include software for all phases of design cycle except for the real-time implementation of the controller. Recently the term computer-aided control engineering (CACE) has become more popular than CACSD. Often CACE is taken to be a specialized form of computer-aided engineering (CAE) and is defined as:

"A specialized form of modelling and simulation with emphasis on design and implementation of feedback control systems."

Software tools are not normally used in the problem formulation stage of the control design cycle, primarily because decisions made during this phase are based mainly on non-numerical operations and reasoning. Several software tools exist for the other stages of the design cycle. Several numerical processing toolkits have been developed for modelling, simulation, analysis and design phases. Many problems and questions asked in the development of CACE environments are common to many other domains of computer-aided engineering, for example, circuit design, software engineering, mechanical engineering, manufacturing engineering, etc.

Packages, *tools* (or toolboxes) and *environments* are the three most common terms used these days to refer to CACE software. The word 'package' is often used to refer to 'large numerical program libraries'. It was first used by IBM in 1961 when they developed the SSP (Scientific Subroutine Package) library [82]. The term was also used to refer to application programs such as SPSS (Statistical Package for the Social Sciences) developed in the early 60's. Numerical subroutine libraries such as EISPACK and LINPACK developed in the early 70's were also called packages. Since then, the word package has often been used to refer to an application program or to a large subroutine library. 'Environment' is a broader and more general term than package. It includes (in the context of CACE) integrated packages, modelling and simulation languages, and programming languages for control systems analysis and design. The word environment is most often used for an integrated set of tools. The concept of a tool is similar to that of a package except that a tool is generally used as a part of, and is dependent for its execution on, an environment. In this sense, an environment has a part that is common and accessible to all tools and which is based on some principle facilitating data transfer between tools.

1.1 Research Objectives

Present day CACE environments provide flexible programming support for numerical computations. However, they lack adequate support for non-numerical support areas particularly database management. In this thesis we aim to:

1. Review popular CACE environments over a period of about 20 years to trace the progress of these environments in the non-numerical support areas. The survey will also help determine the impact that developments in the general area of computing have had on CACE environments.

2. Formulate user requirements, especially in the context of language structure (modelling and simulation, and command language), programming style and database management.

3. Explore the database management aspects of CACE environments and find the useful database management features that are missing from present day popular CACE environments. A review of different approaches that have been suggested to provide better database management facilities and experiences of projects that incorporated these facilities but failed to gain popular support will give a good insight into the problem.

4. Implement a prototype system in a widely used CACE environment (in our case Matlab) that would enhance the overall database management functionality. This will help explore the database management aspects in CACE environments in general, and will also help determine whether there is a real need for such features as well as provide a possible way of implementing these features.

1.2 Thesis Overview

We now give a brief overview of the chapters in this thesis.

Chapter 2 reviews CACE environments and discusses user requirements. The chapter focuses on non-numeric issues such as programming style, command language, extensibility, adaptability to different kinds of users and database management issues (these are discussed in more detail in chapter 3). A classification of different kinds of users of CACE environments is also given.

Chapter 3 looks in detail at database management aspects of CACE. The issues discussed include support for compound data structures, user defined data structures, data models for maintaining semantics in a database, and standardization efforts for resolving inconsistencies of data representation in different tools in an integrated environment. Software architectures for database support in integrated CACE environments and some possible future architectures and 'framework reference models' for CACE environments are also discussed.

Chapter 4 examines a novel approach to solving the data management problem (outlined in chapter 2 and chapter 3) within the confines of a programming environment for control systems analysis and design. Matlab was selected as a test-bed for exploring this idea. A new concept called *Matlab Containers* as well as some other functionalities such as class-composition hierarchy and computational chains that have been built on top of Containers are discussed. The chapter discusses the concepts and implementation of these new features with the help of some examples drawn from control engineering.

Chapter 5 describes a case study based on the application of Containers for analysis and design of multivariable Smith predictors. The case study illustrates the use of Containers both with and without computational chains. The feasibility of defining class-composition hierarchies for representing different system descriptions of a multivariable Smith predictor is also discussed. The aim of the whole exercise is to highlight the advantages of using a facility like Containers.

Finally, chapter 6 states the conclusions drawn from the research along with some suggestions for future work.

Chapter 2

Review and Requirements Analysis

Features of CACE environments have always been determined by the computer technology commonly available at the time. CACE software—ranging from software for modelling, simulation and control design to support tools for documentation, database management and expert aiding—has benefited from research developments in the areas of software engineering, database management systems, interactive computing, object-oriented methodologies, artificial intelligence, parallel processing and high performance graphics. In this chapter, an attempt is made to determine how far CACE environments have progressed since the days of mainframe computing by analyzing the most prominent CACE environments, packages and toolboxes from an historical perspective. The main focus is on non-numeric issues of user interaction support, design support facilities, data structures and database support (discussed in detail in chapter 3), computational efficiency, adaptability to different kinds of users, and software portability, reusability and extensibility.

The chapter is divided into four sections. The first reviews CACE environments and is sub-divided into three sections: model building and simulation, control system analysis and design, and integrated CACE environments. Since modelling and simulation are extensively used in many other fields, the discussion in the section on 'model building and simulation' is more general than the discussion in other sections which mainly address aspects related to control engineering. The second section discusses user requirements and the third gives a classification of different kinds of users of CACE environments. The final section summarizes the main points raised in the chapter.

2.1 CACE Environments—A Review

2.1.1 Model Building and Simulation

As in many other branches of engineering, modelling and simulation play an important role in control engineering. In the early days of computer application to model building and simulation, there was no clear distinction between the two. Models were defined and simulated in the same program. Recently, model building and simulation have emerged as separate fields, with greater emphasis on model building.

A model can have a theoretical basis (fundamental laws such as mass balances, energy balances and phenomenological equations) or it can be developed empirically from experimental data. Modelling on the basis of physical knowledge of the system results in a number of equations describing the system. These are either algebraic equations or differential algebraic equations (DAE) and can be described by:

$$f(x, \dot{x}, u, t) = 0$$

There are several approaches to solving DAEs. Numerical methods that operate directly on DAEs can be developed; in some cases computer algebra packages such as Maple and Macsyma can be used to solve these equations. Alternatively, these equations can be transformed to a more convenient form for simulation, such as state-space description, i.e. a set of first-order ordinary differential equations (ODEs) of the form:

$$\dot{x} = f(x, u)$$
$$y = g(x, u)$$

Ideally, a user should provide the basic physical relationships and not be too concerned with putting them together into a version of the model that can be simulated. It is often a significant effort in terms of analysis and analytical transformations to obtain a model in the state-space form from DAEs (quite frequently it cannot be done or might introduce errors).

The continuous system simulation languages available these days that are based on state-space descriptions differ in syntactical details, but they are all derivations of the continuous system simulation language (CSSL) specification [97] generated by the CSSL committee in 1967. Different modelling and simulation software has different approaches for entering the system description in a systematic way. The system can be defined by subroutines written in a standard programming language like Fortran or C. The routines define the right side of a state-space description. This method is used in classical simulation languages like ACSL [1] (a language based on the CSSL standard). Another way is to enter the state-space description in a specially designed command language as, for example, in Simnon

[8]. The system description might be entered graphically in terms of block diagrams as is done in Simulink [94] and SystemBuild [98]. Often a simulation language gives the user the choice between different methods. Simnon, for instance, can handle only the first two methods while Simulink gives the user a choice between C code, m-files or block diagrams, that is, between the first three methods.

One of the problems in most simulation languages regards handling of algebraic loops. A typical example of an algebraic loop is when two linear systems with direct feed-through are connected in a loop as shown in figure 2.1 [52].

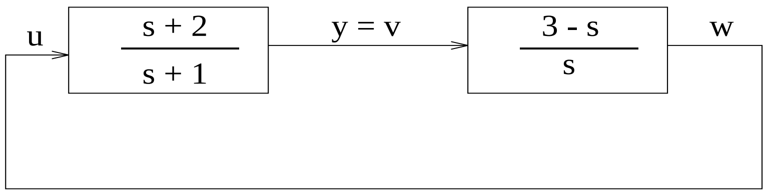

Figure 2.1: Two systems with direct feed-through connected in a loop

For both systems the output depends directly on the input (y depends directly on u and w depends directly on v). The signal u is equal to w, which depends directly on v, which is equal to y and which depends directly on u. The signals chase each other around and this situation is referred to as an algebraic loop.

Simulation software these days has the capability of detecting algebraic loops and of solving the kind of loop described above. An interesting example in this regard is DB-Prolog [102], which with its rule-based programming and back-tracking features presented an efficient way of detecting algebraic loops. A major problem in handling algebraic loops occurs when the system with an algebraic loop is not solvable. Such a case would arise if we consider the transfer function $(3+s)/s$ instead of $(3-s)/s$ in the above example. The state-space equations for the two systems are then given as:

$$\dot{x} = -x + u, \;\; y = x + u \tag{2.1}$$

and

$$\dot{z} = v, \;\; w = 3z + v \tag{2.2}$$

The connecting equations are $u = w$ and $v = y$. These equations can be written as linear system of equations in x and z as:

$$u - v = 3z, \;\; u - v = -x \tag{2.3}$$

which are unsolvable for u and v in terms of x and z. Therefore a state-space representation for the combined system cannot be obtained. In such a situation, simulation languages should report the problem to the user rather than give incorrect solution. Most simulation languages are capable of detecting algebraic loops and can simulate the system shown in figure 2.1, but the problem occurs with unsolvable systems such as that defined by the second system of equations (2.1 and 2.2). A detailed discussion of algebraic loops, equation sorters and various numerical schemes which detect algebraic loops and take appropriate actions accordingly is given in [19] and [20].

Another problem in simulation languages which operate on models in ODE form arises from the causal interaction in the block diagram structure, especially in tools such as Simulink where the model is represented in a graphical form. A graphical representation of a model in ODE form tends to show the *computational* structure of the model rather than the *physical* or the *conceptual* structure of the system. While the physical structure of a system is independent of the environment, this is not the case for the computational structure. A model of a system in ODE form, with defined inputs and outputs, is a specific representation of the model; even a minor change in the assumptions about the environment or about the model may cause the role of inputs and outputs to change. Furthermore, since there are few structuring mechanisms, it is difficult to document models in a way that they can be reused. The modelling procedure, therefore, is costly and difficult to repeat. Since simulation is used in different ways for feasibility studies and design support, the lack of good modelling tools means that different models of the same process may have to be developed for each use. Good models are difficult to develop and the modelling procedure is costly and time-consuming. It is therefore important that the modelling tools should facilitate reuse of models and model components.

Dynamic system models in control engineering come from many different disciplines such as process, electrical, aerospace and mechanical engineering. *Special purpose* modelling and simulation tools are used by engineers in these fields. For example, DIVA [50] and SpeedUp [78] are used for simulation of chemical reaction dynamics, SPICE [74] for simulation of analog electronic circuitry and PSS/E, for power system simulation. It is impossible for a control engineer to use these special purpose tools to model different parts of a complex system and then to merge the parts to make a model of the total system. Firstly, the tools are closed and rigid and it is not easy to interface them with each other. Secondly, considerable time and effort is required to achieve a degree of proficiency in using these tools. On the other hand it is unrealistic to expect engineers from other engineering disciplines to give up their preferred modelling methodologies and tools so that control engineers can have the block diagrams that their tools expect. Also, the user may deal with linear or nonlinear systems, continuous or discrete/sampled-data systems, systems with known or unknown parameters, the models may be self contained or hierarchically organized, and so on. Due to the abundance of different model types, the modelling tools must support a large variety of model representation structures as well as algorithms for general model interconnections. More powerful *general purpose* modelling methodologies are required in order to enable a unified approach for modelling and simulating all kinds of systems.

Two contenders for such an environment are the languages Dymola [27] and its successor Omola [70, 6]. Dymola is an example of an early modelling language that allowed non-causal sub-models and differential equations in a general implicit form. Originally Dymola was not object-oriented. However, it supported hierarchical modularization and abstract interfaces. A later, commercial version of Dymola has been extended in an object-oriented direction with a *model class* concept and inheritance. Omola also uses object-oriented techniques and equation-based models to model systems as a hierarchy of interconnected systems which are themselves described by DAEs. Omola, incorporates more powerful object-oriented techniques than Dymola. For example, it is possible to override an inherited definition in Omola but not in Dymola.

Both Omola and Dymola are an advance over CSSL type languages because they provide a break from an ODE formalism based on explicit cause-and-effect relationships. Acausal models are modular since they separate a system from its environment and allow abstract interfaces to be defined at an early stage in the modelling process without any apriori assumptions about the computational causality of the model. The object-oriented concept of generalization and specialization makes reuse of model definitions possible. Libraries of classes and sub-classes can be built which greatly enhance reusability.

Systems such as Omola and Dymola can be successfully used only if a large number of class libraries for different domains are provided. It is also important that these classes should be general enough and well-documented so that the user can easily select the class that best serves the purpose. There is no doubt a steep learning curve involved in using these systems, especially where users' own classes need to be defined. This approach is not very beneficial for modelling and simulating simple systems; but it pays off well in large systems where there is more scope for reusability. The real benefits are achieved if the user has a good understanding of applying object-oriented techniques and has the expertise to decompose the system into a class-hierarchy.

Example: Representation of an Electric Circuit

We now show through a simple example, the advantages of using a modelling and simulation language based on object-oriented methodologies to those based on ODE formalism such as ACSL and Simulink.

Consider a simple electric circuit as shown in figure 2.2. This circuit can be encoded in ACSL as:

```
Program Circuit
  Constant R1=100.0, R2=20.0, C=0.1E-6, L=1.5E-3
  Constant tmx = 0.01

  u = f(t)
  ic = uR1/R2
  uR2 = R2 * iL
  uC = INTEG(iC/C,0.0)
  iL = INTEG(uL/L,0.0)

  uR1 = u - uC
  uL = u - uR2
  i = iC + iL

  term(t.ge.tmx)
End
```

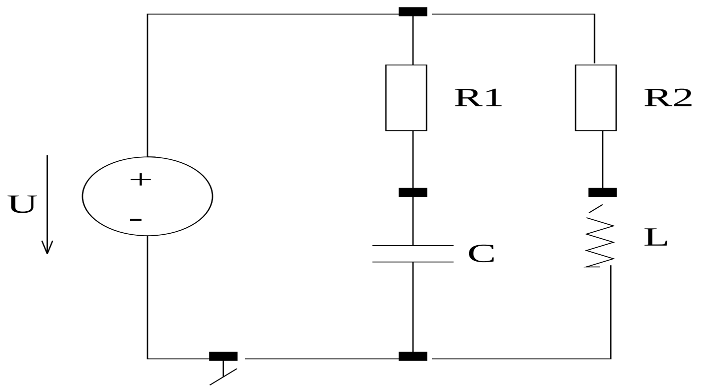

Figure 2.2: Simple electric circuit

The first equation describes the input to the circuit, i.e. the voltage source. The next four equations describe the circuit elements themselves, that is, objects in Omola sense. The last three equations describe the circuit topology, i.e. the interconnections between the objects (Kirchhoff's laws).

The Simulink model in block diagram form is shown in figure 2.3.

This circuit contains two objects of type resistor, and yet the equations describing these two objects look different. In the case of resistor R1, the voltage drop across the resistor causes a current flow, whereas in the case of the resistor R2, the current flow through the resistor causes a voltage drop. Clearly, these observed causalities are purely computational and have nothing to do with the physics of the problem. It would be preferable if objects

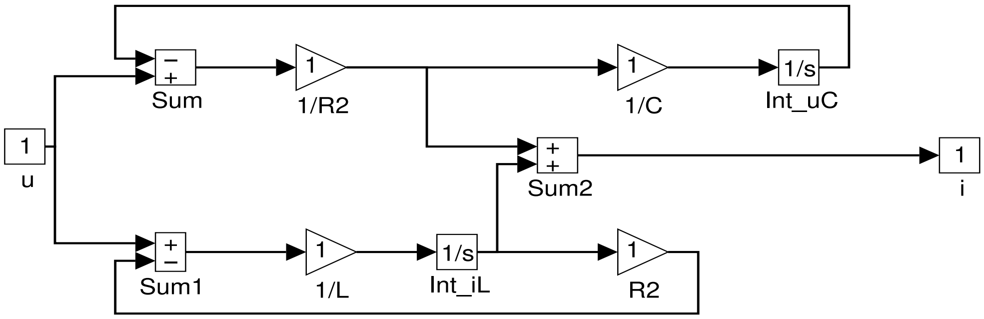

Figure 2.3: Simulink model of the electric circuit

such as resistors could be described definitively in terms of their physical properties and their interactions with the environment. In the case of the resistor, such an approach would call for a description of the resistor itself (Ohm's law) and a description of how this equation interacts with other equations of the neighbouring components (Kirchhoff's laws). Omola and Dymola both incorporate this concept. A brief description of how the electric circuit can be represented in Omola using a structured class-hierarchy of electric components is given below.

A predefined library of electric components in which a set of basic classes is defined is available in Omola. The classes in this library can be divided into three groups: terminals, base classes, and electric components. The most important terminal is the class called ElectricTerminal. It is a structured terminal with two components: V, which is a VoltageTerminal and I, which is a CurrentTerminal. An ElectricTerminal represents a single connection point of an electric component (for example, one end of a resistor). It has an electric potential, related to a system reference point, and an electric current flowing into the component. The convention used throughout this library is that all terminals have a current with a positive value if the current is flowing into the component. An important base class is called ElectricTwoPole. It is used as a super class for many electric components that have two electric terminals. The terminals are called T1 and T2. A property common to all electric two-poles is that the currents through the terminals are equal but are of different signs. This is expressed as an equation in ElectricTwoPole. ElectricTwoPole also defines two local variables, I and V. V is the voltage across the two-pole and I is current flowing from T1 to T2. The class ElectricTwoPole is defined as follows:

```
ElectricTwoPole ISA ElectricModel WITH
  %% Electrical component with two ports
```

```
  T1, T2 ISA ElectricTerminal;

  V, I type Real;
  %% I is current through component, into T1.
  %% V is voltage drop over component; T1 is positive.

  T1.I + T2.I = 0;
  V = T1.V - T2.V;
  I = T1.I;
END;


Resistor ISA ElectricTwoPole WITH
  %% Resistor with parameter resistance

  R ISA Parameter WITH
     default := 1.0;
  END;
  V = I*R;
END;
```

These classes are then called in a script file and their inter-relationships are defined for a specific circuit design. Alternatively, a graphical interface is provided in which the circuit can be drawn. No matter how a resistor is used it will always be represented as an object of class 'resistor', i.e. it does not depend on computational causalities. To summarize, in Omola models are declarative in the sense that they define facts and relations, rather than being procedures for computing data. For example, the following three models of a resistor are equivalent: $R \times I = U$, $R = U/I$ and $U/R = I$. The equation operator '=' is a symmetric equivalence operator; not an assignment as in some programming languages. Reusability of classes and clarity of code are other important advantages of a class-based approach. This approach is of course not suitable for all kinds of models. For example, models of information processing, like digital controllers, may be more conveniently represented as sequential procedures.

2.1.2 Control System Analysis and Design

In the late fifties and early sixties, the computers available to control systems designers were neither very powerful nor easy to program and interact with. They were used to automate difficult numerical calculations. Most of the work in designing a controller was still done manually. In the late sixties, batch-operation of large computers became widely available which made it feasible to use computers for tasks such as calculation of frequency responses, time responses, root loci and simulation. However, the programs could only be used on the specific hardware and software platforms for which they were developed. It was virtually impossible for different systems to share data and programs. Programs for solving the same kind of problem had to be written separately for each computing site. Furthermore, with batch processes it was not feasible to write interactive programs for experimenting with different algorithms and data. The situation improved with the introduction of terminal-access to mainframe computers which made it possible to write interactive programs with question-and-answer dialog interfaces that prompted the user for input and displayed the results on the screen. In the mid-seventies, several Fortran numerical subroutine libraries were developed, whereby Linpack [26] and Eispack [95, 32] became the basis for more specialized libraries for solving control problems. These libraries had common data structures and integrated the different numerical routines that were available at the time. Nonetheless, a main program had to be written in which these routines were called. This was often a difficult task since data structures had to be adjusted to make them compatible with those supported in the libraries. Experimenting with different algorithms or parameters mostly resulted in source code changes and re-compilations, which at that time were very time consuming. Many control designers found it easier to write their own subroutines.

In the mid-seventies, research efforts in CACE focused on finding an interaction style better suited to the requirements of control engineers, one that would include the control designer as an integral part of the design process. By then it became apparent that the wide range of users for CACE software—from novices to experts—all with different abilities and requirements, was an important factor in designing interactive software for CACE [8]. For experts, it is preferable to have a system where users retain the initiative and receive advice and help only on request. For novices, who need a lot of guidance, it is natural to have a system where the computer has the initiative and users are gently led towards a solution of their problem. This makes it possible for novice users to grow with the system and once they become proficient the initiative can be shifted from the computer to the users. The main interaction style at that time (mid-seventies) was *question-and-answer* dialog style. Gradually it became apparent that this style was too rigid and at times very frustrating for an experienced user because the computer, not the user is in command of the work. The payoff of these research efforts came in the form of packages that had a *command-driven* style of interaction similar to that in interpreted languages such as Basic and APL. The interpreted and interactive style of a command driven approach made experimenting with algorithms and data easier and saved the user from the time consuming compile-debug-execute cycle of traditional numerical processing languages such as Fortran. It also provided a way for the users to use the software according to their own ability. In fact Rosenbrock in late 1960's [88], suggested an approach similar to interpreted command-driven approach for control systems analysis and design but it was not till late seventies that such an approach was actually made possible by advances in computing technology.

Two relatively successful commercial programs that emerged in the late seventies as a result of research in command-driven style of interaction were CLADP (The Cambridge Linear Analysis and Design Programs) [62] and the set of packages for modelling analysis and design of control systems called IDPAC, MODPAC, SIMNON, SYNPAC and POLPAC which used a common package of subroutines called INTRAC for user interface, [8], developed at the Lund Institute of Technology. These interactive software environments explored the potential of interactive computing by combining an engineer's intuition with the power of interactive computing. In both packages operations were specified by *commands* whose names were easily identified with the desired operation. However, the two packages differed in terms of how the options were specified. In the packages developed at Lund, the options were specified by additional subcommand qualifiers given on the command line. In CLADP, where a question-and-answer sequence was used to specify options and to minimize the restrictions to the expert user, there was a type-ahead feature that allowed one to give the answers to several questions on the same line as the initial command. Both had a macro facility for grouping a series of commands. INTRAC also had commands for program flow control and for passing parameters which considerably enhanced the utility of macros. Such macros provided a convenient simplification of the command dialog. Users could extend the normal command structure by developing their own sequence of commands which were called by a single name in the same way as any other 'built-in' command. These packages went a long way in establishing the importance of the 'command-driven' approach and had a vital influence on the interaction style of subsequent CACE software.

One of the main features of the command-driven packages—similar to those developed at Lund which had a macro facility with commands for program flow control as well as parameter passing—was *extensibility*.

> Extensibility is a method that allows new primitives to be constructed out of existing ones. This is most adequately fulfilled by a highly structured command language containing the necessary flow-control elements (e.g., FOR, WHILE-loops, IF statements). It is then possible to use the new primitives as if they were baseline primitives. This is a unique view of programming in which the user thinks of programming as consisting of creating new 'words' in the language. Once a new word is created, it is used in the same way as a permanent word. It is an important feature of most interpreted languages such as Basic, Lisp, APL and Forth.

A well-designed CACE environment should have a minimum set of reliable baseline primitives, plus a mechanism for extensibility. However, any developer of such a command language has to make compromises. If the language is made too rich, the complexity of the system makes it hard to use. On the other hand, if too few language elements are included, the system will not be flexible enough to let the users solve all their problems. As no general CACE environment can include every conceivable control algorithm, users must be supplied with both a set of baseline primitives and an interface flexible enough to let them extend the environment according to their own needs. In other words, many users of CACE environments are capable of writing their own algorithms for use with data structures of their own creation; when this happens a good CACE facility should support them.

CLADP and the packages developed at Lund used complex and arbitrary data structures which were not generally transportable to other packages. Moreover, the subroutines in which they were defined and used were binary coded so the user could not know the structure of the different entities used in these packages. Most of these environments were therefore not fully extensible. They did what they were designed to do and very little more. Users could define their own macros but the facility was not sufficient to build toolboxes.

Matlab [67] gave users the facility to write their own functions, thus opening the field of add-on toolboxes for various domains. This was a major advance on CLADP and the packages developed at Lund. Matlab's ability to easily add toolboxes came mainly from its support of a single data structure, the 'complex matrix', which does not require dimensioning. This single data structure made it possible to easily extend the environment. User-defined functions could be used in the same way as 'built-in' functions. Furthermore, other users could use them without having to find the exact representation of data structures for different data elements. However, as we discuss later, this 'lack' of data structures has some disadvantages in terms of management of complex entities.

Matlab was originally written in Fortran as a convenient laboratory for computations involving matrices. Instead of its own algorithms (which are available in the present version), it provided interface to the Linpack and Eispack Fortran libraries. Although Matlab was not developed as a controls tool, but rather as an interactive calculator to be used in linear algebra, its matrix-based environment makes it suitable for describing many control algorithms particularly in the matrix-based state-space domain. The new Matlab program was written in C by *The MathWorks* and has evolved over the years with input from many users, in both academia and industry. In the current version the user interface is mainly command-driven, but more elaborate graphics facilities are also available. The number of algorithms it supports is very large and is increasing with every toolbox appearing on the market. The scripting language performs matrix algebra naturally and provides a good programming environment for writing functions, script files and toolboxes. The simple programming environment makes it possible for toolboxes to be written by experts in their respective domains even though they might not be trained software engineers. This ensures good quality algorithms. The users can easily go through the algorithms (which are in the scripting language) to see what they do and modify or extend them according to their own needs. Matlab has acquired the status of a de-facto standard due to its powerful 'expressiveness' of the command language, natural support for matrix algebra and language extensibility.

> Several programming languages have addressed the issue of language syntax. An interesting example in this regard is the programming language APL [23]. The language syntax of APL does not have any keywords. Instead it uses short 'non-verbal' symbols. The basic philosophy behind this is that too much text (key words) obscures a program. Some examples of how APL uses 'symbols' to perform operations are:
>
> - $\Box \leftarrow 14$, output the number 14;
> - (a)O, multiply by π;
> - (b)$*$, exponentiate;
> - (c)$\otimes$, logarithm to base e.
>
> Some of the symbols can be combined into new compound symbols with a different meaning. But such a syntax can be too cumbersome for complex numerical calculations. It can make a program even more obscure than if keywords are used (which goes against its basic design philosophy). To work with APL, a special keyboard is required which takes into account the different 'symbols' and is significantly different from conventional keyboards. ALGOL 68 [71] is another example of a language that uses 'symbolism'. But it uses more 'natural' mnemonics to those used in APL, for example, pi, sin, log, etc.

Matlab is frequently criticized for its poor support for representing complex structured entities and type checking. Some research groups believe that a wide range of data structures is required to deal naturally with control problems and that it is not sufficient to just have matrices. They believe that control entities should be represented by data structures at the same level of abstraction as their appearance in theories and algorithms. Control engineers use entities such as polynomial matrices, transfer functions and linear system descriptions. For example, in control system design one has to deal with networks of systems, some of which may be represented by sets of state-space matrices, others by frequency responses, and others by simulation programs. It is also necessary to represent signals, performance indices, design specifications and constraints, and so on. The main advantage of defining a data structure for a control entity is that if an algorithm needs a whole system—including various representations, comments, experimental conditions, and so on—this can be passed as a single entity. Moreover, data structure design keeps a 'fence' around the data collection so that pieces of it are not changed by accident. In a matrix-based environment, all entities are represented as matrices so it becomes very difficult to keep track of the different kinds of data. It is common to confuse one matrix for another or to accidently overwrite or change the matrix, which gives rise to meaningless and misleading results. The argument against providing more data structures is that they limit extensibility, because in order to write functions users would need to know the exact representation of different data structures. An agreement on standard data structures for control engineering entities can help retain the feature of extensibility in tools where compound data structures or user-defined data structures are used.

Some Matlab toolboxes use 'packed representation', whereby several matrices are grouped together as one matrix. This form is used in the SYSTEM matrix in μ-Analysis and Synthesis Toolbox [11], MVFR matrix in Multivariable Frequency Domain Toolbox [14] and theta matrix in System Identification Toolbox [93] for Matlab. The advantage of this approach is that it provides a relatively easy way to group related data. However, it adds an index coding and de-coding overhead. Also, data related to different matrices is not distinguishable inside a packed matrix.

There are several academic and commercial CACE environments which are extensions of Matlab. CTRL-C [51] and Matrix$_x$ [108] are early examples of CACE environments which extended the capabilities of the original Matlab. CTRL-C extended Matlab with numerous control analysis and design algorithms. It was based on the proposition that a matrix environment can lead to a unified, versatile software environment for related fields in control systems analysis and design, such as matrix analysis, engineering graphics and digital signal processing; a unified approach to these disciplines was possible through matrices because they are a common object in these fields. Matrix$_x$ extended the original Matlab environment to provide support for the complete control design cycle. In these environments the primary focus was on the provision of better numerical support for control systems analysis and design than that available at the time in Matlab.

Some CACE environments developed during the mid-eighties and onwards (such as IMPACT [87], Caesy [109], Ψlab [42], Xmath [110] and more recently ControlH [28]) aim at extending the Matlab environment with *control-oriented data structures*. They support Matlab's complex matrix type as well as several other special matrix types and objects which are particularly well-suited for control engineering. Most of these environments use some kind of object-oriented features. Object-oriented features provide a close correspondence between programming constructs and their counterparts in the modelled realm. Object-orientation has particular importance for systems in which the end-user has more freedom in defining computational tasks. In this context, it is often stressed that inheritance and encapsulation, both of which are attributes of object-oriented programming, enhance the consistency of user code and make its misinterpretation less likely.

The scripting languages of both Ψlab and Xmath (Xmath is part of a bigger integrated system called the Matrix$_x$ family of products) have several common features mostly based on object-oriented concepts. In both, many constructs for storing and manipulating data are similar to Matlab, but they both extend the amount of information stored with a given object, and the number of actions a command or function can take depending on the type of data passed.

Each Xmath data object is an instance of a class which defines its structure. The classes are arranged in a hierarchy so that subclasses inherit characteristics from a superclass and have additional special characteristics of their own. For example, the 'matrix' class is very general consisting of all matrices of any shape. The 'square matrix' class is a specific kind of matrix class that requires an equal number of rows and columns, but otherwise *inherits* the characteristics of the general matrix class. A class is either *primitive* or *complex*. A primitive class contains only one kind of data. Therefore, a matrix class and all its subclasses are primitive. A complex class combines different kinds of objects, e.g. dynamic systems, transfer functions, lists etc. Ψlab also supports control related data structures (for linear state-space system and transfer function representation) in an object-oriented fashion and stores them as *lists* (which are described later). The object-oriented structure provides significant benefits in terms of simplified data management and natural syntax. For example, once a state-space system is created, it is stored in a single variable and all constituent variables composing the state-space system can be deleted. The special properties of each object are internally recognized, so the operations performed on these objects are more intuitive. For example, it is more natural to multiply two polynomials by typing p1×p2 than it is to call conv(p1,p2) as is done in Matlab. The operator '∗' can be overloaded as:

- Matrix multiplication: matrix1∗matrix2.
- Multiplication of two polynomials: poly1∗poly2.
- State-space systems in series: syst1∗syst2.
- Time-domain simulation: syst∗timehist.

Similarly, functions can be overloaded. For example, freq(G) calculates frequency responses of a system G which is either in state-space form or transfer function form.

An important data type supported by both environments is the *list* data type which allows a collection of data objects (not necessarily of the same type) which can be other lists, functions, and libraries of functions. Lists are useful for defining abstract data types. In Ψlab linear systems are treated as lists. Once a linear system is created, it can be combined with other linear systems or the transfer function representation can be obtained by treating the linear system as an abstract data type. The basic system which is used for defining linear systems is `syslin`. This function takes in constant matrices which define a linear system either in state-space form or in transfer function form. To be more specific the calling sequence of `syslin` is either:

```
S1= syslin('dom',A,B,C,D)
```

or

```
S1=syslin('dom',trmat)
```

`dom` is one of the character strings 'c' for continuous time or 'd' for discrete time systems. `trmat` is a rational matrix, i.e. it is defined as a matrix of rationals (ratios of polynomials). `S1` is an abstract data object of type list and represents a linear system. Once `S1` is defined, the matrices `A`, `B`, `C`, `D`, and `trmat` can be deleted.

In both Xmath and Ψlab with the notable exception of the list type, there are no direct ways of creating user-defined objects. Both environments support symbolic manipulation of polynomial expressions. Xmath has certain other useful features, for example: *PDMs*—parameter dependent matrices which store a matrix as a function of an independent parameter (the domain) which is often a physical parameter such as time, frequency, speed, etc.; *partitions*—a means of controlling the local scope of variables; and *optional keywords* which enhance flexibility of function calls.

Caesy [109] (Computer-Aided Engineering SYstem) provides a strongly typed programming environment (which resembles Ada) for development of algorithms and software for computer-aided control system design. Caesy includes capabilities of Matlab-like environments, provides specialized structures for computations on system models and also supports user defined data structures. While designing the language, a syntax was developed that could be parsed by parsers such as those generated by the UNIX utility YACC. The YACC implementation makes it easy to change the syntax of the language.

IMPACT was both implemented in, and also resembles the language structure of, Ada. It was one of the first environments to show the benefits of implementing numerical environments in a more structured programming language. Considerable benefits have been achieved since the implementation of CACE environments in highly structured computer languages instead of in a language like Fortran. Highly structured languages have better exception handling features for both user-errors such as erroneous interactive input sequences and system/programming errors such as division by zero or array index out of range. In particular, the functionality of command language expression parsers has improved significantly since the high-level language feature of 'recursion' has been used in their implementation.

Since the user input and any other functions described in the command language have to be interpreted rather than compiled, the execution of complex algorithms described by the command language is rather slow. But this time is compensated for by the time not spent on compilations during the development of new algorithms (as would be required in a compiled language). Here again, IMPACT [85] showed that if the command language is similar to or derived from the implementation language, it is possible to obtain shorter execution time by compiling and incorporating any complex algorithm into the command language itself as soon as it is developed and tested. If the implementation language is similar to the command language, such a transition can be made with a minimum of recoding. This feature is also provided in Matlab in the form of Mex-files which are C-coded Matlab function files. A drawback of this feature is that the algorithm code is not readable by the user (since it would be a binary-coded executable file).

Matlab has evolved over the years; the recent additions and upgrades are influenced by object-oriented programming ideas—notably Simulink and Handle Graphics. These have greatly enhanced the capabilities of the Matlab programming environment. More recently, graphical solutions for domain specific applications or for specific techniques for the analysis and design of control systems have become popular. Examples of some Matlab toolboxes that have been developed to provide such solutions are the System Identification (SID) Toolbox [93], the Nonlinear Control Design (NCD) Toolbox [75] and the Multivariable Frequency Domain (MFD) Toolbox [14]. These toolboxes are written entirely as M-files using Matlab Handle Graphics so the user interface and the routines can be modified to add new functionalities. This has proved to be a popular alternative style to the command-driven interface when it comes to direct manipulation and visualization of data.

SID builds on Matlab to provide tools for creating mathematical models for dynamical systems based on observed input/output data. Its graphical user interface (GUI) helps in the organization of data and models and presents a road map of the system identification process. The process begins by importing data, proceeds through analyzing and preprocessing, and culminates in generating models from the data and analyzing the model properties. Data sets are represented by icons which the user can drag-and-drop. Data and model analysis operations are performed by checking the appropriate check-boxes.

NCD Toolbox operates with Matlab and Simulink to provide a time domain-based optimization technique for designing linear and nonlinear control systems. Controller designs are developed as block diagram models in Simulink. Users can select a set of tunable model parameters and can graphically place time-response constraints (by moving boundaries with the mouse on a time vs. response graph) on selected output signals. Successive simulation and optimization methods are applied automatically, thereby tuning the selected model parameters. The toolbox uses the graphical time-response boundaries and selected output signals to automatically formulate a constrained optimization problem.

The approach used in these toolboxes solves the problem of choosing from a large number of algorithms available by guiding the user through the design process.

Many problems can be solved using a matrix language like Matlab or its derivatives Xmath, Ψlab, CTRL-C and Impact. They provide an intuitive environment for control design experimentation. However, experimenting with different algorithms and parameters is not simple in these environments when it comes to managing the large amount of information that is generated during any control systems analysis and design session. What these environments lack are proper tools for integrating database management into the design process itself and for managing the design history of a project. Some environments that provide some degree of support in this regard are discussed in the next section. The database management aspects are covered in more detail in the next chapter.

2.1.3 Integrated CACE Environments

No single program can incorporate all stages of control systems analysis and design without adding to the size and complexity of the program. Several research projects in CACE have concentrated on developing integrated environments of different tools and packages.

The set of packages developed at Lund was the first to provide an integrating mechanism in the form of the package INTRAC which provided a common interface to other packages called IDPAC, MODPAC, SIMNON, SYNPAC and POLPAC (each of these served different numerical purposes). This was fairly simple given that the packages it integrated were developed at the same site. A more challenging project was the Federated Computer-Aided Control Design System [96] which tied together several independently developed packages through a unified database. It integrated the following packages: CLADP for multivariable control system design by frequency domain methods, IDPAC for system identification, SIMNON for non-linear simulation and SSDP (State-Space Design Package) for multivariable control system design by state-space and time domain methods.

These early attempts at integration were mainly concerned with providing a unified system for numerical and algorithmic support in the control design cycle. The integration facility provided in INTRAC was only at the user interface level. The central database in the Federated Computer-Aided Control Design System only took care of the inconsistencies in data formats in the core packages and offered no other functionality for tasks expected from a database management system, such as keeping a record of the design history of projects. Some later projects on integrated environments such as ECSTASY (Environment for Control System Theory And Synthesis) [72], GE-MEAD (Multi-disciplinary Expert-aided Analysis and Design) [107] and ANDECS (Analysis and Design of Control Systems) [37] incorporated enhanced database management functionalities (less so in ECSTASY) for supporting data relations and project design histories while taking maximum advantage of existing software modules.

ECSTASY integrated Matlab, ACSL, PA Set Tools (a commercially available software engineering tool-set which provided routines for user interface), a database based on the relational database model, and some document preparation tools. The database facilities in ECSTASY allowed a variety of primitive data types to be defined, and provided a range of operations such as creation, deletion, renaming, and copying of data entities. All data entities within the database had time stamps, access protection and relational pointers associated with them. The database acted as a consistent central store for all data required by the other major facilities in the infrastructure and also for the data required by the design, synthesis and identification tools embedded in the infrastructure.

GE-MEAD integrated the following tools:

- Pro-Matlab and Matrix$_x$ for linear analysis and design;
- SIMNON for non-linear simulation;
- GE Computer/Human Interface Development Environment (CHIDE) which combined a point-and-click menu and forms-driven interface with other access modes for the more experienced user;
- ROSE a database management system;
- DEC Code Management System (CMS) a database manager which organized models and corresponding results and other related data elements in terms of projects, and also included a version control mechanism; and
- GE Delphi expert system shell which performed routine higher-level CACE tasks.

The database management framework of GE-MEAD was the first (except for an early proposal in [54]) to address the problems of maintaining user models and analysis and design results. In other environments, user models and results simply accumulated in the workspace. It was up to the engineer to perform version control, to relate results to specific model instances and conditions, to relate linearized results to parent non-linear model and operating points and so on. The user interface in GE-MEAD provided a unified interface for all core CACE tools, with as little as possible left to user's memory. For example, the framework for simulation was identical whether using Pro-Matlab or SIMNON. The user therefore did not have to learn all the intricacies involved in using each tool.

The design philosophy in ECSTASY and GE-MEAD focused on providing high-level support while taking maximum advantage of existing software modules. They were both based on the belief a single program capable of accomplishing all tasks performed by a control engineer will probably never exist, so it is best to shell the existing CACE software in an open software environment rather than to commit to a specific set of data structures and algorithms. But neither of these projects was technically successful. As pointed out in [106, 73], they both failed because of the high maintenance costs of the links to constantly changing external packages. These systems became very large and created serious problems both in the maintenance and update areas for developers and in the usability areas for users. There were problems in fully encapsulating stand-alone packages within a framework. Error messages that were reported by the packages had to be interpreted by the shell and displayed to the user using the language of the framework. There were documentation problems: user manuals, application notes, and on-line help systems for the integrated interface had to be written and these did not always keep pace with the introduction of new features and new versions of tools. Even though these integrated environments were based on the idea of a central repository where data could be stored in a standard format and on the use of existing software to provide the user with a unified system spanning the entire control design problem, the user was still aware of working in distinct individual packages and had little help other than that of permanent storage of data in the core environment. The costs of GE-MEAD and ECSTASY was high due to the price of the proprietary software used in the systems. Another major disadvantage of these shells was that developers continued to extend the capabilities of the core packages. In the case of Ecstasy, this happened several times during the development of the shell and caused considerable problems. In fact, the correct implementation and maintenance of the shell and its documentation, which depended on factors beyond the control of the integrators, became an impossible task.

An example of a fully operational (only at the site of development though) integrated environment is ANDECS. It is a multi-objective design automation system which supports computational experimenting in control synthesis, dynamic system simulation, analysis and optimization. ANDECS makes use of the 'engineering operating system' RSYST, an engineering database system and a user interface with macro facilities which can be driven either by graphical or command inputs. It integrates the following modules: CONTROL (Linear systems analysis and synthesis), MATHLAB (Matlab and mathematical utilities), SIMULATE (Dynamic systems simulation), GRAPHICS (Interactive result visualization), and OPTIMIZE (Multi-objective parameter and trajectory optimization within a design environment, where the design history is recorded on an automatically evolving database, for backtracking as well as for branching of the design process). ANDECS was developed at a research lab (German Aerospace Research Lab) over a number of years and it may not be wrong to say that it is perhaps the most sophisticated integrated project of its kind. The functionality of ANDECS, especially in the context of computational experimenting and database management, is discussed in detail in the next chapter.

The Matrix$_x$ [68] family of products is another example of an integrated set of CAE tools that aid engineers in the design, simulation and implementation of control systems. It integrates the following modules: Xmath for mathematical analysis and visualization, SystemBuild for graphical modelling and simulation of nonlinear dynamic systems, AutoCode for automatically generating C or Ada code from SystemBuild models and DocumentIt for generating detailed documentation for the models. The constituent tools of Matrix$_x$ are all developed by the same company and so there is no need for an integrating mechanism for differences in data formats as is required in environments which integrate tools developed by different companies. In Matrix$_x$, all functions are available within a single design environment backed by common data structures.

Matlab can also be regarded as an integrated environment. As discussed earlier, one of the most attractive features of Matlab is language extensibility. This feature has led to the development of several Matlab toolboxes, covering fields such as control systems analysis and design, identification, optimization and signal processing. The ability to interface directly with Fortran and C subroutines bridges the gap between Matlab and the physical computing system, enabling the Matlab environment to communicate with, for example, data acquisition and signal processing platforms, parallel platforms and graphical devices. With its set of toolboxes and the ability to interface with languages such as Fortran and C, Matlab forms its own integrated environment.

Both Matrix$_x$ and Matlab are examples of single-vendor integration. Although this approach is quite effective, the closed architecture of such environments precludes easy addition of tools from other vendors.

2.2 Requirements Analysis

In this section, some important user requirements are listed. These requirements have been put together from different sources such as [6, 13, 84, 86, 103]. Most of the requirements have already been discussed in section 2.1. Here they are summarized and listed separately as requirements for modelling and simulation, programming language and database management support. These three categories were chosen primarily because of the difference in the nature of their respective user requirements. The order in which these requirements appear within each category does not depend on their importance, though some are more important than the others. In fact, some requirements can be classified as 'essential' and others as 'useful'.

User requirements specific to *modelling and simulation languages* are as follows:

- Support for the most commonly used mathematical and logical frameworks for representing model behaviour.
- Reusability of parts of models in other models.

- Documentation of models for enhanced clarity, maintainability and reusability.
- Standardization of model representations to facilitate exchange of models between different tools.
- Direct graphical manipulation such as model diagram editors.
- Facilities for structuring large models in a hierarchy and for defining standard classes for modelling general dynamic systems.

User requirements specific to a CACE *programming environment* are as follows:

- Natural support for numerical processing: The language syntax should provide natural support for numerical processing especially for matrix manipulation. The need to manipulate on matrix expressions follows from the heavy use of these objects in engineering analysis.
- Interactive environment: This is essential to enable easy experimenting with different data sets, parameters and algorithms. The requirements for user interaction support are closely related to the iterative nature of the design process. Most existing CACE tools support sequential interaction style. As pointed out in [84], there is a need for an asynchronous interaction style, where many tasks are at the disposal of the user at the same time and sequencing within one task is independent of sequencing within the other. These include mechanisms for inter-leavability, concurrency, reversibility and repetitiveness.
- Interpreted environment: The programming environment should be based on an interpreter rather than a compiler. An interpreted environment is better suited for interactive computations which in turn allow easy experimenting.
- Extensibility and configuration: It should be possible to easily add a new function to the environment, extend the command language, reconfigure the graphical user interface and perform customization without recompiling the whole system.
- Symbolic expressions: This feature is particularly useful for polynomial manipulations.
- User defined functions and data structures: Users should be allowed to define their own functions and data structures and to use them like any other 'built-in' function or data structure.
- Function overloading: The language should allow for the overloading of functions. Overloading should be based on the number of arguments and their types. That is, the environment should have the capability to distinguish between two functions with the same name but different argument types.

- Operator overloading: The language should also provide for the overloading of operators such as $+$, $-$, $*$ and $/$. The user should also be allowed to use these operators with user defined objects (transfer functions etc.).
- Program control constructs: Common program control constructs such as looping, branching, etc. should be provided.
- Batch processing or user defined script files: The environment should support batch processing of commands. In modern CACE environments, this feature is supported in the form of script files.
- Support for graphics: It is important that the environment should support presentation and visualization of data in 2-D and 3-D graphics.
- Integration: Tools should be able to communicate with each other. Data produced by one tool should be processed easily by another.
- Exception handling features: It is important that a program should not abort in case of an interactive input error and/or program error or division by zero since it would result in the loss of all intermediate results. Environments implemented in highly structured languages are good at exception handling.

User requirements specific to *database management support* (these issues are discussed in detail in chapter 3) in CACE environments are as follows:

- Compound data structures: Most CACE use very simple data types which makes it difficult to handle complex design and analysis problems. Compound data structures should be provided to overcome this problem.
- Control-oriented data structures: CACE environments should support data structures for control entities, for example for state-space representation, transfer function representation, frequency responses, etc.
- Semantic links in a database: One of the major problems in control systems design is keeping track of the information generated during control systems analysis and design, especially when the iterative nature of the design process is taken into account. Minor modifications of an existing design may require the modification of many items of information. These range from apparently trivial items such as names of files containing system descriptions, to items which can only be obtained after considerable computation, such as time responses and assessments of stability. It is important to have a well-defined database for storing and retrieving design data.
- Project management and version management: This is concerned with the management of a database of experiment histories over the life time of a project. Essentially,

a project management system keeps track of the various models and results corresponding to a specific project, and incorporates some kind of version management capabilities.

- Data sharing: This is an important requirement for collaborative engineering. It also gives rise to the need for managing long transactions.
- Large data files: There is also a need for a mechanism for efficient storage and retrieval of large data files.

Furthermore, with the introduction of sophisticated graphical user interfaces (GUIs), there is a growing demand for *direct graphical manipulation and presentation* of dynamic data. GUIs give tools a certain degree of intelligence by displaying highlighted menus, buttons and plot handles which guide users through different procedures.

2.3 Users of CACE Environments

Users of CACE environments can be classified in several ways. A simple classification is that of *expert* and *novice* users. The requirements of these two types of users differ when it comes to user interaction support. As we saw earlier, the emergence of command-driven mode of interaction along with facilities for writing macros and function files were based on the requirements of these two different kinds of users. Another simple classification is to think of users as being based either in *industry* or in *academia*. The impact of this classification goes beyond user interaction support. The projects undertaken in industry require more support for project management, database management, documentation and design history. Academics mostly use these tools for testing new ideas, techniques, theories and algorithms, whereas in industry the main focus is on using already existing technology for implementing real-time controllers. Also, in industry, there are teams of engineers working on a project and relatively few projects going on at any one time. Due to this difference in requirements a product like Matrix$_x$, which has better design support facilities, has a larger customer base in industry than Matlab, which is oriented more towards academics. Also, the powerful command language and natural matrix algebra support of Matlab make it an ideal environment for teaching and this is one of the reasons for its widely acclaimed success in academia.

These two simple classifications have had no doubt a great influence on development of CACE software, but modern applications of control theory and the different purposes/tasks for which CACE software is used these days require a broader classification of its users. One such classification is given in [83], where three distinct users of CACE environments are identified as *algorithm designers* (extenders), *software developers* (implementors) and *end-users*. Extenders are numerical analysts or domain specific theorists who usually develop prototype systems to test algorithms, abstract formulations and designs. Implementors are computer scientists and software engineers who implement these concepts; their main concerns are system performance and efficiency as well as reliability of the code. The end-users are control engineers who use these environments for specific design problems.

The use of the word 'distinct' in such a classification is potentially misleading since there is usually an overlap in the roles played by these different groups. Companies which specialize in software for control engineering employ control researchers in addition to professional programmers. The control researchers in these companies contribute substantially not only to algorithm development but also to the actual implementation. In fact, many commercial products were 'born' in academic centres and were developed not by software engineers but by control researchers. However, the current software technology is becoming increasingly specialized, which makes it almost impossible for control engineers to write full-grown commercial quality software by themselves.

Popular CACE products such as Matlab and Matrix$_x$ provide functionality for all three kinds of users. For end-users, they provide tools for designing and analyzing signals and systems; for extenders, they provide sophisticated visual data analysis capabilities and an extensive library of numerical routines; and for implementors, they provide platform-independent environment for custom math and graphic applications with sophisticated custom user interfaces. Therefore, the same environment caters for different users with different demands. But, as stated earlier, Matlab is more popular in the academia and Matrix$_x$ is more popular in industry.

Thus there are various ways of looking at the different types of users of a specific product and the different types of products popular with a certain group of users. Most of the discussion in this thesis, from users perspective, concerns control systems analysis and design researchers including both extenders and end-users.

2.4 Summary

Advanced computer technology has now made it possible to develop interactive, extensible and flexible integrated software environments, something unthinkable at the time when mainframes were used. Early CACE products closely prescribed what a user (control engineer) was *allowed to do* and generally had a rather narrow model of what users would *actually like to do*. Currently there are products available such as Matlab, in which the need for flexibility is seen as a paramount design aim to the extent that the software is like a programming environment with some carefully chosen kernel algorithms, which can be used to develop other algorithms. Expressiveness of the command language is another important feature in most present CACE environments especially when it comes to matrix manipulation. But what these environments lack is proper database management support for the control design cycle (this aspect is discussed in detail in chapter 3).

There is a growing interest towards integrating tools and packages for different stages in the control design cycle and support tools for database management and expert systems. Impediments to the construction of integrated environment of tools within the newer generation of CACE products include the complexity of data which has to be passed from one tool to another and the complete lack of data structuring and type checking facilities. As long as there is no clear accepted standard for the dynamical exchange of data between separate programs, all attempts at integration are more or less temporary. (These issues are also discussed in detail in chapter 3).

Chapter 3

Database Management Aspects in CACE

Before the mid 1980s, database management support in CACE environments was at a very rudimentary level. Primary emphasis was mostly on core CACE activities such as non-linear simulation, linear analysis and design in the frequency and time domains and model identification. From the mid 80s onwards, several research efforts in the field of requirements analysis for CACE such as [20, 36, 24, 53, 56, 64, 104, 107] identified a role for database management facility of some kind. Considerable research in database management aspects has also been carried out in other branches of engineering, such as software engineering, VLSI, ECAD and mechanical design [38]. In this chapter, an attempt has been made to trace the progress made so far in database management support in CACE environments, the directions that various research groups have taken while addressing this issue, and the different levels of database support that these groups have thought important for CACE.

Organization of the large amount of data and intermediate results produced by iterative step-wise design procedures, and management of inconsistencies in data formats of various tools that are collectively used in an integrated environment for analysis and design of control systems, are the two main database management problems in CACE. The three different aspects of database organization discussed in the chapter are: *support for compound data structures*, *data models for control engineering* and *database organization based on project management*. This is followed by a discussion on *databases in integrated CACE environments* and *data interchange between software tools*. Some possible architectures for future integrated CACE environments are also discussed. The chapter is divided into three sections. The first section on database management covers the general database organization aspects related to CACE. In the second section, the software architectures for database support in integrated CACE environments and data standardization efforts for data interchange between software tools are discussed. The final section summarizes the main points raised in the chapter.

3.1 Database Management

Database management in CACE tools is most often based on storage. For example, data for different projects may be kept in different subdirectories or tapes. Data files for a project are then distinguished by assigning meaningful filenames. Some packages provide additional functionality such as tagging data files of different types by using different extensions. For example, CLADP generated filenames by appending characters to a user-supplied 'run name'. All such support is very rudimentary and leaves it entirely up to the user to maintain the integrity of the database.

> **An Example of the CACE Database Management Problem:** The magnitude of the control engineer's database management problem can be appreciated by an example given in [105], which shows how in many industrial applications, control engineering activity develops a substantial database. In flight control, a typical database may contain one non-linear airframe model, 20 linearized models (corresponding to 20 points in the flight envelope), 20 linear control system designs, one or several candidate non-linear (full-envelope) control systems, and innumerable time-histories and analysis results (step responses, eigen-values, frequency responses, root-locus data, singular values etc.). This database picture may be multiplied many times over by two additional factors. Firstly, in a multi-disciplinary application such as integrated flight and propulsion control, one can produce an enormous database by combining 20 flight regimes with 9 engine operating conditions, for example. Secondly, it is not unusual for the primary non-linear model of the controlled object to be modified several times in the course of the complete analysis and design cycle. We can thus 'size' the database management problem by assuming 3 instances of each model and perhaps 6 results per non-linear or linear model on the average, giving us the final product $3 \times 6 \times (1 + 20 + 20 + 2 + 1 + 9 + 9 + 1) = 1134$ files.

The above example shows that CACE project activity can generate hundreds of results and model files, of which the user may wish to retain and manage a substantial percentage. This may not be a large database in terms of commercial database management systems, but it is neither small nor simple, and is thus difficult for most human users to manage effectively without support. The large number of files generated in the complete design cycle and the relations between these files can be very difficult to comprehend and manage. In order to overcome this problem, information associated with a design should be organized as a formally defined database.

The starting point for such a formally defined database is to identify the different entities used in the process of control systems analysis and design and to define appropriate data structures for them. Control engineers work with networks of systems, some of which may be represented by sets of state-space matrices, others by frequency responses and others by simulation programs. They also need to operate on other entities such as signals, performance indices, design specifications and constraints. If proper data structures are defined for these entities, the control engineers would then be able to represent and manipulate them in a database at the same level of abstraction as their appearance in theories and algorithms. For example, if one algorithm needs a whole system, including various representations, comments, experimental conditions, and so on, this could be passed as a single entity; if another module requires a single matrix extracted from that system, it could be passed just that matrix. Considerable efforts have been made to identify appropriate entities which need to be represented for control engineering purposes, such as [45] which primarily identifies a set drawn from the needs of linear systems analysis and [4] which provides examples of the representation of nonlinear models.

Early software such as Simnon [8] and CLADP [62] defined data models which made use of such system theoretic objects. Simnon distinguished between entities such as 'continuous system' and 'connecting system', while CLADP distinguished between 'state-space models', 'transfer function matrices' and others. These facilities gave rise to some sophisticated features. For instance, a CLADP user could connect in series systems such as constant gains, state-space models or transfer function matrices using a single command without knowing the internal representation of each system.

Conversely, Matlab, one of the most popular CACSD software package available today has just one data structure: the complex matrix. In Matlab, the user must know how each system is defined because either different functions must be applied (by the user) or the same function must be applied with different arguments. An explanation of why Matlab is more popular than software such as Simnon and CLADP, in spite of its having just one data type, is provided in [56], which summarizes a comparison made between these environments:

> It is not just the definition and provision of sophisticated data structures which is necessary for CACE software. The data structures are valuable only if they are exploited in the construction of a powerful software environment. Most users apparently prefer to have the flexibility of the software tools provided by the Matlab approach, even without data typing, to the convenience of a package like CLADP, which is provided at the cost of inflexibility. As far as

the user is concerned, the structures provided in CLADP are not really data types because they are not made available to the user for programming or querying.

The success of the flexible Matlab environment led to the development of several software environments, the so-called ‘Matlab plus data structures’. For example, Xmath [110], CTRL-C [22], Robust Control Toolbox for Matlab [21] and μ–Analysis and Synthesis Toolbox for Matlab [11]. These environments give the users the flexibility of the Matlab environment but the data structures in these tools cannot be modified and there is no facility for user-defined data structures, so they are only suitable for representing simple structures. Some languages such as Impact [87] and Basile (now called Ψlab) provide a larger set of data structures. Impact also allows user-defined data structures. This is an enhancement to earlier products since the user can define and modify data structures.

As proposed in [54], a useful way of defining a database is to view it as a ‘simulation’ of an organization or of an activity. For example, a conventional commercial database ‘simulates’ a corporation in the sense that an interrogation of the database yields information about its current state: how many orders it has, the names of its employees, what its products are etc. Similarly, a control engineering database can ‘simulate’ the process of designing a control system. By interrogating it, one can see the current state of progress: which design method is being used, whether stability has been achieved, what the step response is, etc.

In most commercial applications, a typical database contains many items of information whose interrelationships are rather simple. By contrast, a database which is to ‘simulate’ a control design process needs to contain relatively few items (because separate design projects are associated with separate databases), but the interrelationships between the items can be very complex. In particular, so-called *semantic constraints*, such as ‘the frequency response is defined only if the system is linear and time-invariant’, and *associations*, such as ‘a particular state-space model is equivalent to a specific interconnection of systems’ may hold between various objects in the database. The database management system should ‘know’ about as many of these constraints and associations as possible, since this removes much of the burden of maintaining database consistency from the human user. In present CACE environments, it is most often left to the control engineer to decide how to keep track of such constraints and associations, and as pointed out in [54], most of the efforts for defining compound data structures and control specific abstract data types allow for the alternative representation of systems but do not support the association of alternative representations of the same system. Therefore, a data organization facility is required that would keep track of the data and the relations (as outlined above) between them. In other words, in addition to data structures, a data organization facility is needed which ‘understands’ these data structures.

The usual models of databases (i.e. network, hierarchical and relational) do not have the power to represent this complexity in a natural way. The focus of traditional database management systems remains on the problems of efficient retrieval of relatively few data types from extremely large databases. In CACE (and other branches of CAE), the problem is the management of a changing population of complex data types in a relatively small database. Traditional commercially available databases are therefore not suited for control engineering purposes. What is required is a database which can cope naturally and efficiently with complex structured data.

Semantic hierarchical data models that can represent such complex structured data were proposed for CACE in [55] and [9]. These data models attached great importance to incorporating semantic information in the database and the ability to change the data model dynamically by defining new attributes for a system or creating system entities of a possibly higher level. The hierarchical data model proposed in [55] was later implemented in a functional programming environment [16], an object-oriented programming environment [79] and a logic programming environment [101]. The principal idea behind all these implementations was to provide a full programming environment in addition to data management facilities. One of the most important features in these environments was the ability to define new data types and procedures as well as semantic constraints between different data types and procedures. The functional programming based implementation exploited the functional programming feature that does not make any distinction between 'functions' and 'data' to define semantic constraints between complex data structures and functions. For example, it made no distinction between tools (functions) for handling data and tools (functions) for calculating, say, stability margins [56]. The object-oriented implementation made use of 'class mechanism' with full inheritance of properties and methods from general to specific classes for defining new data structures and methods. The logic programming implementation, in addition to providing the features in the earlier implementations, concentrated more on the declarative style of interaction for querying and updating a semantic database for control engineering. The semantic hierarchical data model proposed in [9] was implemented in ExperLisp, which has limited support for object-oriented programming, and the implementation was therefore impeded by a lack of proper inheritance mechanism.

However, control engineering database requirements go beyond semantic data modelling capabilities. They require more than complex information modelling, semantic schema design, dynamic schema evolution and rigorous constraint management capabilities. Other facilities they require include the management of large volumes of data, data sharing (for collaborative engineering which also necessitates management of long transactions), data versioning, efficient storage mechanisms for fast data access and retrieval, and computationally complete database programming languages (most control engineering applications involve complex mathematical manipulation of data, which necessitates a computationally complete or sufficiently low level database definition and programming language; it would be difficult, for instance, to perform control engineering design using a conventional database programming language like SQL). Almost all of these extra features are included in object-oriented database management systems [111, 18]. Object-oriented databases are in fact gaining popularity in the comparatively new 'complex' domains such as CASE, VLSI, CAD/CAM and other branches of CAE. A survey of object-oriented database management systems specifically designed for engineering applications is given in [2]. The object-oriented databases surveyed include ORION/ITASCA, ObServer/ENCORE, GEMSTONE, ONTOS, ObjectStore and VERSANT. A table showing the comparative ratings of features is also given which can be useful for selecting a particular product for a specific domain.

So far, the approach of using an off-the-shelf commercially available object-oriented database management system for control engineering purposes, although promising, has not been thoroughly investigated. Rather, object-oriented concepts have mostly been used to define general data models for control engineering, primarily because they offer close correspondence between programming constructs and their counterparts in the modelled realm. Furthermore, object-oriented concepts of inheritance (which allows general class properties to be inherited by specialized classes) and encapsulation enhance the consistency of user code and make its misinterpretation less likely. An example from control engineering where inheritance could be useful is of a general class 'system' containing definitions for a general system for which sub-classes can be defined, for example, a 'linear system' which contains for example four state-space description matrices, a 'non-parametric system' which contains for example a measured frequency response, etc. The general class concept with its inheritance mechanism allows designers to easily set up the database in the first instance and, if desired, to extend it to comply with their particular needs.

A basic database scheme based on an object-oriented data model and considered sufficient for building a 'control engineering database' is proposed in [45]. It defines five general classes for general nonlinear dynamic systems, time invariant state-space systems, transfer functions, real functions (such as time functions) and complex functions (such as frequency response) as well as methods for operating on these classes. This basic database scheme supports inheritance and is used as a central repository for data exchange between different tools. It manages complex structured data in a consistent and reliable manner over the design period. Users are, however, allowed to modify or add to the basic scheme. Another such object-oriented model for CACE is described in [41]. It allows changes to be made to existing tools/methods and also allows for the addition of new systems. There have been several developments of Matlab-like tools that incorporate object-oriented functionality such as Impact, ControlH, Ψlab and Xmath (these tools are discussed in detail in chapter 2). The object-oriented functionality in these tools is geared more towards 'programming style' than database management. An early work on object-oriented programming environment for CACE, mentioned earlier, is [79]. This environment did not use any object-oriented database management schemes directly. Rather, a small area of the control system data model was used to evaluate the suitability of object-oriented systems for representing Matlab designs. The main drawback of the environment was that each Matlab command or function had to have its equivalent defined in the object-oriented system and this limited the extensibility of the environment. An object-oriented kernel for CACE proposed by Kolb *et al* in [49] for *workspace and project management* is discussed later in this section. Object-oriented approaches for integrated CACE environments are discussed in section 3.2.

We now concentrate on a different aspect of CACE database management concerned with *workspace organization and project management.* When looked at from this aspect, a database is a collection of experiment histories over the lifetime of a project. Control engineering projects (and other engineering design projects) usually last for several months. A *special* database system is required for storing, retracing and retrieving the large amounts of data in a consistent way which would help to compare new results with previous results and to restart or to branch to a specific design. We will refer to a database management system that can perform these tasks as a project management system. Essentially, a project management system keeps track of the various models and results corresponding to a specific project and incorporates some kind of *version management* capabilities.

Since a project management system keeps track of relatively low-level entities such as an experiment, a model description or a computed result, there is a need to define a hierarchy in order to organize these different levels. An example of a typical hierarchy is to view a *project* as being composed of several *sessions* where each session is composed of several *experiments*. In turn the experiments are associated with parameters, algorithms, model descriptions, computation results, etc.

Usually several experiment runs are made in any control design and analysis project. There are several reasons for iteration of experiment runs: models created early in a project have to be modified because additional information (such as experimental results) becomes available as a result of a computation or because of changes in the design of the plant itself, etc. The loss of integrity, in the sense of not knowing which analysis and design results were obtained with which instances of the model, is the most common fault encountered in documenting the control system design cycle, and as stated in [105], results in irreproducible results, unsupportable design decisions and unproductive reiteration. This is where, a version management system which keeps track of different versions and alternatives of models and control systems designs can be very useful. Thus a version management system is part of the total functionality of the project management system.

The problem of version management is generally addressed at two different levels. One is at low-level and concerns the efficient storage and retrieval mechanisms of different versions and alternatives of large design files. An example of research into these issues is [46] which proposes storage structures based on 'differential files' in order to overcome large disk requirements in an engineering database. The second level is at high-level and concerns the way users perceive the creation and the use of different versions and alternatives. This level deals with the functionality of a version management mechanism as it appears to the user. The examples from control engineering cited in this chapter take into account the high-level view of version management. A survey of major issues involved in version management is given in [47] along with a framework which can be useful for designing a version management mechanism. The framework includes questions such as: What should be versioned? How are versions used? How is update managed?, etc. A five dimensional model for design data management including versions (results of iteration), views (different representations of a complex entity), hierarchy (management of complex designs by breaking them into smaller parts), status (for verification and validation; useful for managing quality, consistency, functionality, reliability and safety) and variants (similar but not identical entities; they require management of both common and different features) is proposed in [38]. Usually two or three dimensions are used for most problems. This framework can serve as a tool for analyzing user requirements and alternative solutions for a version management mechanism.

A useful feature that a project management scheme for CACE can provide is a mechanism for 'remembering' that a particular computation such as evaluation of the frequency response of a subsystem has already been performed and that its results are available for further use. But for this, it is important to keep a record of all the conditions and parameter values, etc. before a result is generated. (We have explored this feature while designing computational chains using Matlab Containers as explained in the next chapter.)

The first effort to track models and results and to integrate database management functionality in a CACE environment appears to be CATPAC [17]. The structure of the package allowed users to document their experience in applying an algorithm in a library which consisted of a 'procedure library' containing a 'procedure catalogue' and a 'model library' containing a 'model catalogue'. The information stored in these catalogues formed a kind of engineering knowledge base. A structure for a standard model file was defined which included model information, procedure specific information, parameter list and model data. This standard model file acted like a documentation of an experiment (a procedure run) and made it possible to repeat experiments with certain parameter modifications.

A detailed design for project management support for CACE including a complete hierarchy of projects, models, components, and results, and for version control is presented in [105] and [104]. The design is based on the experiences gained from the GE-MEAD and ANDECS projects. It catalogues and categorizes database elements for control engineering, establishes the relations between them, and proposes a hierarchical database framework with the levels project, model, attribute and element as shown in figure 3.1.

The first level, *project*, provides a convenient separation of workspace. The second, *model*, accommodates (for example) non-linear ODE descriptions of aircraft dynamics or matrix/vector (state-space) models of linear systems. The third level, *attribute*, supports the decomposition of models into components and provides a repository for analysis, design and simulation results obtained using a given model. The lowest level, *element*, comprises the components of a model or the individual results collected under the above attributes. Such a hierarchy reflects the belief that control engineers tend to think of projects and models as being of paramount importance; all data elements produced during CACE activity are 'children' of these entities. Important features of this organization include: a version control mechanism for models that change over the life of an investigation or design life-cycle; references or pointers that relate a subsidiary model such as a result from linearization, model reduction, etc. to the model from which it was produced; and setup definitions that capture and record exactly the conditions under which various results were generated, e.g. definitions of input signals used in producing simulation results.

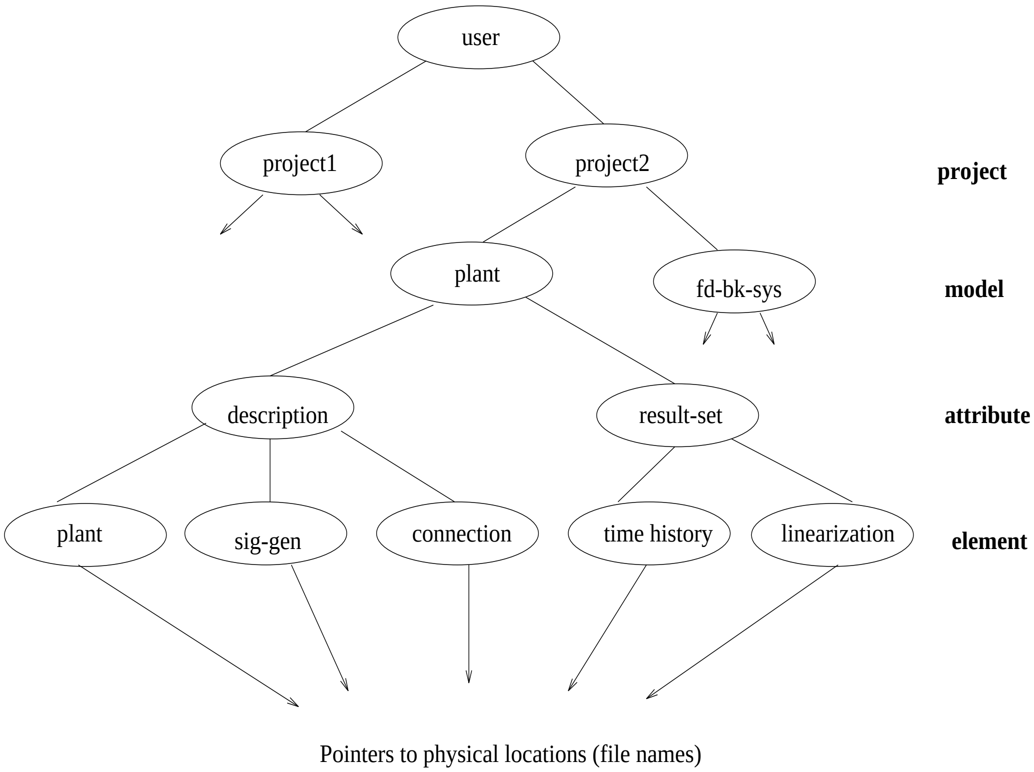

Figure 3.1: Database hierarchy for CACE [104].

In GE-MEAD [103], the data element categories included are project, model, component, description, result and condition specification. Models comprise of components and a description (type, connection, etc.). Associated with each model are results, for example, files containing frequency response data or time-history data. Models and results are organized within projects. Condition specification is directly related to each result. It contains information regarding operations performed on a model before a specific result is obtained. These include such actions as changing a parameter value, specifying an initial condition and/or input signal before performing a simulation, defining a frequency list used in obtaining Bode plot data, and so on. Condition specifications also record algorithmic conditions such as selecting an integration algorithm for simulation. Capturing this data is critical since it is the combination of model instance and condition specification that determines the result and thereby allows the engineer to document or repeat the result. Every element in the user's GE-MEAD database is fully documented as to how it was obtained. These database management capabilities solve to some degree the problems associated with version control of the user's models and of tracking related files such as those containing results. In fact, in the database management capabilities of GE-MEAD emphasis was placed on better project management facilities rather than on standard definition of data structures representing each type of data element. Data structures used or created by the core CACE packages were used and, when required, these were converted to the format used in a specific tool.

The computational design experimenting technology in ANDECS [37] supports the iterative and exploratory nature of the control systems analysis and design cycle (and hence supports what we call project management to some extent) by allowing three levels of experiments: model experiments, method experiments and parametric experiments. The experiments are structured as modular and configurable computational chains. During different experiment runs the system automatically maintains the evolving data chains. The computational chains support automatic storage and retrieval of design data in a design-history database. Unlike GE-MEAD—which did not use any standard data structures—ANDECS defines its own control-oriented data structures. According to [36], ANDECS automates the process of control design to such an extent that it makes it more like a computer-*integrated* control engineering environment than a computer-*aided* control engineering environment. The seven technology layers of ANDECS are discussed in detail in section 3.2 as a reference framework for other CACE environments.

ANDECS integrates project management into numerical processing (sequences of computational modules) tasks, which makes it more practical than an environment like GE-MEAD, which used an external database management system. In other words, in ANDECS, project management is inherent in the 'programming style', whereas in GE-MEAD, it was not so automated and the user was aware of the links to the external database management system. As stated in chapter 2, there were some major technical problems with the GE-MEAD project such as maintaining links to constantly changing external tools and the *bottleneck problem* that resulted from all information being exchanged through the *supervisor*. The software architecture of GE-MEAD is explained in the next section.

The idea of integrating project management aspects within the programming environment is also proposed in [49], whereby a kernel for a control systems engineering tool based on an object-oriented data model which aimed at providing better project management support is presented. This approach uses object-oriented concepts to propose a 'programming model' and in this sense it is different from other object-oriented approaches which mainly use the object-oriented concepts for abstract data typing. The 'programming model' proposed incorporates the user's work strategies and is based on the following conceptual categories: projects (sessions showing the whole history of the user's work related to one topic); systems, signals and numerical data (basic numerical objects); algorithms (operate on the basic numerical objects); and experiments (controlled execution of algorithms on basic numerical objects provided the values for parameters are supplied). The important features of the kernel are that it allows important steps in any control design session to be recorded, which helps users keep track of their work; all such records for a whole project (collection of several sessions) can be displayed in the form of a tree diagram; and the kernel also allows parts of work to be re-executed. Like ANDECS, this research project emphasizes the fact that users can concentrate more on the numerical task if data management is integrated within the 'computational tasks' and automated as far as possible.

Some tools take a somewhat different view and extensively use graphical user interfaces (GUI) for managing computations and organizing the workspace. These interfaces are mostly used as a front-end to database functionality. For example, browsers (a user interface feature) are used for managing file space (a database feature). An important feature of GUIs is that the effects of changes in parameters or methods can be readily studied. GUIs give tools a certain degree of 'intelligence' by displaying highlighted menus, buttons and plot handles which guide the users through different procedures. The ISID (Interactive System Identification) Module in Xmath [43, 77] makes use of these features of GUIs. The user can interactively identify a model from input-output data, analyze the influence of parameter changes on the resulting model quality, compare and validate different models and re-examine new results. The GUI in ISID has been implemented as a set of single-window interfaces (one each for 8 different functions), each with a standardized pull-down menu. The data associated with each GUI is stored in a separate Xmath partition. Whenever a GUI-based identification tool is instantiated, a specific 'hidden' partition with a name prefixed by an underscore is created and used to store all variables related to the identification. The data in this partition is not altered by non-GUI based operations, which means that even if one exits a GUI tool, one has the option to start again and to continue from exactly the point where it was left. Within any given GUI, results can be graphically compared with those of other models by exchanging models and/or data through variables in the Xmath main partition.

In [61] two models for GUI, especially in the context of CACE software, are classified as 'object-oriented' or 'action-oriented'. In an object-oriented GUI the basic notion is that one manipulates 'objects' or 'things'. The selection of an operation to which the object is to be submitted is secondary. An example of such a GUI is the one provided in the new version of the Matlab System Identification Toolbox [93] which focuses on management of objects, especially data sets and models. Operations on these are selected one at a time. In an 'action-oriented' GUI the basic notion is that one selects 'actions' or 'operations'. The focus is on selecting an operation; the selection of objects to be operated upon is secondary. The GUI Xmath System Identification Module described above is an example of an 'action-oriented' GUI. The GUI in Xmath System Identification module concentrates on construction of 'computational chains' to which specific objects (data sets) can be attached later. The 'object-oriented' style of GUI gives more support with data management in the sense that a single GUI is present throughout the user's interaction with the toolbox so that track can be kept of filespace and workspace. However, 'object-oriented' approach is useful only if there is a well-defined model of processing data. This exists for system identification, for example: data takes the form of measurements or of models, and there is a well-prescribed sequence of steps required to process measurements in order to obtain a model. At each step a user-defined choice of algorithm is available. In this case a GUI can help with the algorithm selection, but its most useful roles are to prompt the user, keeping track of which stage has been reached and what alternatives are available, and keeping track of the models created and their relationships to the measurements.

The two models for GUIs are in fact similar to the two approaches for information processing called 'data-centered' approach and 'tool-centered' approach. These two 'views' of information processing are discussed in the context of CACE in [7] in which it is pointed out that the tool-centered model is best suited for complex operations on simple data while the data-centered model is suitable for simple operations on complex data. This discussion is taken further in the direction of 'tool abstraction' and 'data abstraction' in section 4.5.

3.2 Software Architectures for Database Support in Integrated CACE

It is not possible for a single package to support the complete control design cycle without adding to the size and complexity (in terms of both usage and implementation) of the system. If the answer lies in developing extensible tools for the different stages of the control design cycle, then clearly a need arises for an integrated system for these tools. Since design is inherently iterative, it is important that such an integrated environment should support flexible use of several tools in an unknown order. The simplest way to achieve this is to use a multi-tasking operating system and run the various tools on the same machine, switching between them as necessary. This approach was adopted in the expert-aided sequential design of multivariable systems [15]. But there are obvious problems with this approach: translators are needed to resolve the inconsistency of data formats so that data can be transferred between different tools; and users have to switch between different tools with different interfaces which can be extremely confusing. Overcoming the inconsistencies in data formats and user interfaces of tools in an integrated environment are the main problems faced when integrating tools. The problem of inconsistency of user interfaces in the different tools is beyond the scope of our research. In this section, we will concentrate primarily on the software architecture aspects of the problem of data organization in integrated environments, the problems of inconsistent data entities in the different toolboxes and the standardization efforts in terms of both 'open systems' and data structures.

In several proposed architectures for integrated CACE, databases hold a central position so that they can be used as a convenient channel for passing data between different software modules. One such architecture is proposed in [7] which emphasizes that since models are central to the process of control systems design, the kernel of an integrated environment should support the representation of models and have basic functions for manipulating and accessing models. This kernel could then act as a central database of models accessible to all tools as shown in figure 3.2. It is suggested that Omola [5] could be used as a textual format for the model database. Omola is useful for model documentation and as a standardized exchange language between users and tools.

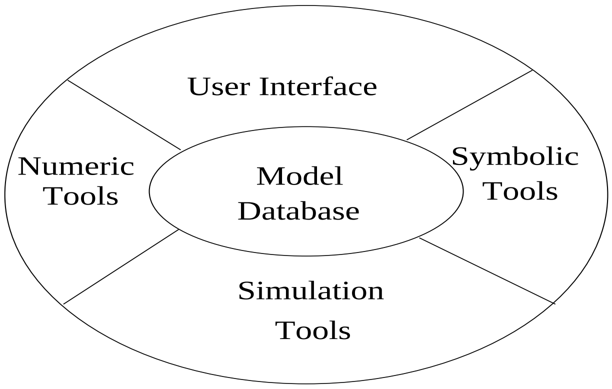

Figure 3.2: An integrated CACE environment [7].

Federated computer-aided control design system, ECSTASY, and GE-MEAD were also based on the idea of a central database. But the tools did not directly interact with the database. Rather, these environments had a 'shell' consisting of some sort of supervisor program through which the user and the database management system communicated with the 'core' packages. As an example, we consider the software architecture of GE-MEAD which is illustrated in figure 3.3. A module called the 'supervisor' combined and coordinated various CACE packages and support modules (user interface, database management system and an expert system). Multiple packages were run under the supervisor and data was reformatted or converted, when necessary, to ensure compatibility among packages. The supervisor accepted 'GE-MEAD commands' and translated them into 'package commands' so that the user did not have to learn all the intricacies involved in using each package unless advanced functionality was required which could be accessed via Package Mode (using a core package under GE-MEAD environment via its own interface). The Supervisor contained most of the 'intelligence'. It accessed data files that defined how to use the underlying packages (e.g. how to convert GE-MEAD commands into package commands), directed the activity of the database manager, and managed the command and data flow between itself and the user interface. It tracked the high-level activities of the user, including knowing what model(s) had been configured. The supervisor also controlled the invocation and the use of the expert system.

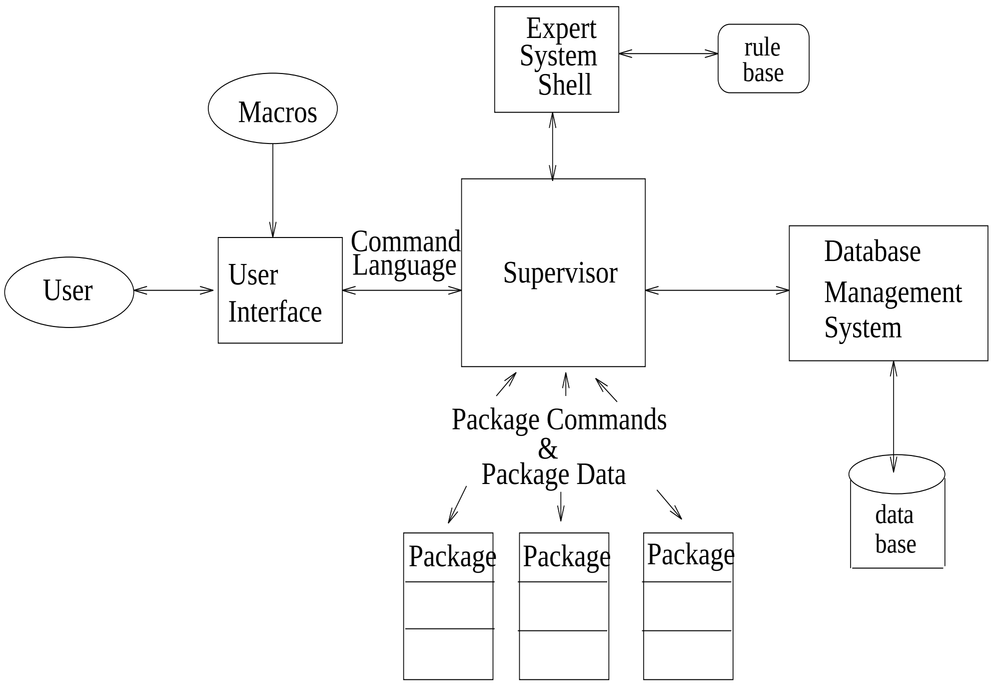

Figure 3.3: Integration of CACE packages by means of a shell.

As discussed in section 3.1, the drawback of shell or external integration architectures, such as GE-MEAD, is that they are very difficult to create and maintain. Moreover, in these architectures, the database is not recognized as the core of the system and the packages have to access the database via the supervisor, which results in a *bottleneck problem* and the users find that the supervisor just impedes their communication with the packages. The main problem, however, is maintaining the links to constantly changing external tools.

This would not be a problem in an integrated environment such as ANDECS which has well-defined *software levels* (the seven technology layers explained later in the section) for different 'services', and where groups working on different levels closely collaborate about any modifications and the effect these modifications would have on other 'levels'. The reason, of course, is that the whole environment is maintained at one site. The problem occurs when tools developed by different vendors are integrated. Several *Open Systems* architectures have been proposed for overcoming this problem. Within the CACE research community several, framework reference models have been proposed. Considerable work has also been done towards the standardization of control engineering data structures which is an essential requirement for moving towards open systems.

The open systems approach enables any tool from any vendor to work co-operatively with any other by sharing common data formats, control mechanisms and user interface. In CACE, the starting point for such an approach is to agree on a framework reference model (FRM) that would give a clear picture of what is needed in an integrated CACE environment. One such FRM that has been widely discussed recently for CACE is based on the European Computer Manufacturers Association's (ECMA) proposed framework reference model for computer-aided software engineering (CASE) environments. The FRM for CACE based on the FRM for CASE is shown in figure 3.4. The tools provide the functionality of the environment; they just slot-in to the framework which enables them to cooperate with other tools by means of the services provided by the framework. The framework provides all the common services that the tools need to communicate with the user, with each other and with the framework itself. These services include (a comparison of the services in the FRM and the seven technology layer in ANDECS, given later in the section, provides an example of these different services):

- `Common user interface services`: provide a consistent user interface for the environment.
- `Task management services`: insert a layer of abstraction between the user and the facilities provided by the environment which allows the user to concentrate on the task at hand without the distraction of low level command sequences.
- `Database services`: provide a store for all data objects in the environment together with facilities for their naming, maintenance and management.
- `Modelling services`: provide facilities for development of models of virtually any dynamic system.
- `Communication services`: provide managed communication between all the facilities in both the framework and the tools.

In [12] it is suggested that the user interface services, database services and message services in the CACE FRM are application independent and may thus form part of any general computer-aided engineering framework, whereas the task management services and data integration services tend to be application specific and so must be tuned for the application area. This simplifies the task of defining standards for CACE as it may be possible to simply adopt application independent standards that are developed in other better funded areas such as CASE, electronic CAD and mechanical CAD, and concentrate on specifying standards for the application specific components.

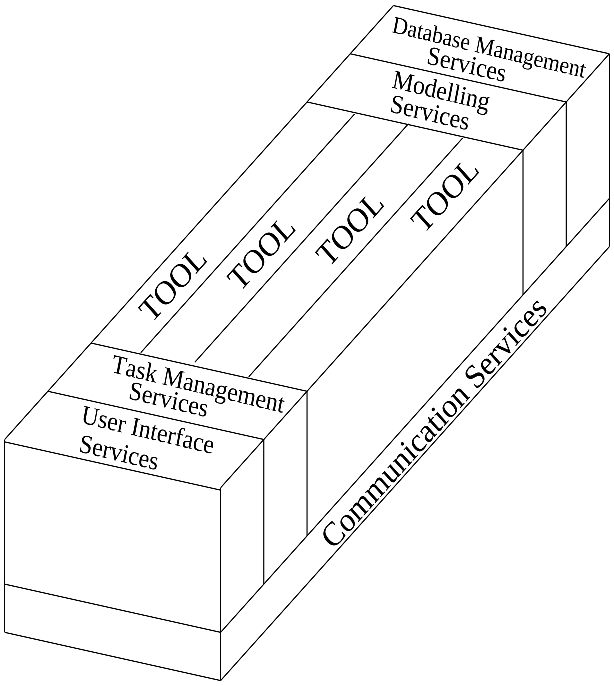

Figure 3.4: A framework reference model for CACE [12].

The question that arises here is: how far a comparison between computing environments for control engineering and those for software engineering justified? There are obvious similarities between the two. Both include the stages of problem formulation, analysis, design, implementation, testing and maintenance. They are both feed-back design processes in which there is a great deal of iteration involved. But when looked at closely there are several notable differences in the two. Software engineering is based on structured methodologies that are carried through from problem formulation to implementation, testing and maintenance. The goal of CASE technology is to separate an application program's design from its coded implementation and to automate the generation of software that is based on a design built with CASE tools [80]. This is however not true for CACE environments because once an off-line controller design is available, the actual implementation of the controller is not a major problem. There are several tools available for this, for example, AutoCode which is part of Matrix$_x$. The main problem in CACE is the off-line analysis and design which generally does not follow a structured approach and in which the designer plays an important role. Normally, no tools or methodologies are used for problem formulation or requirements specification in the control design cycle and most of the time is spent on the analysis and design phase. In software engineering there is no 'programming' involved in the analysis and design stage and most tools deal with 'specifications'. On the other hand control systems analysis and design is an 'experimental-study' and programs have to be written to conduct these experiments. Another important factor which differentiates CACE from CASE is models. Models are important in all steps of the control design process and there is a need for a separate modelling phase. Models are used by the control designer to get a basic understanding of the properties and behaviour of the plant. They are used as inputs to design methods and for validation of candidate controllers. Usually, it is the diversity of models used by the various packages and tools within CACE that provides the greatest hindrance to integration. Therefore modelling services have been specified separately in the CACE FRM.

In fact, it is difficult to make a clear *architectural mapping* between the reference model based on CASE and any of the present integrated CACE environments except for ANDECS for which a one-to-one comparison with the FRM can be made. The technology layers for control design experimenting in ANDECS, as shown in figure 3.5, match the services in the FRM. This mapping is shown below where the numbers in brackets correspond to the circled numbers in figure 3.5.

- Database services → (1),
- Modelling services → (2),
- Tools → (3,4),
- Task management services → (5),
- User interface services → (6).

A brief description of the seven technology layers shown in figure 3.5 is given here: (0) the heterogeneous hardware layer with various computer operating systems, (1) a homogeneous software operating layer called the engineering operating system RSYST, (2) a data abstraction layer based on object-oriented principles which also serves as data and model import and export layer (CDOs and DSBLOCK), (3) a numerics layer formed of a subroutine library (RASP), (4) a functional layer providing domain specific computational modules such as DSSIM for simulation, (5) an experimentation layer which realizes the multi-objective programming system (MOPS) and provides a macro-definition environment for computational sequences, (6) an interactive steering layer (VISTA) with 'plug-in' visualization and command software.

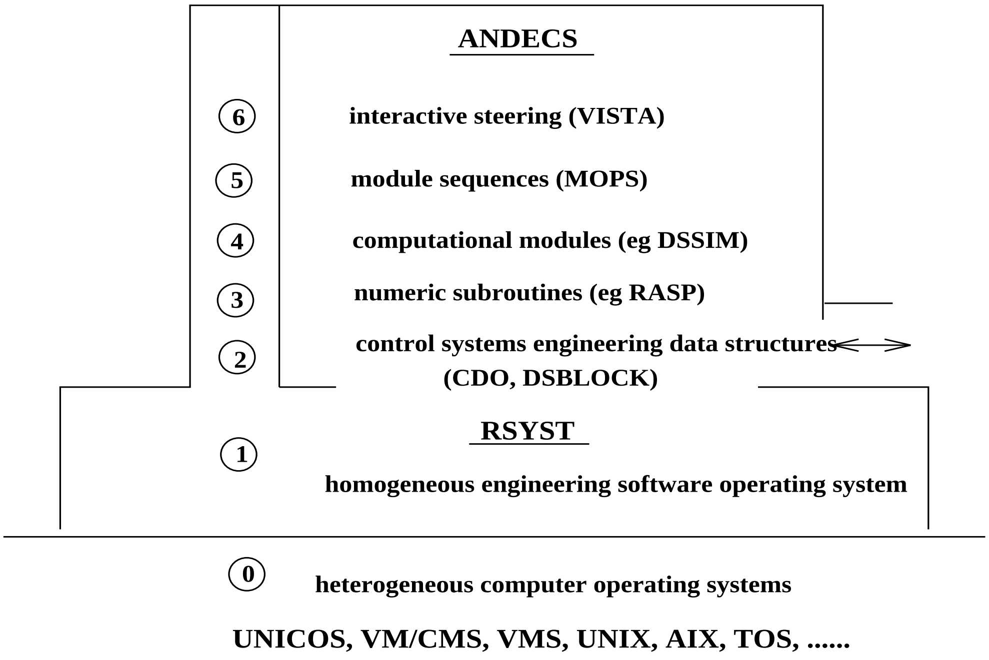

Figure 3.5: The seven technology layers required for integrated computational experimenting [34]

The purpose of a CACE FRM is to provide a common basis for integration of tools. From the user's perspective, the environment should appear as a seamless collection of tools. From the environment builder's perspective, the FRM provides a means of easily integrating tools. The environment builder's perspective most often leads to the common argument against a FRM that the choice of a particular framework model such as the reference model based on CASE might narrow the scope for good architectures. Most researchers working in the area recommend the FRM as a model for services that should be provided by an integrated environment and to aid discussion, rather than it being used as a definite model for the physical architecture of an integrated CACE environment.

Another framework called A_RSYST (ANDECS_enhanced RSYST) framework has been proposed in [35]. The basis for this new framework, as stated in [35], is that the CACE FRM based on the CASE FRM is not complete because it does not include 'algorithmic services' which are important to CACE. The rest of the services it supports are similar to the CASE-based FRM. There is no doubt that algorithmic services are an important aspect of CACE environments. However, the need for the extra service included in A_RSYST FRM is debatable since one might argue that this service is included in 'tools' in the CASE-based FRM.

As stated above, if the idea of open systems is to be taken further, it is important that a FRM model (if chosen) and requirements specification for an open system must be discussed and agreed upon by a wide cross-section of the users and implementors of these environments. Only by reaching consensus will the developers of tools and packages be persuaded to support the development of framework-conforming tools or to move towards true ‘openness’. The object-oriented operating systems and high-level frameworks for graphical interaction being developed may help in moving towards true ‘openness’. This is already available to some extent in the Macintosh and Microsoft windows environment where tools have much more potential for integration than in traditional workstation environments. [31] gives an example of how Microsoft’s OLE 2.0 and OLE Automation technology allow features of one application such as Microsoft Word to be used in another application such as Microsoft Excel.

There is a basic problem in CACE at the moment: the end-users, extenders and implementors (see section 2.3) have different perspectives on how ‘open’ the open framework should be [13]. The end-users need a user-friendly tool which makes it somewhat restrictive for extenders who require open systems which would give them more flexibility. The developers, in order to sell more products, limit the degree of openness so as to appeal to the larger group of end-users.

Object Management System Architecture (OMS) is an example of another architecture for achieving the desirable goal of ‘plugging in and out’ of various applications [40]. The PCTE (Portable Common Tool Environment) [30] project, concerned with specification, design and implementation of a host structure for software engineering environments, was one of the first frameworks to be based on this architecture. The OMS architecture is shown in figure 3.6. The database of the OMS, called the ‘object repository’, stores knowledge of the objects’ properties and inter-object relationships (schema). OMS provides a transparent interface called the Object Request Broker (ORB) to a database distributed across a network. ORB specifications for services and interfaces to the repository are provided by the pioneer in this area, the Object Management Group (OMG). Features supported by OMS in PCTE which are of considerable interest to control engineers include: organizing the changes and updates to information (transaction management); chronological and logical trails of user actions history during the software development process; versions and variants of objects; efficient storage and retrieval of relatively large grain information such as program modules and documents; access to the repository so that users working simultaneously on the same data can work productively without affecting one another (concurrency control). These services to be provided by the repository can be found in many commercial object-oriented database management systems.

An important area in which progress should be made in parallel to research in open systems for CACE is the standardization of data structures for CACE. Data exchange between tools, possibly on different machines and in different organizations is an important requirement in CACE as well as in other areas of computer-aided engineering. The requirement exists for passing data around within a single integrated environment

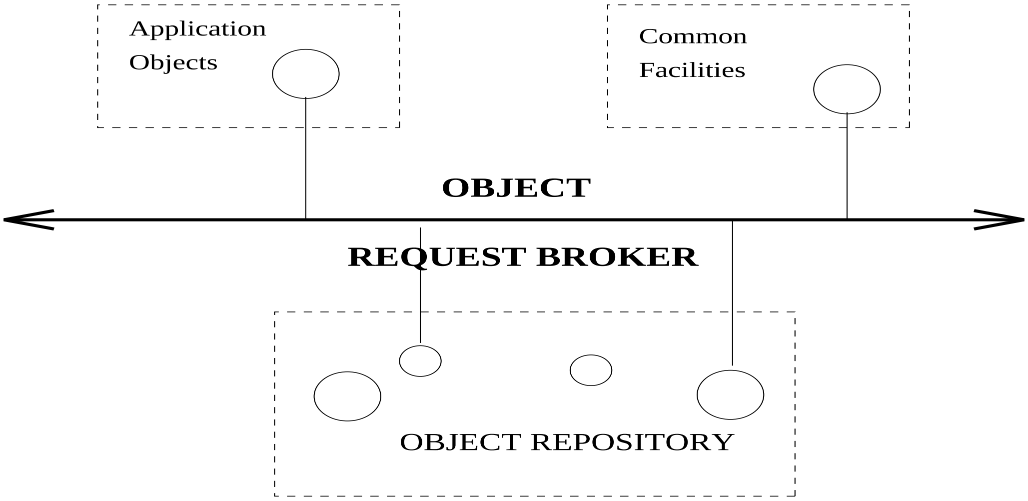

Figure 3.6: Object management system architecture [40].

as well as for exchanging data between project collaborators [65]. An essential prerequisite for this is an agreed definition of the data to be exchanged in terms of both the syntax and the semantics. Once agreement is reached, there are additional benefits to be gained. For example, the agreed definitions could serve as the basis for application specific object-oriented database systems and could be used for the implementation of numerical algorithms that operate on these standard data definitions.

The use of the EXPRESS data definition language [92] for the definition of control engineering data is explored in [59] and [39]. EXPRESS allows for the definition of data types, class structures, and constraints of various kinds. It is an object-oriented language in the sense that class hierarchies can be conveniently defined by means of an inheritance mechanism and by means of individual objects that have distinct identities. EXPRESS is part of the STEP/PDES [66] standard for engineering data exchange which has been adopted by the ISO. The use of a language adopted as a standard in other fields of engineering clearly offers a number of advantages. The German aerospace research organization DLR has produced an EXPRESS schema, part of which is presented in [39]. The classes relating to linear models are based on standards of control engineering data developed by a working group of the German engineering institution VDI. These standards are specified using the schema of the control data objects already defined in the ANDECS project which closely match the data definition in EXPRESS. For non-linear models, the class definitions which already exist in the modelling language Omola have been used. An example of an EXPRESS schema for a 'dynamic system' taken from [39] is as follows:

```
SCHEMA cace;

 ENTITY ds;      (* dynamic system *)
```

```
  SUPERTYPE OF (ds_ode XOR ds_dae XOR ds_lin);
  note  : STRING;  (* comment text *)
  sigt  : signal;  (* Time *)
  sigu  : signal;  (* Input signals *)
  sigy  : signal;  (* Output signals *)
 END_ENTITY;

 ENTITY signal;
  ns    : INTEGER;
  name  : OPTIONAL ARRAY[1:ns] OF STRING;
  unit  : OPTIONAL ARRAY[1:ns] OF STRING;
  scale : OPTIONAL ARRAY[1:ns] OF REAL;
  group : SET OF UNIQUE LIST[1:#] OF INTEGER;
          (* defines a subset of signal components,
           eg. control_inputs,disturbance_inputs *)

 WHERE
  ns > 0;
  HIBOUND(group[1]) <= ns;
 END_ENTITY;

END_SCHEMA;
```

There are several groups and committees working on the standardization of data in different branches of engineering. Some groups working on standardization in control systems analysis and design or related areas are specified in [58]. The International Federation of Automatic Control (IFAC) has set up a working group on guidelines for CACSD software. IEEE's Control Systems society (IEEE-CSS) also has a standing committee on standards. These two are now merged. The main idea behind their establishment was to form a standards body analogous to the committees that draw up IEEE, ANSI and other standards. The German engineering institute VDI has also set up a working group with an aim similar to that of IFAC/IEEE-CSS. The CDIF (CASE data interchange format) family of standards for information exchange between CASE tools has working groups in several specialized areas, of which CACSD is one [1].

NMF [2] is a standardization effort that aims at standardizing simulation modelling languages. An NMF model is essentially a structured way of stating equations, variables and component model boundaries. A translator parses NMF model descriptions and generates environment code for different target environments in which the model is then used.

[1] Further information available from the *WWW on Internet*, http://www.cdif.org/.

[2] NMF reference report and handbook are available by *ftp* from urd.ce.kth.se/pub/reports.

NMF was initially developed for building simulation problems. However, there is nothing in NMF syntax that restricts its application to this field. An example of a simple NMF model for a thermal conductor is given below. In NMF, variables, parameters and (interface) links are globally defined to enhance model compatibility.

```
CONTINUOUS_MODEL     tq_conductor
ABSTRACT    "Linear thermal conductor"
EQUATIONS
   /* heat balance */
   0 = - Q + a_u * (T1 - T2);
LINKS
   /* type       name           variables... */
      TQ      terminal_1        T1, POS_IN  Q ;
      TQ      terminal_2        T2, POS_OUT Q ;
VARIABLES
   /* type       name    role      description */
      Temp        T1      IN        "1st temp"
      Temp        T2      IN        "2nd temp"
      HeatFlux    Q       OUT       "flow from 1 to 2"
PARAMETERS
   /* type        name    role    description */
     Area          a      S_P    "cross section area"
     HeatCondA     u      S_P    "heat transfer coeff"
     HeatCond      a_u    C_P    "a * u"
PARAMETER_PROCESSING
     a_u := a * u;
END_MODEL
```

In the UK, the Institution of Chemical Engineers and the Institution of Mechanical Engineers have established a joint working party on electronic data interchange. In the USA, the American Institute of Chemical Engineers has set up the process data exchange institute (PDXI), and a group of petrochemical companies together with the DOE have set up the Petrotechnical Open Systems Corporation (POSC). While these bodies are concerned entirely with moving live information around their organizations, and in particular around process plant, it is suggested in [58] that their activities should be of interest even to off-line control engineers concerned with control design. It will be easier to validate models, to acquire operational data for modelling and identification, to put together 'hardware-in-the-loop' simulations, to download the controller design to the plant computer, and to tune controller parameters, if there is a closer match between the interfaces of the relatively abstract simulation models and systems used in control engineering to those used in actual plants.

3.3 Summary

In this chapter, we have discussed the different database issues specifically related to CACE and the various approaches that have been adopted in software tools and environments for solving some of these issues. The chapter starts with some very basic aspects of CACE databases such as support for compound data structures, user defined data structures, and semantic constraints and associations in CACE databases. The importance of database management services based on how a user perceives the creation and inter-relationships of data is emphasized. This is followed by a discussion on the recent trend towards object-oriented approaches. Object-oriented approaches are gaining popularity because they offer a close correspondence between programming constructs and the way a user 'thinks'. The growing need for a project management system with version management capabilities for keeping track of the different models, analysis results, control systems designs and validation study results that evolve over the course of a project is also discussed. Finally, the role of databases in tool integration and the different efforts towards standardization of data structures are described.

The question that still remains unanswered is whether integrated environments and open-systems architecture are the right way forward, and whether research in these areas would eventually help overcome the deficient areas, or would it merely help in identifying the different services an ideal CACE environment should provide. One thing is clear though: the demand for 'open systems' comes more from the extenders than the end-users of CACE environments.

The different approaches for database management discussed in the chapter have their own justifications. But the most 'practical' seems to be the one proposed by Kolb et al in [49], where they emphasize that data management should be integrated into CACE numerical functionality and automated as far as possible so that users can concentrate more on the numerical processing involved in control systems analysis and design. Another example in this regard is that of ANDECS which, with its modular computation sequences that automatically maintain the evolving data chains in a central database, integrates data management into computational tasks. The point we would like to stress here is that database management facilities should be designed as an integral part of a CACE environment and not as a separate tool in which the integrity of the database is usually the responsibility of the user. What is required is a facility that would structure data to match the semantics of the application without losing the flexibility given to the control designer in popular numeric environments such as Matlab.

Chapter 4

Matlab Containers for CACE

In this chapter, the importance of data management and programming (organization of computational processes) in the control engineering design environment is acknowledged, but a novel approach is taken to the realization of facilities to support these. Whereas most of the work cited in chapter 3 has striven to be independent of specific products and has had the integration of software from various sources as one of the prime objectives, here the opposite point of view is taken in an attempt to realize some of the desirable functionality within the confines of a particular product. We have chosen Matlab as a 'test-bed' to explore our ideas but similar implementation could have been developed for other Matlab-like products.

As discussed in previous chapters, a major problem confronting users of Matlab and other similar products is the proliferation of data objects and custom 'scripts' which accumulate rapidly in the workspace and/or filespace during any non-trivial project. The example on the problem of data management by Taylor *et al* [105] in section 3.1 shows how easily thousands of such objects can be generated in even modest projects. *Matlab Containers* [63, 100] provide a means of solving some of these problems. The features of class-composition hierarchies [89] and computational chains [90] that have been built on top of the basic Container concept enhance the overall functionality of Containers.

The development of Matlab Containers was initiated by Maciejowski and Szymkat [63]. The explanation in section 4.2.1 is a result of their research, except for the concept of 'static' and 'dynamic' properties which were introduced later as a result of a need that arose during the course of this research. Some descriptions in section 4.3 are also taken from the earlier research by Maciejowski and Szymkat reported in [63]. The remaining work reported in this chapter and in chapter 5 was conducted later as a part of research for this thesis.

Containers have some, but not all of the properties of *objects* which appear in 'object-oriented programming' and 'object-oriented databases' [10, 3, 69]. The term 'Container' has been chosen in order to avoid the appearance of claiming real object-like behaviour because certain crucial properties of objects, such as inheritance of class definitions, are absent from 'Containers'. The features in Containers common to objects are: 'Classes' of Containers can be defined, 'properties' of Containers can be updated and queried, Containers can be interrogated about themselves (e.g. 'What are you?'), 'data' can be associated with 'methods', and so on. We think of this work as providing a step towards 'object-oriented interaction'.

We would like to emphasize that we are describing a concept rather than a 'product'. In order to explain Containers we frequently present fragments of code and discuss some implementation details. Everything described has been implemented, but as an unstable prototype. The idea is to explore the *kind* of new functionality that can be of most value in a Matlab-like environment rather than to provide a particular solution.

The first section is on 'Motivation'. Parts of this section have already been discussed as 'requirements', 'database aspects' or 'programming style aspects' in earlier chapters, but they are stated here again so as to give a clear background picture of the need for something like Matlab Containers.

4.1 Motivation

The original motivation for the development of Containers arose during the development of a Matlab toolbox for working with systems containing time delays [99]. It became apparent that a way of aggregating data was needed in order to avoid the users having to specify very long argument lists every time a function was called. At the same time a more flexible and understandable means of doing this was desirable than the 'packing' commonly used by some existing toolboxes. Packed representation is used by several toolboxes to represent heterogeneous information. For example, the SYSTEM matrix in the μ-Analysis and Synthesis Toolbox, the theta matrix in the System Identification Toolbox, MVFR matrix in the Multivariable Frequency Domain Toolbox, etc. The main drawbacks of 'packing' heterogeneous data into a matrix are lower clarity of code and an overhead (such as index coding) due to packing and unpacking of matrices. So the main motivation for Matlab Containers was to suggest an alternative to a 'packed representation'. As we hope to demonstrate, however, Containers turned out to be a more powerful tool than this original idea might suggest.

Ideally, software behaviour should support the user's way of thinking. As far as descriptions are concerned, computer models should follow conceptual models closely. For example, if certain parameters have only a local meaning their representation should have local scope. This is easily implemented through encapsulation in object-oriented programming. Similarly, if the user thinks of certain properties by symbolic names, their software representations should use similar names. This is the convention which most often contributes to self-explanatory and self-documenting features of application software. The Container construct presented supports both of these notions: local scoping and symbolic names.

There are also objectives related to programming style, or 'interaction style' in the case of interactive software. To some extent, this depends on the capabilities of the language which is provided, and to some extent, on the preferred type of user behaviour. In general the approach adopted should enable systematic creation and manipulation of model representations by providing standard (repeatable and consistent) programming techniques. Consistency of model descriptions, standardization of retrievals, updates and transformations, and the construction of variant versions should all be facilitated by the software provided to the user.

In order to motivate the concept of Containers, we present the classification of data objects shown in Table 4.1. The complexity of data objects increases from the top of this table (simplest objects) towards the bottom (most complex objects). The second column describes the characteristics of each type of object in the hierarchy. The third column gives some examples of each type. The final column shows the additional information and/or services which are required in order to exploit the complexity of each object.

Thus the simplest objects, which we have called 'raw data', require only pairs of 'stop' and 'start' markers to delimit them. As the object complexity grows, more information (such as array sizes or pattern templates) is required in order to interpret the objects correctly. In addition to 'passive' information, 'active' services, such as the capability to follow links or an ability to check on environment resources (such as availability of memory, existence of certain files, etc) become necessary at some point. With increasing object complexity, these services require more processing. This processing may be delegated to some interpretative mechanism or to the object itself. In the latter case, the 'intelligence' of the object is perceived to increase.

To make this classification more concrete, it may be helpful to relate it to Matlab. Version 4 of Matlab, with nothing other than arrays as data types, deals with data at the first two levels of the hierarchy. The forthcoming version 5 of Matlab, which will support record structures similar to those in *C* and *Pascal*, and so-called *cell arrays*, will handle data at the first four levels. The Containers introduced in this chapter allow data objects at the first five levels when used with version 4 of Matlab. Potentially, they will allow objects up to level 6 when used with version 5—their detailed implementation would need to be modified but the conceptual design could remain the same. We can illustrate the last point with an example of porting a complex model description from one toolbox to

Data object	**Characteristics**	**Examples**	**Additional requirements**
1) Raw data	Unstructured information	Bitstreams	Start/stop markers
2) Formatted elementary data	Array of cells (equal size cells)	Text strings Bitmaps Numerical arrays	Cell size Array size
3) Structured data	Patterns of cells (mixed cell types)	C-structs Database records	Pattern template
4) Inter-related data	Patterns of cells and links (pointers to objects)	C-structs with pointers Records in relational database	Pattern template Link following mechanism
5) Inter-related data with internal manipulation capabilities	Class objects with methods operating on themselves only	C++ objects (limited)	Pattern template Link following mechanism Associated code execution (interpretation)
6) Inter-related data with external manipulation capabilities	Objects with methods using environment resources	C++ objects in working application	Pattern template Link following mechanism Associated code execution (interpretation) Environment resource awareness

Table 4.1: A classification of data objects

another. Currently the user has to convert the various parts of the description into the 'native' format of a given toolbox. Of course this is not a problem if purely numerical or graphical objects are considered because they may be just reinterpreted. But if the user is limited to this kind of action, it is necessary to recreate the inter-relations 'manually', whereas the use of Containers allows the automatic preservation of these inter-relations. The key to this capability is the fact that a Container is able to interpret its information content itself.

The specific manipulation capabilities required from Containers fall into two groups. Firstly, there are requirements concerning basic functionality, such as the ability to extract selected properties, to initialize or define selected properties, to retrieve structural information (property names, cardinalities, etc) and the ability to store and retrieve internally accessible information (that is, 'private' or 'local' information, as opposed to the 'public' information resident in properties). Secondly, there are various application specific requirements.

As will be seen later in the chapter, some of the Containers defined serve to provide a greater variety of data structures than are currently available in Matlab. For example, Containers can be used to represent rational transfer functions (as single entities rather than numerator-denominator pairs) or systems with delays. Some alternatives to Matlab, such as Xmath or Ψlab, have built-in capabilities to handle some of these entities. Consequently, the need for Containers with these products is a little reduced. But we will also demonstrate that much of the most useful exploitation of Containers is for representing relatively high-level entities. None of the existing products provide such representations, though some of them have constructs such as lists, which would make their provision relatively straightforward.

Matlab Containers provide something akin to object-oriented database management facilities to engineering users of Matlab. Basically, this enhancement provides a database of objects which is *persistent* [91], namely one which survives after a particular interactive session has ended (this is in contrast to the transient variable form which only appears in the workspace—*dynamic memory*) and interactions with it are in the style of database transactions: Containers can be browsed, queried, or updated. However, since this is done by encoding objects in executable functions ('`.m`' files in Matlab), the distinction between (passive) data objects and (active) functions is blurred, and the same mechanism serves for the storage of data, functions and executable sequences of functions together with associated data objects (which we call *computational chains*).

4.2 Matlab Containers

4.2.1 Concepts

A *Container* is a new object which to some extent generalizes *variables* and *functions* as currently available in Matlab. That is, it has some characteristics similar to those of variables and others usually attributed to functions. The aim of this section is to emphasize the important conceptual characteristics of Containers rather than details of any one implementation. Nevertheless, the use and behaviour of Containers will be clearest with examples drawn from the particular prototype which has been implemented. This prototype is implemented entirely in Matlab. A few Matlab functions have been written which perform all the required creation and interpretation of Containers, which themselves consist of Matlab '`.m` files' with a particular internal structure.

In the presentation of examples, a convention has been adopted that statements which could be issued by the user are preceded by the prompt: `>>`. Output to the screen is not accompanied by any prompt.

There are two basic parts of a Container: a 'local context' and a set of 'properties'. The local context is a piece of code similar to the body of a standard Matlab function. The variable names that appear in the local context constitute a local workspace which is distinct from the global workspace of a Matlab session. Names of Container properties may appear in this local workspace; they do so as names of string variables whose values may be initialized in the property definitions.

A Container is defined in three phases: the definition of the class to which the Container belongs, a declaration of the existence of the Container (instantiation), and the definition of the Container local context and its properties.

A function called `class` serves as a constructor of Container classes. It has the following syntax:

```
>> class(CLASSNAME,[PROPNAME1;PROPNAME2;...],
   [PROPARGS1,PROPARGS2,...])
```

The first argument is a string representing the name for a class of Containers. The second is a matrix of strings representing the names of Container properties, and the last a vector of cardinalities of argument sets associated with specific properties. For example:

```
>> class('ltiss1',['A';'B';'C';'D'],[1,1,1,1])
```

represents linear time-invariant state-space systems; in this case each of the usual matrices is a property.

An alternative representation of LTI systems, which packs the four matrices into a single property called `ABCD` (but as four distinct matrices), is:

```
>> class('ltiss2','sz|ABCD',[1,4])
```

In this variant, the first property `sz` has cardinality 1, and is assumed to be a vector holding the input, state, and output dimensions. The string `'sz|ABCD'` has been used as an abbreviation for `['sz ';'ABCD']`. Also, in this case the individual matrices do not have individual names; the B matrix, for example, has to be referred to as `ABCD(2)`.

Semantically, the definition of a Container class creates a common template for Containers with a fixed but uninstantiated property structure. Individual Containers belonging to a particular class are created by a statement of the form:

```
>> CLASSNAME(CONTNAME,COMMENTLIST)
```

This results in the generation of a Container (of the chosen class) with the given name (first argument) and optional comments (second argument). For example:

```
>> ltiss2('plant');
>> ltiss2('controller');
```

creates two LTI system Containers, called `plant` and `controller` respectively, both belonging to the previously defined class `ltiss2`. At this stage, these Containers exist but are 'empty' in the sense that their properties are still undefined. The actual contents of a given Container may be examined by just entering the name of the Container, or more selectively by a call of the form:

```
>> CONTNAME([PROPNAMEX,PROPNAMEY,...])
```

with property names in an arbitrary sequence, which displays the selected properties only. This usage may be thought of as a 'get' message passing mechanism. It supports user keyboard interaction with the Container, since the result is only displayed and nothing is changed in the Container itself. The important feature of the above construct is that the properties are presented in symbolic (non-evaluated) form. Continuing our earlier example, typing

```
>> plant
```

results in the screen output:

```
  plant = [ a Container of LTISS2 class ]
   LOCAL:
   sz :
    (1)   sz_
   ABCD:
    (1)   ABCD_1
    (2)   ABCD_2
    (3)   ABCD_3
    (4)   ABCD_4
```

Note that the local context is empty. The property definitions can be Matlab strings which evaluate to appropriate values on execution, and in this case, they have been set to names of variables (which do not exist) generated from the property names.

The next step is to fill the contents of the Container by defining the local context and the properties. The local context is defined as follows:

```
>> CONTNAME('LOCAL',LINE1,LINE2,...)
```

where the arguments (apart from the first one) specify the lines of text—as strings, containing instructions to be executed locally. For example:

```
>> plant('LOCAL','num=[ 0 0 1 ];','den=[1 2 1];',...
   '[A,B,C,D]=tf2ss(num,den);')
```

(The local context may also incorporate comments as a means of documentation.) The syntax for defining properties is:

```
>> CONTNAME([PROPNAMEx,PROPNAMEy,...],EXPRLISTx,EXPRLISTy,...)
```

where appropriate arguments, in the form of strings, specify the expressions used to define the corresponding property arguments. For example:

```
>> plant('sz','[ size(B,2), size(A,1), size(C,1)]');
>> plant('ABCD','A','B','C','D');.
```

(We assume here the availability of the function `tf2ss` from Matlab's Control Systems Toolbox). If the contents of the Container `plant` are inspected at this stage:

```
>> plant
```

the result is:

```
plant = [ a Container of LTISS2 class ]
LOCAL:
 (1)   num=[ 0 0 1 ];
 (2)   den=[1 2 1];
 (3)   [A,B,C,D]=tf2ss(num,den);
sz :
 (1)   [ size(B,2), size(A,1), size(C,1)]
ABCD:
 (1)   A
 (2)   B
 (3)   C
 (4)   D
```

Here the local context serves to define variables which can be accessed in the property definitions. The local context can be viewed as 'private' information, while the remaining properties are the 'public' interface to the Container. This example shows a trivial case, but both the local context and the property definitions can involve algorithms of arbitrary complexity. Note that at this stage the Container contains only 'recipes'. Nothing has been evaluated yet.

Both parts of a Container may be treated either as textual definitions (symbolic representations) or interpreted as elements of code (instructions, expressions, etc). This departs from most existing CACE solutions, which are numerically (or graphically) oriented. To some extent this approach generalizes and clarifies the usual approach taken by users, who typically build model descriptions using collections of custom scripts, functions, or stored variables.

Properties serve as slots for information exchange with the working context of a given Container. Usually they are just expressions, optionally involving the local context. In this sense, their actual value is internally dependent. Properties are externally accessible in two ways: they can be evaluated and they can be redefined independently of the local context. Containers also support Composite properties which are associated with several arguments.

We distinguish between dynamic and static properties. Expressions corresponding to dynamic properties are evaluated after execution of the local context. The appearance of output arguments and explicit specification of dynamic property names forces evaluation of the requested properties. The resulting values are passed to the output arguments in statements of the form:

```
>> [VARLISTx,VARLISTy,...]=CONTNAME([PROPNAMEx,PROPNAMEy,...])
```

This can be thought of as the 'get property value' mechanism. It is the basic link between other programming constructs and Containers. For example:

```
>> AA=plant('ABCD(1)');
```

causes the `ABCD(1)` property to be evaluated and the result assigned to variable `AA`. On the other hand, issuing the statement

```
>> plant('ABCD(1)')
```

with no output arguments results in only the definition being displayed with no evaluation.

Static properties are distinguished at the class definition stage by preceding their names with '`-`' (the minus sign). They are treated as variables that are initialized by their defining strings *before* the execution of the local context. They may be directly internally accessed and redefined only in this way which explains the use of the term 'static'. To illustrate their use, we give a third class definition for representing LTI systems:

```
>> class('ltiss3','-filename|ABCD',[1 4])
```

The idea here is that the first property will be the name of a file which will hold data which can be used to define the `ABCD` property of Containers of this class. Thus the statements:

```
>> ltiss3('newplant');
>> newplant('LOCAL','load(filename_);');
>> newplant('filename','plant_def');
```

define a Container `newplant`, of class `ltiss3`, whose static `filename` property is defined to be the string `plant_def`, and whose local context in effect contains instructions to load the file `plant_def.mat` (assuming the usual Matlab file naming conventions). The point here is that we need a means of ensuring that properties such as `filename`, which are used by the local context, do not themselves depend on evaluation of the local context. The distinction between dynamic and static properties enables this to be ensured.

Note that in a query such as:

```
>> AA=newplant('ABCD(1)')
```

any internal names which might have been used for particular matrices stored in the file `plant_def.mat` do not interfere with any identical names which might exist in the working context from which the query is issued.

The first argument of a call to a Container can take certain values which have special significance and which are used to interrogate the Container about itself. For example, if the first argument is `NaN` ('Not a Number'), then the Container's parent class name is returned. This can be used to implement a degree of 'type-checking', namely allowing only appropriate queries and updates to be applied to a Container when programming with Containers. Another reserved word of special significance that a Container can take as its argument is `ALL`. This is used to get definitions or values of all dynamic properties. Similarly `-ALL` is used for static properties.

Most of the constructs for Container manipulation were introduced with user interaction at the keyboard in mind and resemble transaction style (in the database sense) for browsing, querying and updating. The commands for creating a class, instantiating a Container for a class, setting the definitions for the local context and the properties resemble a Data Definition Language (DDL; a database command language which defines or describes database objects). The commands for obtaining property definitions and values and resetting the local context and property definitions resemble a Data Manipulation Language (DML; a database command language which supports manipulation or processing of database objects). All Container constructs resemble other programming constructs in Matlab. This is essentially similar to the concept of 'cognateness' in natural languages.

4.2.2 Implementation

In this section we explain the underlying implementation of Matlab Containers. We shall discuss the implementation aspects of the different stages of the life cycle of a Container which include definition of Container class, instantiation of a Container to a class, definition of local context and properties. Implementation issues related to modification of Container properties and local context and to issuing queries on Containers are also discussed. All of the underlying implementation uses Matlab version 4.

Container classes are created by a function called `class`. It takes three input arguments for the class name, property names and property cardinalities, and produces a function file (say `classname.m`) as a result. The new function file (which corresponds to a new class) takes the class name specified as a string in the first input argument in the call to the function `class` (here we assume this name was 'classname'). The function `class` uses Matlab version 4 file manipulation commands to create the new function file. As such, the creation of a class is actually the creation of a Matlab function file which corresponds to that class.

The new function `classname` acts like a 'Container constructor'. It takes a Container name as its first input argument and a comment as a second optional input argument, and produces another function file, let us call it `contname.m` (assuming this was the Container name given as the first input argument). Two other script files are also created as a result of the execution of function `classname`. One contains information about the properties associated with the Container and the other holds information about the local context for that Container. The function file `contname.m` has links to these script files. As a convention, these script files have the same name as the Container but are appended by either '_p' (for properties) or '_l' (for local context). For the Container `contname` (which is actually a function!), the two script files holding information about the property definition and local context would be `contname_p.m` and `contname_l.m` respectively.

Therefore, when a Container is instantiated to a class, in the sense of underlying implementation, a function file for that Container (here `contname.m`) and two script files (here `contname_l.m` and `contname_p.m`) are created. Once the Container is created, all manipulations on that Container take place through function calls on the function file associated with that Container. In our case, all manipulations on Container `contname` would be through function calls to `contname.m`.

But to the user, the Container `contname` does not seem like a function and appears to have more 'intelligence' than an ordinary Matlab function. It provides the user with a very flexible query style and does not have the same restrictions that a normal Matlab function has, i.e. restrictions about the order and type of input and output arguments. Users can interact with Container `contname` in a declarative style. To provide a Container with these functionalities, a Matlab script file called `gen_cont.m` (for *generic Container*) has been implemented. `gen_cont` is the 'driving engine' behind all manipulations and queries that are issued on Containers. The number of input and output arguments and use of some special 'built-in' names such as LOCAL, ALL, NAN and property names in the call to a function representing a Container help `gen_cont` determine the kind of operation that is required. Each Container function file has links to `gen_cont.m`

Another important function worth mentioning is `loc_edit.m` which is responsible for editing the local context and is called from within `gen_cont` whenever the local context needs editing.

In implementation of Matlab Containers, extensive use has been made of the (new) features of Matlab version 4. These include file manipulation functions that have been used for creating or modifying other Matlab function and script files, the `disp` function for displaying selected parts of a Container, and the `eval` function for evaluating instructions and expressions.

4.3 Tool Integration Using Matlab Containers

A problem faced by a control engineer using Matlab is that various toolboxes use different data structures for the same entity.

Examples of inconsistent entity representation in different toolboxes:

State-Space Models In the Control Systems Toolbox (CST), the Multivariable Frequency Domain Toolbox (MFD) and the System Identification Toolbox (SID), the state-space models are represented by four separate matrices usually named A, B, C and D. But in the μ-Analysis and Synthesis Toolbox (MUT) these are stored in a single Matlab data structure, referred to as the SYSTEM matrix which has the following format:

$$\begin{pmatrix} A & B & nx \\ C & D & 0 \\ 0 & 0 & -INF \end{pmatrix}$$

where nx is the number of states. In the Robust Control Toolbox (RCT), the four matrices are packed into a single vector. The System Identification (SID) Toolbox supports a structure called the theta matrix which contains information about model structure, estimated parameters and their estimated accuracy. There are functions in the SID Toolbox that then convert the theta format matrix into state-space or transfer function form. Some functions directly operate on the theta format matrix for cases where model structure is required.

Transfer Functions In the CST and SID Toolboxes, the transfer function is represented by two matrices NUM (numerator coefficients with as many rows as there are outputs) and DEN (common denominator coefficients in the descending powers of s). In the MFD Toolbox, the single input multiple output (SIMO) representation of a transfer function system has been extended to cover multiple input multiple output (MIMO) systems. The extra columns of numerator polynomial matrix which describe the other inputs are appended to the columns of the first input. The denominator polynomial is common to all the numerators.

Frequency Responses In the MFD Toolbox, in order to store the frequency response of a MIMO system at k frequency points, the individual frequency response matrices are appended by row to form a single matrix with $k * m$ rows and n columns, where m is the number of outputs and n is the number of inputs. For example, if $G_1, G_2, G_3, \ldots, G_k$, are the component frequency response matrices at the frequencies ω_1, ω_2, ω_3, …, ω_k, then the multivariable frequency response matrix F is stored as:

$F = [G_1; G_2; G_3; \ldots; G_k]$

and the associated frequency vector W is stored as:

$W = [\omega_1, \omega_2, \omega_3, \ldots, \omega_k]$

The vector W indicates the frequencies at which the system responses are evaluated. Given the frequency vector W and the matrices A, B, C, and D, the function `mv2fr` generates the required frequency responses. In MUT, the function `frsp` calculates frequency responses of SYSTEM matrices. Given an input vector W of N real frequencies and a SYSTEM matrix sys, the command $sys\_g = frsp(sys, W)$ calculates frequency responses for each independent variable ω_i and stores it in sys_g. The MUT stores MIMO frequency responses in packed matrices which include the frequency vector whereas the MFD Toolbox represents them by packed matrices with the frequencies in a separate vector. The data structure (packed matrix) for storing the frequency responses and the frequency vector in MUT is called a VARYING matrix. A VARYING matrix is also useful for storing time responses.

A way of resolving this problem using Matlab Containers was given in [63] and is described here. An obvious way is to define Containers for widely used classes of objects such as the class `ltiss` defined above. For each class a family of functions could be defined, each one translating the Container contents to the form required by a specific toolbox (`ltiss2mu`, `ltiss2mfd`, etc.).

An alternative and less obvious way of using Containers is to put all the definitions of each entity into the local context initialisation and to use the properties as 'read-out functions' to obtain the form required for each toolbox. For example, MIMO frequency responses could be represented by the class `mimofr`:

```
>> class('mimofr',[MUT;MFD],[1,2]);
```

The local context initialisation of *each* Container of this class would derive, say, the input and output dimensions `nin` and `nout`, the frequency vector `w` and its length `N`, and the frequency response `f` itself in MVFR form (the form used by the MFD Toolbox). The properties could then be defined thus:

```
>> mimofr('fr1');
>> fr1(MUT,'[[f;[zeros(1,nin-1),N]],...
         [w(:);zeros(nout*N-N,1);Inf]]')
>> fr1(MFD,'w','f')
```

and variables for use with the MUT and MFD Toolbox could then be obtained by the calls

```
>> frmut = fr1(MUT);
>> [w,frmfd] = fr1(MFD);
```

The query

```
>>fr1(MFD)
```

displays the property definition of the property MFD, i.e. the format for the frequency vector and the frequency response matrix in the MFD Toolbox,

```
MFD:
    1   w
    2   f
```

and the query

```
>>fr1(MUT)
```

displays the format of the frequency response matrix in MUT as follows:

```
>>MUT:
      1 [[f;[zeros(1,nin-1),N]],[w(:);zeros(nout*N-N,1);Inf]]
```

Although not strictly necessary, convenience would demand that new Containers of class `mimofr` were created by a special function (`newmimofr`) to save keying in the same definitions each time. A better way would be to have an option for defining default property definitions at the class definition level so that each time a new container is created it automatically takes the property definitions given at the class-definition level.

Clearly, one would not wish to store a large numerical array in the Container. A practical solution is to call an appropriate function (recomputation on demand) for computing data values or to read the data from a file (precomputed only once). This gives the resulting database the separation between 'conceptual' and 'logical' organization of data, as recommended by the Codasyl standard [25]. The property definitions determine the 'conceptual' (external, public) model while the local context initialization determines the 'logical' (internal, private) model of data. The data values if read from a file would depend on the format of the file such as ASCII or binary MAT-file. Therefore, different Containers (even of the same class) would normally have different local contexts depending on the way the data values are stored and read from files and assigned to variables.

Although this approach still needs to be explored in detail, it seems to be a promising approach to implementing standard classes of control objects described in [13] and [58].

4.4 Class-Composition Hierarchy Using Matlab Containers

In this section we go beyond the functionality of Containers discussed in section 4.2.1 by showing that Containers can be used to support composite classes—namely those whose properties are themselves Containers. Such a composition of objects is desirable in many applications. In this case, properties are defined using properties of lower-level objects. In the context of the 'get property value' mechanism, evaluation at a higher level of the hierarchy then induces evaluation at a lower one.

In object-oriented terminology, a class-composition hierarchy defines a *complex object* which has an *is-part-of* relationship with other objects i.e., an object which has a complex structure consisting of other sub-objects [48]. In control engineering a complete control system can be viewed as a complex object which contains the objects plant and controller (this is the simplest case; an example of a class-composition hierarchy for a control system with several components is given in chapter 5). For linear systems the description of each of these is in turn composed of the state-space matrices A, B, C and D or transfer function matrices NUM and DEN.

Let us say two Containers `sys1_plant` and `sys1_ctrlr` of class `ltiss1` (defined in section 4.2.1) exist. We can then define a Container called, say, `sys1` with properties `PLANT` and `CONTROLLER`, composed of the two Containers `sys1_plant` and `sys1_ctrlr`. The class `system` of Container `sys1` is defined as follows:

```
>> class('system',['PLANT|CONTROLLER'],[4,4]);
```

The class-composition hierarchy is established by defining the property PLANT and CONTROLLER using the appropriate *lower-level* Container properties as follows:

```
>> sys1('PLANT,'sys1_plant(1)','sys1_plant(2)',...
              'sys1_plant(3)','sys1_plant(4)');

>> sys1('CONTROLLER','sys1_ctrlr(1)','sys1_ctrlr(2)',...
              'sys1_ctrlr(3)','sys1_ctrlr(4)');
```

Once the hierarchy is established, meaningful queries can then be issued on the Composite Container sys1 such as:

```
>> cc=sys1(CONTROLLER(C));
```

which retrieves the C matrix in the state-space description of the controller. The four state-space matrices associated with the plant description can be retrieved by the following query:

```
>> [pa,pb,pc,pd]=sys1(PLANT);
```

which assigns the state-space matrices to the four variables pa, pb, pc and pd.

The facility of logically organizing the data in a class-composition hierarchy and issuing meaningful queries on Composite Containers is similar in some ways to what is available in the TREE structure in the Robust Control Toolbox (RCT) for Matlab. The TREE structure in RCT allows matrices along with their names and relationships (hierarchical relationships with other matrices) to be stored. If the same hierarchical structure as defined above is stored in the TREE structure, the command for retrieving (say) the C matrix of the plant would be:

```
>> pc=branch(ProjName,'plant/c');
```

where branch is a function in RCT for retrieving matrices stored in the TREE structure. The same query on a Composite Container would look like:

```
>> pc=ProjName(PLANT(C));
```

The main limitation of the hierarchical TREE data structure in RCT is its limited application. It is useful only if it is used with the SYSTEM vector defined by the function `mksys`. Matlab Containers provide a more general solution and can be used for describing and using any hierarchical structure in Matlab. Class-composition hierarchy in Matlab Containers is also structurally more powerful than the TREE structure. For example, the data values at the lowest level in the class-composition hierarchy can be retrieved by querying Container properties at a higher level such as:

```
>> [pa,pb,pc,pd]=ProjName(plant);
```

This is not possible in RCT. One will have to give the exact path as:

```
>> [pa,pb,pc,pd]=branch(ProjName,'plant/a,plant/b,...
   plant/c,plant/d');
```

or two separate commands as:

```
>> ss_=branch(ProjName,'plant');
>> [a,b,c,d]=branch(ss_,'a,b,c,d');
```

Also, if any of the constituent matrices in the TREE structure change, the SYSTEM vector as well as the tree structure in RCT have to be re-defined. In Matlab Containers, it is possible to make such a change without affecting other property definitions. Matlab Containers also provide the added advantage of 'persistence' and of storing Matlab statements in the local context and expressions in the property definitions, which is not possible in the TREE structure.

There is no limit to the levels of composition but our experience shows that most control engineering applications would require at the most five or six levels of such compositions.

In the present implementation, the only way to achieve class-composition hierarchy is to define the relationships between the corresponding properties at the property definition stage. It would be useful if the class-composition hierarchy in Matlab Containers could be established at the Container level. One way to do this would be to define a function (say) 'Composite' that would take in the higher-level Container as its first input argument and all its Composite Containers as subsequent arguments and establish the links between the higher and lower level Containers. For the above example the class-composition hierarchy could then be established in the following way:

```
>> Composite('sys1','sys1_plant','sys1_controller');
```

4.5 Computational Chains Using Matlab Containers

There is a frequent need, in CACE and in other applications, for a sequence of algorithms, with the outputs of one algorithm forming the inputs of the next one in the sequence. We call such a causal connection of ‘algorithms’ or ‘computational modules’ in series or in parallel a *computational chain*. The idea is similar to the *UNIX* ‘pipe’ model which allows the output of a process to be connected to the input of another process. But the idea of computational chains as described in this section is more powerful than pipes. Pipes can only be formed in series, whereas computational chains allow parallel connection of ‘computational modules’ and have richer interaction. The concept of computational modules is in essence similar to the *tool abstraction* paradigm [33]. That is, despite some differences, each module in a computational chain has a common structure with encapsulated ‘computations’. This gives the computational module a well-defined interaction style and makes it easy to change the function performed by the module, just as data abstraction gives a good interaction style and allows representation of data structures to change easily.

Such an analogy is also made in [29], in which a similarity between tool abstraction and re-usable modular computation sequences in ANDECS is pointed out. ANDECS uses both tool abstraction for forming computational modules in a computational chain and data abstraction for managing the evolving data chains (the data generated as a result of computation of each module). In ANDECS, the data chains are then organized in a formally defined database of abstract data types. In a computational chain for analysis and design of control systems, one would need to define abstract data types for state-space representation, frequency responses and time series data, etc. This, however, contradicts the idea of Matlab Containers which do not make any distinction between functions and data. But for computational chains we prefer to keep this distinction. We think of all Containers as ‘tools’ (even though some ‘Data Containers’ described later might actually store data values) and the results of their computations as ‘data’ (which are stored in files not Containers). This is further discussed later in this section.

GE-MEAD [104] and a more recent proposal by Kolb *et al* [49] also include some kind of computational chain facility. Van Overschee [77] describes a computational chain facility for a system identification module for use with Xmath, with a graphical interface for assembling the required algorithms. These environments have been discussed in chapter 3.

The computational chains that we have implemented are used as a tool for ‘design experimentation’ in Matlab. A special feature has been implemented which *enhances the efficiency* of each experiment run by determining which computations need to be re-executed (we shall refer to this feature as ‘computational efficiency’ of computational chains). The computational chains include enough versioning information to provide a basis for a version management system (though this has not been implemented). Such a version management system could provide facilities for maintaining a complete history of different versions of models and results generated during different experiments; navigating through such a design experimentation history; backtracking, branching and restarting experiments. We have used Containers to implement the computational chain facility primarily because they can store both data and methods and can communicate with other Containers through their properties. For this purpose we have modified our implementation of Containers so as to provide three 'built-in' classes of Containers: *Data Containers*, *Method Containers* and *Composition Containers*. Every Container which takes part in a computational chain belongs to one of these classes.

The use and implementation of computational chains will be illustrated by a chain for the synthesis and analysis of an LQG controller, shown in figure 4.1. The matrices A, B, C, D, G of the state-space model of a system being controlled,

$$\dot{x} = Ax + Bu + Gw \tag{4.1}$$

$$y = Cx + Du + v \tag{4.2}$$

are held in the Data Containers **model_a**, **model_b**, ..., **model_g**. These Data Containers may either hold the matrices explicitly, or algorithms for computing them, but as far as the computational chain is concerned, they are *sources* of data. There are four Method Containers: **rgltr** takes model data from **model_a** and **model_b** and cost function data from **cost** (which contains the Q and R matrices of the conventional LQ cost function) and uses the Control Systems Toolbox function `lqr` to compute a state feedback matrix. **estmtr** computes a Kalman filter gain, using model data from **model_a**, **model_c** and **model_g**, and noise covariance data from the Data Container **noise**. **ctrlr** assembles a complete LQG controller, given the state feedback and Kalman gain matrices; it obtains the model data from the Composition Container **model**, which makes the A, B, C, D matrices available as a single property, with cardinality 4, of a single Container, rather than as four separate properties obtained from four separate Containers. **analysis** is the final Method Container. It performs analysis on the complete closed-loop system, taking the controller description from the Method Container **ctrlr** and the plant description from the Data Container **plant**. Note that the plant may be different from the model in order to allow the effects of plant/model mismatch to be analyzed. The positive integer which appears beside the output of each Container in the figure shows the cardinality of the property being transmitted from one Container to the next.

Each Container involved in a computational chain has one `OUTPORT` property, and each Method and Composition Container has one `INPORT` property. All communication between Containers in a chain takes place through these properties. Both `INPORT`s and `OUTPORT`s have variable cardinality which is defined at the time of class definition as usual, i.e. each Container must be an instance of a class which is a sub-class of Data, Method, or Composition Containers. The `INPORT` of a Container just has an ordered list of the

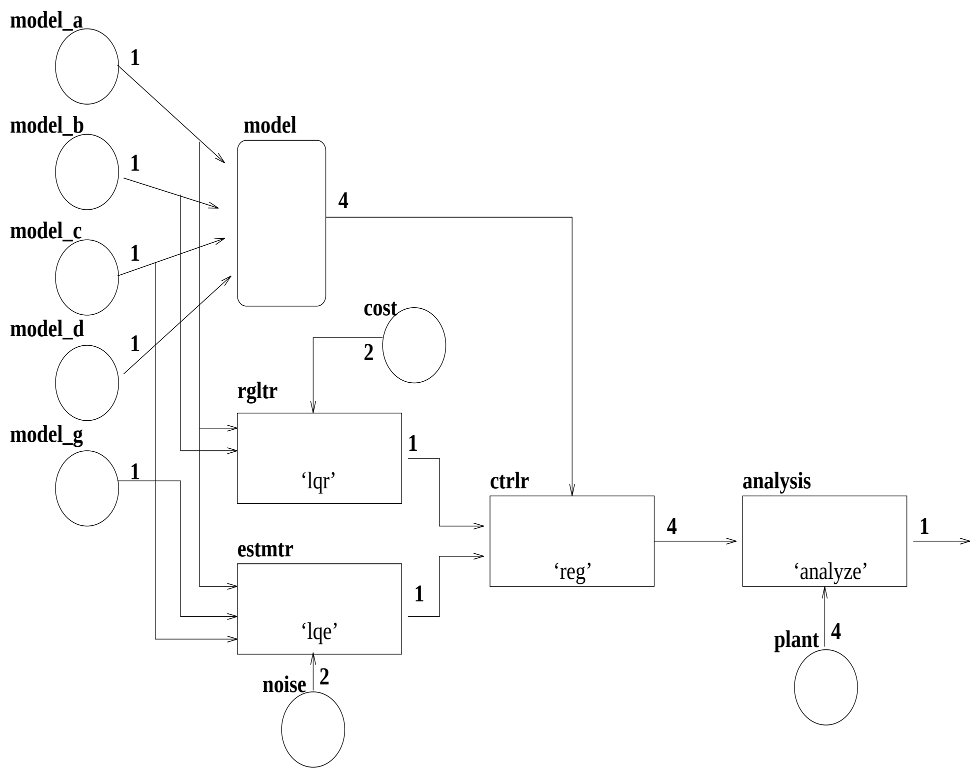

Figure 4.1: Computational chain for synthesis and analysis of an LQG controller

names of those Containers in the chain from which it requires information. (As usual for Container properties, `INPORT` may in fact contain an expression which evaluates to a list of Container names, rather than the names themselves.) A computational chain works in the following way. The user 'queries' the `OUTPORT` of a Container; for example:

```
>> result = analysis(OUTPORT);
```

That Container then obtains the information it needs by issuing similar queries to all the Containers named in its `INPORT` property and each of these Containers reacts similarly. When each Container has assembled all the information it needs, it performs any necessary computations, whereupon its `OUTPORT` property becomes defined and the calling Container's query is answered.

To save recomputation of the entire chain each time a 'get-property' query is issued, separate storage is allocated for `OUTPORT`s of each Container in the chain. `OUTPORT`s of a Container are read directly from this storage, provided that the state of that Container, and all other Containers on which its `OUTPORT` depends, are consistent with the stored values. An execution control mechanism has been implemented which keeps track of any changes made to the chain through property re-definition and determines the *paths* affected by the changes. The execution control mechanism then uses this information to decide whether to trigger a Container for computation of its `OUTPORT`s or to read the `OUTPORT` directly from storage. For example, if one of the noise covariance matrices in Container **noise** is changed and the **analysis** Container is queried again, then the previously computed state feedback matrix will be used rather than being recomputed. This is what we call the 'computational efficiency' aspect of computational chains. We will describe later how this is achieved; for the moment we note that this same mechanism is also used for version management and archiving.

Clearly there are many ways of defining a computational chain to perform a particular task. One of the key choices to be made is the level of 'granularity' at which the chain is defined. For example, in figure 4.1 would it be better to combine Containers **rgltr**, **estmtr**, and **ctrlr** into a single Method Container? The answer depends on the anticipated pattern of use—which data is likely to be changed frequently, which data is likely to be queried (for example, is the user likely to need the state feedback gain or the Kalman filter gain, or just the complete controller), where is the potential to re-use previously computed results?

The allowed structure of a computational chain is more general than shown in figure 4.1, in that several 'terminating' Containers are allowed, which do not pass results on to further Containers. The only structural feature which is not allowed is a loop since this would clearly lead to a non-terminating sequence of queries being issued to Containers. A function which traces such loops in a chain has been implemented. It takes the name of the Container that is to be queried (for results) as its input argument and traces all the paths emerging from it to see if there is a loop in any. An exact model of the allowed structure of a computational chain is a *directed acyclic graph* in which the vertices represent Containers and the edges represent connections between Containers. A directed acyclic graph has no cycles and the edges are directed and reflect a precedence relationship in terms of processing. A computational chain can therefore be modelled as a directed acyclic graph.

We now give brief details of how the execution and version control mechanism is implemented. Apart from the interface properties `INPORT` and `OUTPORT`, which have already been described, every Container in a computational chain has the properties `TYPE`, `VRSN` and `STATUS`. The property `TYPE` takes one of the values `Data`, `Method`, or `Composition`, identifying the type of Container. `VRSN` takes nonnegative integer values. It is initialized at 0 and is incremented by 1 whenever any property of that Container is set. (The property name `VERSION` cannot be used because that would clash with the name of an existing *Matlab* function.) `STATUS` indicates whether a Container's `OUTPORT` needs to be computed rather than read from storage. Whenever a property is set, `STATUS` is (automatically) set to -1. This indicates that the stored `OUTPORT` is not consistent with the current version of the Container. When the `OUTPORT` is computed, `STATUS` is set equal to `VRSN` in the case of a Data Container, but in the case of a Method or Composition Container it is set equal to the *list* of VRSN values of the immediately preceding ('upstream') Containers. When an OUTPORT property value is queried, a depth-first search of the computational chain is conducted in order to identify inconsistent Containers. If an inconsistency is detected (STATUS either with value -1 or a value different from the corresponding VRSN value(s)), the STATUS property of all Containers dependent on the inconsistent Container (i.e., those 'downstream' of it) is set to -1. When the consistency of the whole chain has been determined, response to the query is initiated, reading from storage or recomputing as appropriate.

Table 4.2 shows how the execution control mechanism works in the computational chain shown in figure 4.1. The third column in Table 4.2 shows the initial values of VRSN and STATUS. When the property OUTPORT of the **analysis** Container is queried, the fourth column shows that property VRSN in all Containers is incremented by 1 and STATUS is set to the list of values of VRSN in 'upstream' Containers. The fifth column in Table 4.2 shows how a 'set property' action in Data Containers `model_a` and `cost` affects their VRSN and STATUS properties; VRSN is incremented and STATUS is set to -1. The next time OUTPORT of Container `analysis` is queried, the inconsistency between the state of the chain and the values in the allocated storage is detected by the execution control mechanism and the values are re-computed. The last column lists VRSN and STATUS after re-computation of the OUTPORT property of all the Containers.

Method Containers have two further properties FNCTN and PARSTR. FNCTN is the name of a Matlab function which implements the algorithm executed by the Container. This may be a existing function, as in the Containers **rgltr**, **estmtr** and **ctrlr**, which use functions from the Control Systems Toolbox, or one written specially for the implementation of a Method Container, as in **analysis**. (In figure 4.1 the name of this function is shown inside the box representing each Method Container.) The property PARSTR holds names of parameters which may be passed to the function identified by FNCTN in a manner similar to the use of 'keys' in Xmath or the OPTIONS vector in the Optimization Toolbox. These parameters are evaluated in the local context of the Method Container.

Having defined the functionality of computational chains using Matlab Containers, we would like to resume the discussion on tool abstraction (discussed in the beginning of this section). All Containers (Data, Method, and Composition) in a computational module can be treated as 'tools'. The OUTPORTS of each container are stored in files. These OUTPORTS are in fact results of computation of a container and can be regarded as the 'data' associated with a container. Referring to a Data Container that includes some computations as a 'tool', and the results of its computation as 'data', is completely justified. But for cases where actual data is stored in the Data Container, referring to it as a 'tool' is not very meaningful. These Containers however also have their own allocated storage for OUTPORTS so there is some justification in calling them tools. The present implementation

Container	Type	Initial State V,S	'Get' `OUTPORT` of `analysis` V,S	'Set Property' in `model_a` & `cost` V,S	'Get' `OUTPORT` of `analysis` V,S
model_a	Data	0,-1	1,1	2,-1	3,3
model_b	Data	0,-1	1,1		1,1
model_c	Data	0,-1	1,1		1,1
model_d	Data	0,-1	1,1		1,1
model_g	Data	0,-1	1,1		1,1
model	comp	0,-1	1,[1 1 1 1]		2,[3 1 1 1]
cost	Data	0,-1	1,1	2,-1	3,3
noise	Data	0,-1	1,1		1,1
rgltr	Method	0,-1	1,[1 1 1]		2,[3 1 3]
estmtr	Method	0,-1	1,[1 1 1 1]		2,[3 1 1 1]
ctrlr	Method	0,-1	1,[1 1]		2,[2 2]
plant	Data	0,-1	1,1		1,1
analysis	Method	0,-1	1,[1 1]		2,[2 1]

Table 4.2: `VRSN` (V) and `STATUS` (S) state table for computational chain shown in figure 4.1

concentrates mainly on the tool abstraction level. Formally defining the data abstraction level, i.e. managing the data stored for each `OUTPORT` as abstract data types would be very useful for implementing a version management mechanism.

4.5.1 Data, Method and Composition Container Properties

The special 'built-in' properties in Data, Method and Composition Containers are summarized below:

- **Data Container Properties:**

  `TYPE` **(static)** is set to the string 'data' and identifies the Container as a Data Container. It has cardinality 1.

  `VRSN` **(static)** keeps track of all changes made to the Container through 'set property definition'. It is a numeric scalar quantity and has cardinality 1. Each time any property is set, `VRSN` is (automatically) incremented by 1; thus representing a new version.

  `STATUS` **(static)** indicates whether 'own' stored `OUTPORT`-s need to be updated. Whenever a property is set, `STATUS` is (automatically) set to -1. The value '-1' in `STATUS` indicates that the stored `OUTPORT`-s are no longer consistent with the present version of the Container. If the stored `OUTPORT`-s are consistent with the present version of the Container, then `STATUS` has the same value as `VRSN`.

  `OUTPORT` **(dynamic)** exports numeric values of specific data components stored in the Data Container. In a computational chain, these numeric values are exported to a Method Container. The cardinality for `OUTPORT` is not fixed. It depends on the number of numeric components (matrices in Matlab) associated with the Data Container and can therefore vary from one class of Containers to another.

- **Method Container Properties:**

  `TYPE` **(static)** is set to the string 'method' and identifies the Container as a Method Container. Its cardinality is 1.

  `VRSN` **(static)** is similar to `VRSN` in Data Containers.

  `STATUS` **(static)** similar to `STATUS` in Data Containers and indicates the consistency or inconsistency of own `OUTPORT`-s not only with respect to its own version but also for versions of Containers defined in `INPORT`-s.

  `FNCTN` **(static)** holds the name of Matlab function as a string and has cardinality 1. The function is called whenever `OUTPORT`-s are recalculated.

`INPORT` **(static)** is a list of Containers that supply Method Container with input data. The cardinality of `INPORT` is equal to the number of Containers the data is imported from. This data then forms the input for the function defined in `FNCTN`.

`OUTPORT` **(dynamic)** exports values returned after execution of function defined in `FNCTN`. The cardinality for `OUTPORT` is not fixed and depends on the number of output arguments associated with the function defined in `FNCTN`.

`PARSTR` **(static)** holds parameter names and is an optional property. The parameters are assigned values in the local context. The definition of `PARSTR` follows a special syntax. The variables are separated by a ',' and their order is the same as it would be in the call to the function defined in `FNCTN`. The parameters appear as the last input arguments in the function call.

- **Composition Container Properties:**

`TYPE` **(static)** is set to the string 'comp' (for soft Composition) or 'Comp' (for hard Composition). This identifies the Container as a soft Composition Container or a hard Composition Container.

`VRSN` **(static)** similar to `VRSN` in Data Containers.

`STATUS` **(static)** similar to `STATUS` in Method Containers.

`INPORT` **(static)** a list of component Containers (of any `TYPE`) which collectively form the composition. The cardinality of `INPORT` is equal to the number of component Containers.

`OUTPORT` **(dynamic)** returns the matrices associated with the composed Containers. The cardinality of `OUTPORT` is equal to the number of matrices in the component Containers.

4.5.2 Example Statements for Creating Computational Chains

The construction of the computational chain given in figure 4.1 is explained with the help of some example statements. The statements are not preceded by the prompt symbol `>>` because these statements are normally a part of a Matlab script file.

The Data Container `noise` is created as follows:

```
class('data2',['-TYPE|-VRSN|-STATUS|OUTPORT'],[1 1 1 2]);
data2('noise');
noise('LOCAL',' Q = [.1 0 0 0 ; 0 100 0  0');
noise('LOCAL','       0 0 1 0 ; 0  0  0 100];');
```

```
noise('LOCAL',' R = [1000 0 ; 0 1]; ');
noise('TYPE|VRSN|STATUS|OUTPORT','data','0','-1','Q','R');
```

The local context of the Data Container `noise` stores the data matrices 'Q' and 'R'. These can be accessed through the property `OUTPORT` which has the names of these matrices as its property definition.

The following statements create the Composition Container `model` and the links with its source Containers:

```
class('cmpstn',['-TYPE|-VRSN|-STATUS|-INPORT|OUTPORT'],...
      [1 1 1 4 4]);
cmpstn('model');
model('TYPE|VRSN|STATUS|INPORT','comp','0','-1','model_a',...
      'model_b','model_c','model_d');
```

The connections between source Containers and destination Containers are established by setting the property `INPORT` of the destination Container to the name of the source Container(s). The source Containers for Composition Container `model` are `model_a`, `model_b`, `model_c` and `model_d`. These are given as definitions of property `INPORT` in Composition Container `model`. Similarly, the Method Container `rgltr` gets its inputs from the Data Containers `model_a`, `model_b` and `cost`. These Data Container names are set as definitions of the property `INPORT` in Method Container `rgltr` as shown in the statements below.

```
class('meth3to1',...
     ['-TYPE|-VRSN|-STATUS|-FNCTN|-INPORT|OUTPORT'],...
     [1 1 1 1 3 1]);
meth3to1('rgltr');
rgltr('TYPE|VRSN|STATUS|FNCTN|INPORT','method',...
      '0','-1','lqr','model_a','model_b','cost');
```

The cardinality of the property `INPORT` of the destination Container is equal to the number of source Containers and is independent of the actual number of matrices passing from each source Container to the destination Container. Method Container `rgltr` has three source Containers, therefore the cardinality of `INPORT` in `meth3to1` (class of Container `rgltr`) is 3. The order in which the source Container names are specified in the destination Container is important because it determines how the different inputs are used in the function defined in `FNCTN` in the destination Container. The class definition for Method and Composition Containers would differ in the cardinality of `INPORT` and `OUTPORT`. For Data Containers the class definition would differ only in the cardinality of `OUTPORT`. The class name meth3to1 has been chosen to stand for all Method Containers with INPORT cardinality 3 and OUTPORT cardinality 1.

Parameters for Method Containers can be defined in the property PARSTR. The Method Container analysis has one parameter freq which forms the last input to the function. Properties for Method Container analysis are defined as follows:

```
analysis('LOCAL','freq=logspace(-2,0,100);');
analysis('TYPE|VRSN|STATUS|FNCTN|INPORT|PARSTR',...
       'method','0','-1','analyze','ctrlr','plant','freq');
```

Parameters can also be defined as separate Data Containers and their values can then be imported into the Method Container through the property INPORT. This approach is useful when a parameter is shared by several Method Containers.

4.5.3 Implementation

Several modifications had to be made in the script file gen_cont.m (see section 4.2.2) in order to implement computational chains within the present framework of Matlab Containers. These modifications mostly involved recognition of special properties for the different types of Containers—Data, Method and Composition Containers—so that if any property in these special Containers in a computational chain is reset, gen_cont knows that VRSN needs to be incremented and STATUS needs to be set to -1. The most important modifications, however, involved implementation of the execution control and version control mechanisms. For this purpose three new 'generic' script files were written: gen_data.m (for *generic data*), gen_meth.m (for *generic method*) and gen_comp.m (for *generic composition*). These new generic script files are called from within gen_cont. They perform several functions such as allocating storage for results of each Container, creating *branch* and *path* stacks to determine which paths (if any) need to be re-executed and the paths for which data can be retrieved from allocated storage. They are also responsible for the 'flow of data' in a computational chain and for function execution defined in the property FNCTN in Method Containers.

4.5.4 Comparison with a Script File

The advantages of using computational chains in terms of managing different experiment runs and 'computational efficiency' become clearer if they are compared to the functionality of a Matlab script file.

After the execution of a typical script file for any design or analysis session, the workspace in Matlab generally fills up with several variables of the same type (the complex matrix type supported by Matlab). The only distinguishable factor for these different variables is the variable name. Computational chains offer a solution to this problem at two different levels. Firstly, only property names appear in the workspace because all variables in the statements in the local context and the expressions that form the property definitions are local in scope. Secondly, the computational 'task' is divided into separate computational modules (Data, Method or Composition Containers). So, if an intermediate result is required, one only needs to query the `OUTPORT` property of the appropriate Data, Composition or Method Container. Therefore, from the point of view of work space organization, computational chains offer a compromise between script files that 'fill up' a work space and functions that fully encapsulate the computational 'task' so that all intermediate results are inaccessible (because they have local scope). Of course, the intermediate values generated inside each Container are inaccessible as in a function, but this is acceptable, because it is assumed that the Containers are defined at the level at which the results (`OUTPORT`s of Containers) are useful to the user. This issue has been discussed earlier in terms of the level of 'granularity' at which the chain is defined.

All changes made to a computational chain are persistent, therefore an audit trail of the state of a chain exists even if one quits the Matlab environment and starts it again. This gives a 'full documentation' of data, algorithms and parameters used in a session. A script file can also contain such a documentation, but in a computational chain it is more 'structured'. For example, in a computational chain a parameter is always associated with a certain Method Container whereas in a script file a parameter can appear anywhere as long as it is defined before the function call where it is used.

The use of `INPORT` and `OUTPORT` properties for passing messages between different Containers in a computational chain does not require specification of input and output arguments as in function calls in a script file. The problem is resolved by connecting the `OUTPORT`s of 'source' Containers to `INPORT`s of 'receiving' Containers forming a kind of a channel between two Containers through which the data can pass directly. This is very useful, especially in cases where one would normally have to specify long argument lists in function calls.

Computational chains, with their ability to logically organize the computations in different Containers (which are then collectively used for an experiment run), can form a basis for a version management mechanism which, as discussed in chapter 3, is an important requirement for CACE environments. A script file does not have the kind of structure for organizing computations for different experiment sessions on which a version management system could be built.

The most important advantage, however, is what we call 'computational efficiency', i.e. the ability of computational chains to remember that a computation has been performed and its results are available for further use. This saves considerable time in compute intensive jobs. In a script file each modification would require re-computation of the entire script file.

4.6 Summary

In this chapter, a new concept called Matlab Containers has been explained and the use of Containers in control engineering applications has been explored. Some other functionalities such as class-composition hierarchy and computational chains which have been built on top of Matlab Containers are also explained. Matlab Containers provide a general solution based on object-oriented concepts for the systematic creation and manipulation of data, functions and executable sequences of functions (which we call computational chains). Other important features they support include persistence of all user actions, flexibility and object-oriented style of interaction. Class-composition hierarchy is useful for structuring and retrieving complex system representations. Computational chains considerably enhance the organization and efficiency of computational 'tasks' and can form the basis for a proper version management mechanism.

Since Containers can store both data and functions, they provide data abstraction as well as tool abstraction. However, in a computational chain each Container is strictly a 'tool abstraction'.

All programming constructs are expressed in, and are consistent with, current Matlab syntax, which makes the use of Matlab Containers relatively straightforward. The functionality of the proposed solution indicates possible improvements of Matlab itself. However, the purpose of the whole exercise was to explore the usefulness of such a feature rather than to suggest any one particular implementation.

Chapter 5

Case Study

The advantages of Containers become more apparent as the complexity of the work undertaken with them increases, so in this chapter we describe a relatively complex 'case study', which we think is fairly typical of parts of industrial control systems design and analysis. It is based on the design and analysis of a multivariable Smith predictor for a multivariable plant which contains time delays. We shall discuss the use of Matlab Containers both with and without computational chains and the advantages of establishing a class-composition hierarchy for representing different system descriptions.

5.1 The Process

The case study is based on the so-called 'Shell process control problem' [81], which has become well known as a 'benchmark' problem. The plant is a distillation column with 7 outputs and 5 inputs, though in this example we will use only 3 inputs and 2 outputs. This will be enough to demonstrate the utility of Containers, without introducing unnecessary complexity. A linearized model of the distillation column is given as a matrix of transfer functions, each transfer function being a first-order lag with a time delay [81]:

$$P(s) = \begin{bmatrix} \frac{4.05e^{-27s}}{1+27s} & \frac{1.77e^{-28s}}{1+60s} & \frac{5.88e^{-27s}}{1+50s} \\ \\ \frac{5.39e^{-18s}}{1+50s} & \frac{5.72e^{-14s}}{1+60s} & \frac{6.90e^{-15s}}{1+40s} \end{bmatrix}$$

In [60], a controller is designed for this plant with the structure of a multivariable Smith predictor. The controller is designed using LQG/LTR technique. The controller performance is analyzed for the nominal plant model and for the case where the plant deviates from the assumed model.

5.2 Multivariable Smith Predictors

A multivariable Smith predictor controller has the structure shown in figure 5.1 [76].

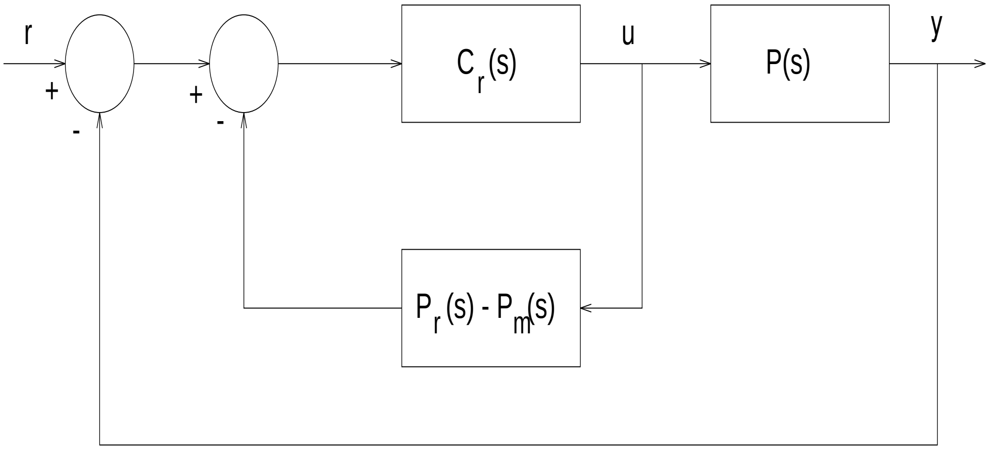

Figure 5.1: The multivariable Smith predictor

The plant is assumed to be stable and is described by a transfer function matrix $P(s)$. There is a time delay T_{ij} between input j and output i, and the plant transfer function matrix can be factored as: $P(s) = P_r(s). * D(s)$ where '.∗' denotes element-wise multiplication, the (i, j) element of $D(s)$ is of the form $\exp(-sT_{ij})$ and $P_r(s)$ has only rational elements. In other words, the time delays are the only sources of irrational elements in $P(s)$, and $P_r(s)$ is the 'rational part' of the plant. $P_r(s)$ is a 'predictor', since each of its outputs is composed of predictions of signals which compose the outputs of the plant. $C_r(s)$ is the transfer function matrix of any feedback controller which stabilizes $P_r(s)$, and it is usually rational. Indeed, the major reason for introducing the Smith predictor structure is that $C_r(s)$ can be designed using standard techniques for rational systems, such as LQG. In fact, we shall use the LQG/LTR technique to design the controller (see section 5.3.1). The complete Smith predictor controller consists of a feedback system formed from $C_r(s)$ and the difference of the predictor $P_r(s)$ and the plant model $P(s)$, as shown in figure 5.1.

5.3 Design and Analysis of Multivariable Smith Predictors Using Computational Chains

In this section, the use of computational chains (see section 4.5) is discussed for design and analysis of multivariable Smith predictors.

5.3.1 Data and Method Containers for Plant, Plant Model and Predictor

Let us first look at how the various components of a Smith predictor controller and the plant being controlled can be represented in a computational chain. Most of these components are linear systems with time delays. A suitable representation for systems with time delays, compatible with Matlab is available in the Time Delay Toolbox [99]. This toolbox supports *delay matrices* and *delay multivariable systems*; the latter can be used to represent the systems which appear in the multivariable Smith predictor. A delay matrix represents an object with transfer function $\sum_{i=1}^{N} M_i e^{-h_i s}$ where each M_i is a matrix and each h_i is a delay. The Time Delay Toolbox representation of a delay matrix is a list of matrices (the M_i's) and a vector of delays (the h_i's). A delay multivariable system represents a system defined by

$$\begin{aligned} sEx^{(1)} &= A^{(11)}x^{(1)} + A^{(12)}x^{(2)} + B^{(1)}u \\ x^{(2)} &= A^{(21)}x^{(1)} + A^{(22)}x^{(2)} + B^{(2)}u \\ y &= C^{(1)}x^{(1)} + C^{(2)}x^{(2)} + Du \end{aligned}$$

where each of $A^{(ij)}$, $B^{(i)}$, $C^{(i)}$, and E is a delay matrix, and D is an ordinary matrix.

In the Shell process control problem, the plant $P(s)$ is given as a delay multivariable transfer function. A Data Container `plant` is defined which stores this representation in the form of a numerator polynomial matrix, denominator polynomial matrix, the degree of polynomial and the delay matrix and then uses the function `dmtf2dmv` to convert the delay multivariable transfer function to delay multivariable system representation. It is defined as follows:

```
class('data10',['-TYPE|-VRSN|-STATUS|OUTPORT'],...
      [1,1,1,10]);
data10('plant');
plant('LOCAL','NUM=[0 4.05 0 1.77 0 5.88;...
        0 5.39 0 5.72 0 6.90];');
plant('LOCAL','DEN=[27 1 60 1 50 1;50 1 60 1 40 1];');
plant('LOCAL','DEG=1;');
```

```
plant('LOCAL','DEL=[27 28 27;18 14 15];');
plant('LOCAL','[CMA,DVA,CMB,DVB,CMC,DVC,CMD,DVD,CME,DVE]=...
       dmtf2dmv(NUM,DEN,DEG,DEL);');
plant('TYPE|VRSN|STATUS|OUTPORT','data','0','-1','CMA',...
      'DVA','CMB','DVB','CMC','DVC','CMD','DVD',CME','DVE');
```

The plant model is $P_m(s)$ which we have hitherto assumed is the same as $P(s)$. But it is sometimes useful to allow the plant and its internal model to differ, for example in robustness studies) is represented as a separate Data Container `model`.

The predictor $P_r(s)$ is the delay-free part of the plant model and is represented by a Method Container `predictor` with property `FNCTN` defined as `dmvzd`. `predictor` takes its inputs from the Data Container `plant`. When queried for `OUTPORT`-s it produces the required representation for the predictor.

```
predictor('TYPE|VRSN|STATUS|FNCTN|INPORT',...
     'method','0','-1','dmvzd','model');
```

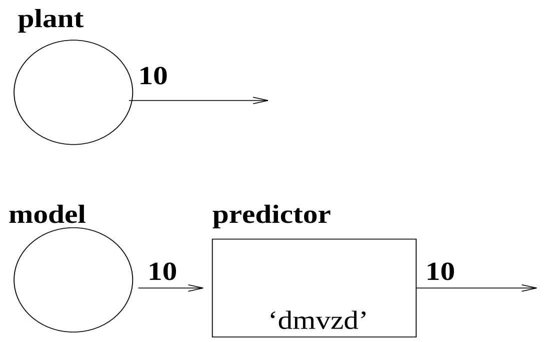

Figure 5.2: Data and Method Containers for plant, plant model and predictor

The Containers which we have described so far serve to hold definitions of components of feedback systems in the form of Smith predictors. They illustrate part of the use of Data, Composition and Method Containers in a computational chain and are probably helpful for any given analysis of Smith predictors. They could be supplied as a part of a *Smith Predictor Toolbox*, for example.

5.3.2 Computational Chains for LQG/LTR Design of Predictor Controller

A computational chain for designing the predictor controller $C_r(s)$, using the LQG/LTR procedure, is similar to the chain for LQG design discussed in section 4.5. The difference is that in LQG/LTR the cost weights Q and R, and the covariance matrices W and V are iterated in a particular way on the basis of computed frequency response properties. (For details see chapter 5 of [57].) In particular, a common choice for the cost function is to set $G = B$ and $Q = C^T C$. The Container `cost` therefore becomes a Method Container, rather than a Data Container, as shown in figure 5.3, since it obtains B and C from the Container `predictor`.

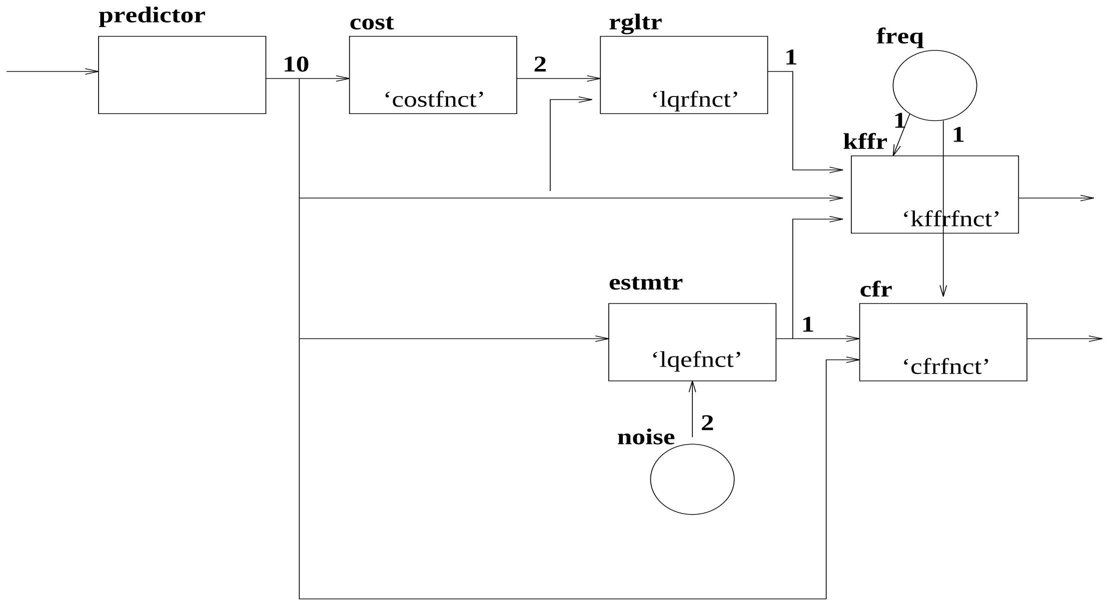

Figure 5.3: Computational chain for LQG/LTR design of $C_r(s)$.

The noise covariance matrices W and V are chosen by examining the frequency response of the Kalman filter return ratio

$$cfr = C(sI - A)^{-1}L \tag{5.1}$$

where L is the Kalman filter gain. Generally one starts with simple choices of W and V, such as $W = I$ and $V = I$, and modifies them until the frequency response of cfr is satisfactory. Once a satisfactory Kalman filter return ratio has been obtained, the next step is to 'recover' this return ratio at the output of the plant. This is done by setting $R = \rho I$ (in the cost function), and reducing ρ towards zero. The final value for ρ is selected by examining the frequency response of the return ratio evaluated at the plant output, which is

$$kffr = -K(sI - A + BK + LC)^{-1}LC(sI - A)^{-1}B \tag{5.2}$$

where K is the state feedback gain matrix.

To support this design process, two new Method Containers, `cfr` and `kffr`, have been introduced to compute and store the return ratios (5.1) and (5.2), respectively. This is shown in figure 5.3. Also a new Data Container, `freq`, is used to generate the list of frequencies at which all frequency responses are computed. The statements for building the computational chain are given below:

```
freq('LOCAL','w=logspace(-2,0,200);');
freq('TYPE|VRSN|STATUS|OUTPORT','data','0','-1','w');

noise('LOCAL',' W=eye(3); V=eye(2);');
noise('TYPE|VRSN|STATUS|OUTPORT','data','0','-1','W','V');

estmtr('LOCAL','G=feval(''predictor'',''OUTPORT(3)'');');
estmtr('TYPE|VRSN|STATUS|FNCTN|INPORT|PARSTR',...
   'method','0','-1','lqefnct','predictor','noise','G');

kffr('TYPE|VRSN|STATUS|FNCTN|INPORT','method','0',
   '-1','kffrfnct','rgltr','predictor','estmtr','freq');

cost('LOCAL','rho=0.03;');
cost('TYPE|VRSN|STATUS|FNCTN|INPORT|PARSTR','method',
   '0','-1','costfnct','predictor','rho');

rgltr('TYPE|VRSN|STATUS|FNCTN|INPORT','method','0',
   '-1','lqrfnct','predictor','cost');

cfr('TYPE|VRSN|STATUS|FNCTN|INPORT','method','0',
   '-1','cfrfnct','predictor','estmtr','freq');
```

The function `lqrfnct` in `rgltr` calls the Control Systems Toolbox function `lqr` and the function `lqefnct` calls the Control Systems Toolbox function `lqe`. Functions `cfrfnct` and `kffrfnct` both call the Multivariable Frequency Domain Toolbox function `mv2fr`. Function `costfnct` computes the cost function matrices Q and R.

Once the computational chain is defined, a Bode plot of the `OUTPORT` of the Method Container `cfr` can be plotted as follows:

```
plotdb(feval('freq','OUTPORT'),feval('cfr','OUTPORT'));
```

This uses the Multivariable Frequency Domain Toolbox function `plotbode`, which has the syntax

`plotbode(`*frequency list, frequency response*`)`

The point to note here is that the use of `feval` forces evaluation of the Container property; otherwise only the 'symbolic' definition of the property would be returned.

Let us say that after inspecting the Bode plot the designer wishes to change the value of the parameter *rho* in the Method Container `cost` from 0.03 to 0.01. This can easily be done by a statement such as:

```
cost('[LOCAL,-1]','rho=0.01;');
```

which sets $\rho = 0.01$. As a result of this redefinition, the `VRSN` of the Method Container `cost` is incremented and its `STATUS` property is set to -1. So the next time the `OUTPORT` of `cfr` is queried:

```
plotdb(feval('freq','OUTPORT'),feval('cfr','OUTPORT'));,
```

all Container `OUTPORT`-s *upstream* of `cfr` are re-computed.

After setting the parameter 'rho' and the noise covariance matrices W and V, the state-space realization for the predictor controller $C_r(s)$ is computed. This is obtained by another computational chain, as shown in figure 5.4. Here the Method Container `pred_ctrlr` computes the realization by querying the Containers `predictor`, `rgltr` and `estmtr`. Note that these Containers are shared by the chains shown in figures 5.3 and 5.4 — in fact, the two chains could be amalgamated into a single computational chain. The simple chain shown in figure 5.4 is created by defining the Container `pred_ctrlr`:

```
>> data10(pred_ctrlr);
>> pred_ctrlr('TYPE|VRSN|STATUS|FNCTN|INPORT','method',
   '0','-1','regfnct','predictor','rgltr','estmtr');
```

where `regfnct` is a suitable Matlab function.

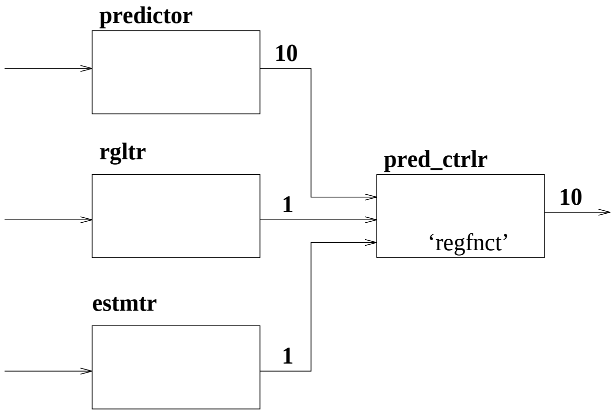

Figure 5.4: Computational chain for the realization of $C_r(s)$

5.3.3 Computational Chain for Sensitivity Analysis of Multivariable Smith Predictors

We now look at other Containers we require to perform sensitivity analysis. Containers for performing analysis, however, are less likely to be universally applicable because they have to be tailored to the analysis which is required for a particular project or by a particular organization. The closed-loop sensitivity $S(s)$ is given by:

$S(s) = I - T(s)$,

where $T(s)$ is the complementary sensitivity given by:

$T(s) = P(s)C_r(s)[I + P_r(s)C_r(s)]^{-1}$

The Multivariable Frequency Domain Toolbox functions defined in the property `FNCTN` in different Method Containers are `faddf` (adds two frequency responses together), `fmulf` (forms the product of frequency responses) and `ffbf` (which forms the response of the feedback combination of two responses).

Statements for building the chains for complementary sensitivity and sensitivity are as follows:

```
%chain for complementary sensitivity
pdf('TYPE|VRSN|STATUS|FNCTN|INPORT',...
    'method','0','-1','dmv2fr','predictor','freq');
pcf('TYPE|VRSN|STATUS|FNCTN|INPORT',...
    'method','0','-1','dmv2fr','pred_ctrlr','freq');
plf('TYPE|VRSN|STATUS|FNCTN|INPORT',...
    'method','0','-1','dmv2fr','plant','freq');
```

```
pdpcf('TYPE|VRSN|STATUS|FNCTN|INPORT',...
      'method','0','-1','ffbf','freq','pcf','pdf');
cmpsns('TYPE|VRSN|STATUS|FNCTN|INPORT',...
       'method','0','-1','fmulf','freq','plf','pdpcf');
```

Figure 5.5 shows the different Containers in the computational chain for complementary sensitivity and figure 5.6 shows the computational chain for sensitivity.

```
%chain for sensitivity
cmpidt('TYPE|VRSN|STATUS|FNCTN|INPORT|OUTPORT',...
       'method','0','-1','eyef','freq','cmpsns','-OUTPORT_');
snst('TYPE|VRSN|STATUS|FNCTN|INPORT',...
     'method','0','-1','faddf','freq','cmpidt','cmpsns');
```

Figure 5.5 shows the different Containers in the computational chain for complementary sensitivity and figure 5.6 shows the computational chain for sensitivity.

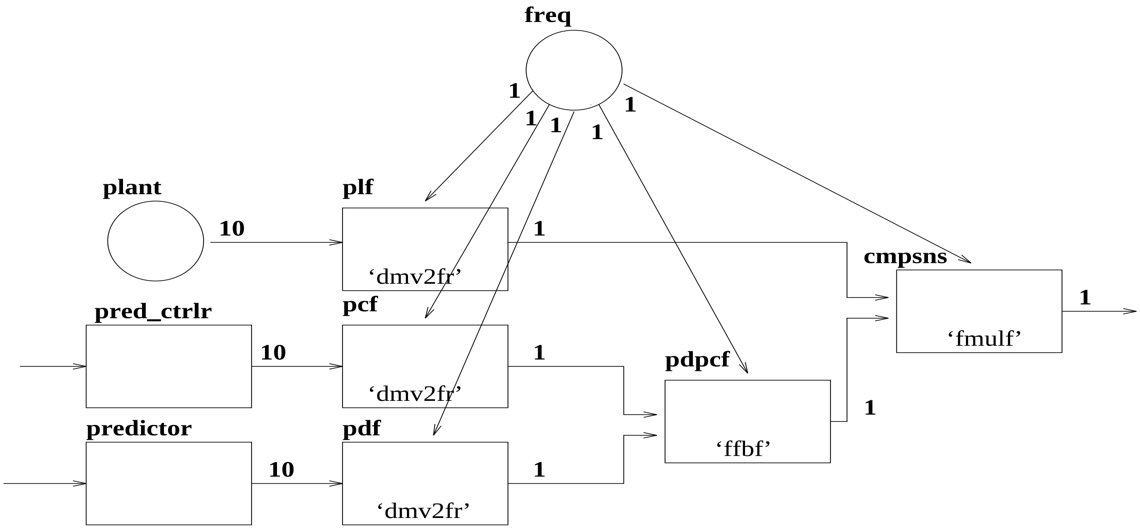

Figure 5.5: Computational chain for complementary sensitivity

Once the chain is built, the `OUTPORT` of Method Container `snst` can be queried as follows:

```
   %For the nominal plant
>> snstf_nom=snst('OUTPORT');
```

It is now very easy to perform 'what-if' analyses on the designed controller. Suppose that the performance of the controller is to be examined if the delays in the plant vary by ±20%

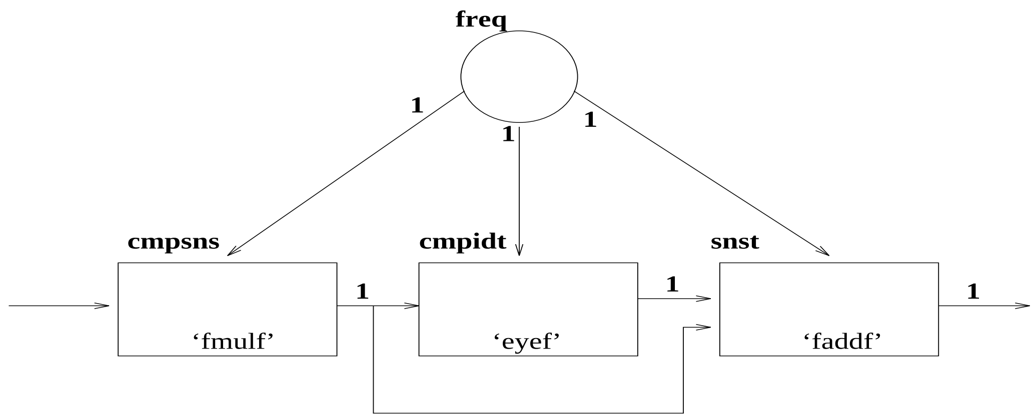

Figure 5.6: Computational chain for sensitivity

from their nominal values. Then all that has to be done is to change the plant definition in the Data Container `plant` and then query the `OUTPORT` of the Method Container `snst`.

```
  %The delays in the Plant increase by 20%
>> plant('[LOCAL,4]','DEL=1.2*DEL;');
```

The Data Container `plant` would now have the statement DEL=1.2*DEL inserted after the 4th line (line 4 has the definition for the delay vector DEL) in the local context. As a result of the change in the local context, the `VRSN` of Data Container `plant` is incremented and its `STATUS` is set to -1. So the next time `OUTPORT` of `snst` is queried:

```
>> snstf_incdel=snst('OUTPORT');
```

the execution control mechanism detects the change and all Containers *upstream* of the Data Container `plant` are recomputed.

To examine the performance of the controller when the delays in the plant decrease by 20%, the statement DEL=1.2*DEL (which is on line 5 now) can be easily replaced by the statement DEL=0.8*DEL as follows:

```
  %The delays in the Plant decrease by 20%
>> plant('[LOCAL,-5]','DEL=0.8*DEL;');
```

Again as a result of this change in the local context, the `VRSN` of the Data Container `plant` is incremented and its `STATUS` is set to -1. So the next time the `OUTPORT` of `snst` is queried:

```
>> snstf_decdel=snst('OUTPORT');
```

the execution control mechanism again detects the change and all Containers *upstream* of the Data Container `plant` in the computational chain are recomputed.

The three graphs shown in Figure 5.7, which show the maximum singular values of the sensitivity function for the three cases, can now be obtained very easily using a few standard Matlab and Multivariable Frequency Domain Toolbox or Control Systems Toolbox commands.

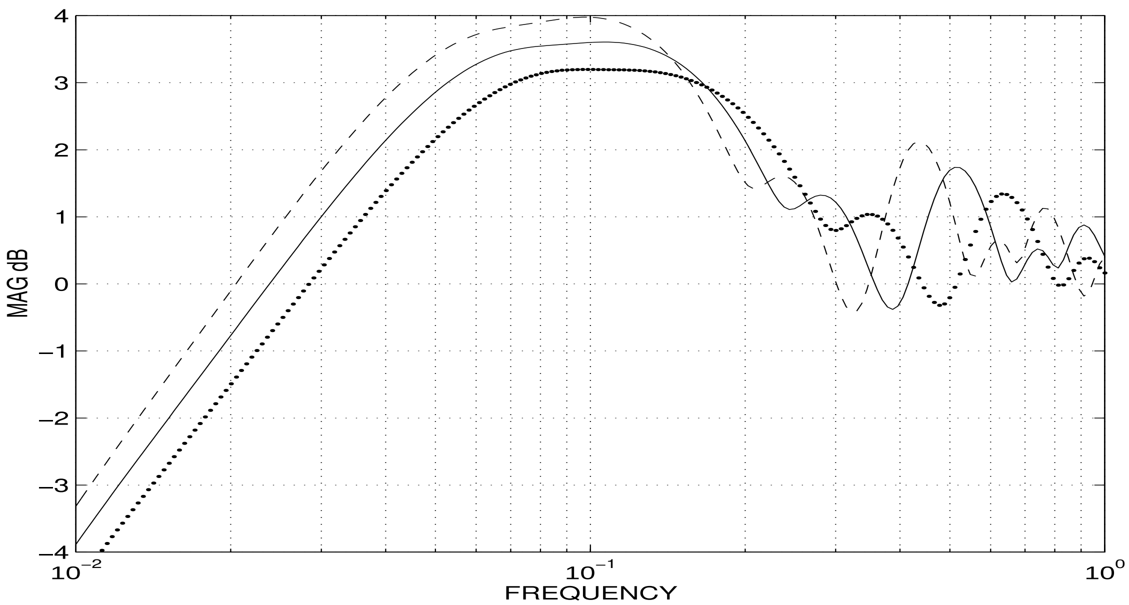

Figure 5.7: Sensitivities for nominal (continuous), long (dashed), and short (dotted) delays

The Data Containers described above are probably helpful whatever analysis of Smith predictors is required. They could be supplied as part of a *Smith Predictor Toolbox*, for example. The Method Containers for performing analysis, however, are less likely to be universally useful because they depend on the analysis which is required for a particular project or by a particular organization.

5.4 Analysis of Multivariable Smith Predictors Using Matlab Containers

So far we have discussed the use of special 'built-in' classes of Matlab Containers that form computational chains. In this section, we explore how Matlab Containers can be used on their own (without computational chains) for analysis of multivariable Smith predictors. Such a use of Matlab Containers has been discussed in detail in [89].

Two classes of Containers, `smith_analyze_d` and `smith_analyze_f`, can be defined for the purpose of analysis as follows:

```
>> class('smith_analyze_d',['CONTROLLER|SENSITIVITY|
   CMPL_SENSITIVITY'],[5,5,5]);
>> class('smith_analyze_f',['FREQUENCY|CONTROLLER|
   SENSITIVITY|CMPL_SENSITIVITY'],[1,1,1,1]);
```

The properties of the former are delay multivariable systems, while those of the latter are multivariable frequency responses. The difference in the cardinalities shows the different representations. The additional property `FREQUENCY` in `smith_analyze_f` represents the vector of frequencies at which all the frequency response properties are calculated.

The type of class used, i.e. whether the properties are computed as delay multivariable systems or as frequency responses depends on several factors such as number of states, inputs and outputs and the length of the frequency vector. These in turn determine whether the computation is 'compute intensive' or 'storage intensive'. For example, for a large number of inputs and outputs, the frequency responses require a considerable amount of storage space but the compute time is not intensive. So in this case it is better to compute properties as delay multivariable systems. On the other hand, if the number of states is quite large the compute time for frequency response calculation becomes quite intensive but the storage requirement is not high. In this case, it is better to compute properties as frequency responses which can be stored for future use.

The local contexts of Containers classes `smith_analyze_d` and `smith_analyze_f` contain statements which retrieve the basic system descriptions from Containers holding descriptions of the plant, plant model, predictor and predictor controller. These system descriptions are then processed by functions from the Time Delay Toolbox or the Multivariable Frequency Domain Toolbox as appropriate, either in the local context (in which case the computation is performed on every query of that Container) or in the property definitions (in which case the computation is performed only when that particular property is queried). For example, a Container `sm_analyze_f` of class `smith_analyze_f` can have a local context that looks like:

```
(14)    [pla,pldva,plb,pldvb,plc,pldvc,pld,pldvd,ple,
        pldve]=plant('DMVSYS');
        .
        .
(17)    w=eval(['feval(SELFNAME,''-FREQUENCY'')']);
(18)    P=dmv2fr(pla,pldva,plb,pldvb,plc,pldvc,pld,...
        pldvd,ple,pldve,w);
        .
        .
(26)    T=fmulf(w,P,ffbf(w,Cr,Pr)); % Compl. sensitivity
(27)    It=eyef(w,T);     % Identity in MVFR form
        .
        .
(32)    Ct=ffbf(w,Cr,faddf(w,Pr,-Pm));  % Controller
```

The local context computes the plant description as a delay multivariable system (for this it passes a 'get property message' to the Container `plant` to retrieve the matrices associated with its `DMVSYS` property) and then as a multivariable frequency response. The complementary sensitivity is computed as a frequency response. Some of the local context has been omitted here. The functions `fmulf` (forms the product of frequency responses) and `ffbf` (which forms the response of the feedback combination of two responses) are taken from the Multivariable Frequency Domain Toolbox. The property definitions of the same Container may then be:

```
FREQUENCY        :
 (1)
CONTROLLER       :
 (1)   Ct
SENSITIVITY      :
 (1)   faddf(w,I,-T)
CMPL_SENSITIVITY:
 (1)   T
```

which shows the property definitions for the class `smith_analyze_f`. The function `faddf` is taken from the Multivariable Frequency Domain Toolbox and adds two frequency responses together.

The `FREQUENCY` property for the Container `sm_analyze_f` can be set, and the `SENSITIVITY` retrieved by the following two statements:

```
>>sm_analyze_f(FREQUENCY,'logspace(-3,0);');
```

```
>>snstf = sm_analyze_f(SENSITIVITY);
```

A number of Containers can be defined for the class `smith_analyze_f`. The local context and the property definitions can vary from Container to Container depending on how the system definitions are retrieved (whether they are retrieved from Containers holding these definitions or from data files), on the statements that a user wishes to keep in the local context, or on the expressions that a user would like to put as property definitions, etc.

The main advantages of this approach over using a Matlab function or a script file are that the analysis results can be queried using meaningful ‘ get property value’ statements so the user does not have to remember the exact sequence of arguments that is required in a function call and that all ‘user actions’ and the source data for computations are persistent. Using a computational chain for such analysis, however, considerably adds to the advantages of using Matlab Containers on their own. Although it is simpler to set up a Container like `sm_analyze_f` than to define a computational chain for the purpose, such a use of Matlab Containers does not offer the advantages of storage, execution and version control mechanisms that are available when computational chains are used. Each time a property is queried in `sm_analyze_f`, the local context is executed again. One could introduce a property `FILENAME` for storing the values of a specific run but the user would have to be careful about the consistency of the stored data, i.e. the user will have to decide whether the data in a previously written file is still valid or not. Also, since a significant part of the algorithm is encapsulated inside the local context, it is not possible to inspect the different intermediate values whereas in computational chains it is possible to inspect the `OUTPORT`s of the different Containers in the chain.

5.5 Class-Composition Hierarchy for Multivariable Smith Predictor Structure

In this section, a class-composition hierarchy (see section 4.4) for a multivariable Smith predictor structure is defined. The complete multivariable Smith predictor can be seen as being composed of a plant and a controller which in turn is composed of a plant model, a predictor and a predictor controller. Since most of these systems are linear systems with time delays, we can represent them all as delay multivariable systems; the systems without delays will have their delay vectors set to zero. As stated earlier, a suitable representation for a delay multivariable system is given in the Time Delay Toolbox [99], where it is defined as a composition of five delay matrices where each delay matrix is composed of a matrix of coefficients and a delay vector. These representations add two more levels to the class-composition hierarchy. Abstract definitions of the delay multivariable system and the delay matrix are given here:

1. *Delay matrix* has *properties*:
   - list of coefficient matrices (of equal size)
   - list of delays (a delay vector)
2. *Delay multivariable system* has *properties*:
   - right state matrix (a delay matrix)
   - input matrix (a delay matrix)
   - output matrix (a delay matrix)
   - direct input-output matrix (a delay matrix)
   - left state matrix (a delay matrix)

Based on the above definitions, two classes `dm` for delay matrix and `dmv` for delay multivariable system are defined as:

```
>>class('dm',['CM';'DV'],[1,1]);
>>class('dmv',['DMA';'DMB';'DMC';'DMD';'DME'],[2,2,2,2,2]);
```

Each element in the cardinality list of the class `dmv` is 2 because each property of the class `dmv` is *composed-of* the class `dm` which has two properties `CM` and `DV` both of cardinality 1. The above class-definition commands do not establish the desired class-composition hierarchy between the two classes. The commands only create two separate classes with the associated properties and property cardinalities. The hierarchy between them, i.e. each property of the class `dmv` is composed of the class `dm` is established later when Containers for each class are defined. This concept has been explained in section 4.4.

The complete system can be represented by the class `smithpredictor` that has two properties `PLANT` and `CONTROLLER`. The property `PLANT` of type `dmv` describes the plant and the property `CONTROLLER` of type `smithcontroller` represents the Smith controller. Again, this relationship is not established at the point where the classes are created but is explicitly established later through property definition. The Smith controller class `smithcontroller` has three properties: `PLANTMODEL`, `PREDICTOR` and `PRED_CONTROLLER`. The property `PLANTMODEL` of type `dmv` represents the plant model. `PREDICTOR` points to the delay-free part of the plant model description. `PRED_CONTROLLER` describes the predictor controller, i.e. the controller of the rational part of the plant model. It is also of type `dmv`, but since it is a multivariable system without delays it has all its delay vectors set to zero.

Figure 5.5 shows how the different Containers are linked through their property definitions to establish a class-composition hierarchy.

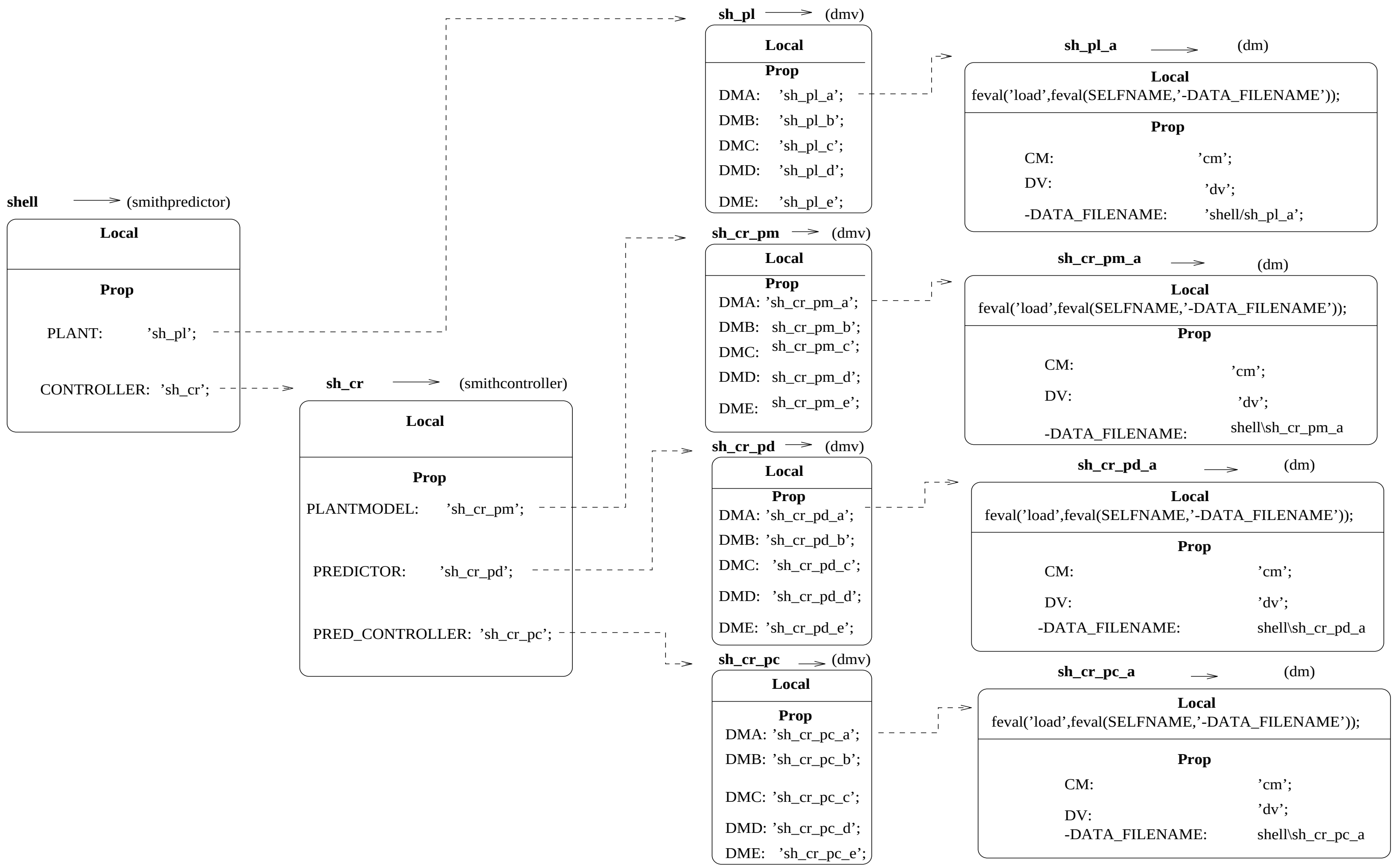

Figure 5.8: Class-composition hierarchy for system descriptions in multivariable Smith predictors

Once the Containers and their inter-relationships are established, meaningful queries can be issued. Some example queries on the Container `shell` of class `smithpredictor` are given here:

```
>>shpm_b_dv = shell(CONTROLLER(PLANTMODEL(DMB(DV))))

  shpm_b_dv =

           14     15     18     27     28
```

retrieves the delay vector of the B matrix of the plant model (a part of the controller) description and assigns it to the variable shpm_b_dv. Similarly,

```
>>shpm_b_cm = shell(CONTROLLER(PLANTMODEL(DMB(CM))))

  shpm_b_cm =

          (16,1)        0.1078
          (19,1)        0.1500
           (5,2)        0.9533
          (26,2)        0.0295
          (12,3)        0.1725
          (21,3)        0.1176
```

retrieves the matrix of coefficients of the B matrix of the plant model description. The above matrix is stored and displayed as a sparse matrix. The command,

```
>>shpm_b_cm = full(shell(CONTROLLER(PLANTMODEL(DMB(CM)))));
```

assigns the full matrix to the variable shpm_b_cm.

```
>>[c1,c2,c3,c4]=shell([CONTROLLER(PLANTMODEL(DMC(CM)),...
  PRED_CONTROLLER(DMC(1)),PREDICTOR(DMC(CM))]),PLANT(DMC(CM))]);
```

retrieves the matrix of coefficients of the C matrix from the system descriptions of plant model, predictor controller, predictor and plant.

```
>>[pred_cm,pred_dv]=shell(CONTROLLER(PREDICTOR(DMD)));
```

assigns the matrix of coefficients and the delay vector of the D matrix of the predictor, which is a part of the controller, to the variables pred_cm and pred_dv. The above query can also be formulated as:

```
>>[pred_cm,pred_dv]=shell(CONTROLLER(PREDICTOR(DMD([CM,DV]))));
```

The first four matrices associated with the plant model can be retrieved through the following command:

```
>>[sh_sm_pm_a_cm,sh_sm_pm_a_dv,sh_sm_pm_b_cm,sh_sm_pm_b_dv]=
  shell(CONTROLLER(PLANTMODEL(1:4)));
```

The above query can also be formulated as:

```
>>[shpm_a_cm,shpm_a_dv,shpm_b_cm,shpm_b_dv]= ...
  shell(CONTROLLER(PLANTMODEL([DMA(CM), DMA(DV), DMB(CM), DMB(DV)])));
```

This approach to the organization of system descriptions is similar to some extent to the TREE data structure in the Robust Control Toolbox for Matlab (see section 4.4). Both offer a way of hierarchical data organization, the data specific to different entities can be retrieved using property/entity names, and data specific to a project can be grouped together under one name.

5.6 Summary

In this chapter, we have shown the different ways in which Matlab Containers can be used in connection with fairly complex control analysis and design tasks. We have described their use for design and analysis of multivariable Smith predictors, both with and without computational chains, and for defining class-composition hierarchies for representing different system descriptions of a multivariable Smith predictor.

The case study shows that Containers provide a general solution for organizing data and 'computational sessions' based on how users 'think' about a problem. Containers offer users a considerable amount of flexibility in defining their 'own' Containers and structuring computational chains at any level of granularity depending on the intended purpose. In the course of the case study, object-oriented style of interaction, persistence of both user actions and 'contents' of Containers and 'computational efficiency' (determining exactly which Containers need to be recomputed) were found to be particularly useful features. Moreover, since all constructs are expressed in the native language (in this case Matlab) and are consistent with the programming environment, they help rather than distract (which normally happens if external support facilities are used) control engineers from their main task of designing and analyzing control systems.

Chapter 6

Conclusions

In this chapter, the conclusions drawn from the dissertation and some recommendations for future research are presented.

An historical survey of non-numerical support areas in CACE environments reported in this thesis shows that these environments have come a long way with regard to programming style and user interaction, but they offer poor support for database management. Most research efforts towards better data management facilities have concentrated on some kind of external tool. This not only results in technical problems as was evident in the GE-MEAD project, but an external tool for data management is also impractical from a user's perspective. It distracts them from their main concern which is numerical-oriented control systems analysis and design. An important conclusion drawn from this is that a data management facility should not be provided as an external tool; rather it should be an integral part of the analysis and design process. Furthermore, if such a facility is incorporated into the numerical programming environment, it should closely correspond on a syntactical level to the other programming constructs in the environment. This is essentially similar to the concept of 'cognateness' in natural languages.

The complexity of the data management problem increases considerably when the organization of a whole 'project' and the different versions of models and results generated during a project are taken into account. Organization of the workspace during a control analysis and design session is an important aspect of this problem. Defining a *structure* for performing computations is a pre-requisite for project, version and workspace organization. The structure should define a way for: organization, retrieval and update of different data, algorithms, and parameter settings for various sessions; and maintenance of different versions of models and results in a project. However, it is important that any such structure should not restrict the programming process unnecessarily. Persistence of all user actions and general applicability of the solution are also important considerations in the course of designing such an environment.

Object-oriented methodologies provide a convenient means of programming with entities which have complex inter-relationships. They provide a high degree of abstraction which is particularly useful for an area such as control systems analysis and design where the computational tasks are in the hands of the user. Several CACE environments incorporate object-oriented features which enhance the programming process considerably. We strongly support research in this direction and believe that greater effort should be extended to the provision of object-oriented database management functionality.

Matlab Containers, a concept extended in this thesis, is used to investigate the possibility and the advantages of incorporating these features into the interactive software environments currently being used by the majority of control engineers. Containers incorporate the following important features discussed above:

- They facilitate the logical organization of data and enable the systematic creation and manipulation of data and functions.
- They allow all user actions to persist.
- They provide a general solution, i.e. their use is not limited to a specific application domain.
- They provide a degree of object-orientation which is made available to the user.
- All Container constructs are implemented in and are consistent with the Matlab host environment.

Computational chains based on Containers, as described in the thesis, are a good example of how a 'structure' for computational tasks can be defined without losing the advantages of current packages with regard to programming style and flexibility. Computational chains that we have implemented provide:

- a basic version management mechanism;
- an interactive environment for model updating, parameter setting and algorithm selection;
- a systematic approach to code reuse and documentation of user actions;
- a mechanism that 'remembers' that a computation has been performed and that its results are available for further use.

Future Work

There is still no definitive answer as to what an 'ideal' CACE environment should look like. We therefore suggest that several small scale research projects should be undertaken to investigate different aspects of CACE software before deciding what future direction should be taken. Some suggestions for possible future work on small scale research projects are given below:

- A scheme for a proper version management mechanism should be investigated. Computational chains as implemented in Containers provide a good foundation for exploring such a scheme. A proper version management mechanism should incorporate mechanisms for 'freezing' and 'branching' of versions. By 'freezing' we mean storing an entity (or a group of entities) as a version. The way a version branches can depend on several factors such as whether there is a small change or a major modification. For example, small changes in version 1.0 would result in version 1.1, 1.2 etc. Major modifications in version 1.0 would result in version 2.0. Storing the different entities to be versioned as abstract data types will be useful for maintaining a database of versions.
- Containers have proved to be a powerful tool and it would be worthwhile to implement them properly, paying more attention to efficiency. A more convenient user interface such as a Container Browser would also be useful. This would lead to implementation in *C* or a similar language. It remains to be seen what impact the new version of Matlab (Version 5.0) with several new features (in particular the feature of 'cell arrays') will have on the features as well as the implementation of Matlab Containers.
- The possibility of providing a facility similar to Matlab Containers in other Matlab-like matrix-based environments should be looked into. It would be easier to achieve some features of Containers in Xmath. To some extent the local context initialization corresponds to the Xmath partition concept. List objects in Xmath would be particularly helpful for passing references to evaluated versions of Containers.

Bibliography

[1] *Advanced Continuous Simulation Language (ACSL).* Mitchel and Gauthier Associates, Concord, Massachusetts, 1986.

[2] S. Ahmed, A. Wong, D. Sriram, and R. Logcher. A Comparison of Object-Oriented Database Management Systems for Engineering Applications. Technical Report R91-12, Massachusetts Institute of Technology, May 1991.

[3] A.A.L. Ambler, M.M. Burnett, and B.A. Zimmerman. Operational versus Definitional—A Perspective on Programming Paradigms. *IEEE Computer*, 25(9):28–43, September 1992.

[4] M. Andersson. Omola: An Object-Oriented Language for Model Representation. Master's thesis, Department of Automatic Control, Lund Institute of Technology, Lund, May 1990.

[5] M. Andersson. *OmSim and Omola Tutorial and User's Manual.* Department of Automatic Control, Lund Institute of Technology, Lund, 3rd edition, April 1993.

[6] M. Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, 1994.

[7] M. Andersson, S.E. Mattsson, and B. Nilsson. On the Architecture of CACE Environments. In *IFAC/IMACS Symposium on Computer-Aided Design in Control Systems, Swansea*, 1991.

[8] K.J. Åström. Computer-Aided Tools for Control Systems Design. In M. Jamshidi and C. J. Herget, editors, *Computer-Aided Control Systems Engineering.* Elsevier Science Publishers, Amsterdam, 1985.

[9] K.J. Åström and W. Kreutzer. System Representation. In *3rd IEEE Symposium on CACSD, Arlington, VA*, 1986.

[10] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *The First International Conference on Deductive and Object-Oriented Databases*, December 1989.

[11] G.J. Balas, J.C. Doyle, K. Glover, A. Packard, and R. Smith. *μ-Analysis and Synthesis Toolbox*. The MathWorks, Inc., 24 Prime Way, Natwick, MA 01760, 1994.

[12] H.A. Barker, M. Chen, P.W. Grant, C.P. Jobling, and P. Townsend. Open Architecture for Computer-Aided Control Engineering. *IEEE Control Systems Magazine*, April 1993.

[13] H.A. Barker, C.P. Jobling, M. Szymkat, and O. Ravn. A Requirements Analysis of Future Environments for Computer-Aided Control Engineering. In *Proc. 12th IFAC World Congress, Sydney*, 18-23 July 1993.

[14] J.M. Boyle, M.P. Ford, and J.M. Maciejowski. A Multivariable Frequency Domain Toolbox for Matlab. *IEEE Control Systems Magazine*, 9:59–65, 1989.

[15] J.M. Boyle and J.M. Maciejowski. Expert-Aided Sequential Design of Multivariable Systems. *IEE Proceedings*, 139(5), September 1992.

[16] P.T. Breuer and J.M. Maciejowski. Representing Control Systems in a Semantic Database Management System within a Functional Programming Environment. *Proc. IEE Conference, Oxford*, April 1988.

[17] D. Bünz and K. Gütschow. CATPAC—An Interactive Software Package for Control System Design. *Automatica*, 21(2):209–213, 1985.

[18] R.G.G. Cattell. *Object Data Management*. Addison-Wesley, 1991.

[19] F.E. Cellier. *Continuous System Modeling*. Springer-Verlag Inc., New York, 1991.

[20] F.E. Cellier. Integrated Continuous-System Modeling and Simulation Environments. In D. A. Linkens, editor, *CAD for Control Systems*. Marcel Dekker Inc., 1993.

[21] R.Y. Chiang and M.G. Safonov. *Robust Control Toolbox for Matlab*. The MathWorks, Inc., 24 Prime Way, Natwick, MA. 01760, 1994.

[22] *CTRL-C User's Guide, Version 4.0*. Systems Control Technology Inc., 1801 Page Mill Road, Palo Alto, CA 94304, September 1986.

[23] M.A. Curth and H. Edelmann. *APL—A Problem-oriented Introduction*. Ellis Horwood Limited, 1989.

[24] D.A.Linkens. An artificial intelligence approach to environments for modeling and simulation. In D.A. Linkens, editor, *CAD for Control Systems*. Marcel Dekker, Inc., 1993.

[25] C.J. Date. *An Introduction to Database Systems, Volume 1*. Addison-Wesley, 5th edition, 1990.

[26] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[27] H. Elmqvist. *A Structured Model language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, 1978.

[28] M. Englehart and M. Jackson. ControlH: An Algorithm Specification Language and Code Generator. *IEEE Control Systems Magazine*, 15(2):54–64, April 1995.

[29] R. Finsterwalder and G. Grübel. Visual Programming in ANDECS for Controlled Systems Design Experimenting. In S.E. Mattsson, J.O. Gray, and F.E. Cellier, editors, *IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, Tucson, AZ*, 1994.

[30] F. Gallo, R. Minot, I. Thomas, et al. The Object Management System of PCTE as a Software Engineering Database Management System. *ACM SIGPLAN Notices*, 22(1), January 1987.

[31] S. Gani and P. Picuri. The Object Revolution: How COM Technology is Changing the Way We Do Business. *IEE Computing & Control Engineering Journal*, 6(3), June 1995.

[32] B.S. Garbow, J.M. Boyle, J.J. Dongarra, and C.B. Moler. *Matrix Eigensystem Routines – Eispack Guide Extension*. Springer-Verlag, Berlin, 1977.

[33] D. Garlan, G.E. Kaiser, and D. Notkin. Using Tool Abstraction to Compose Systems. *IEEE Computer*, June 1992.

[34] G. Grübel. ANDECS—The Way Forward in Computer-Integrated Control Engineering. In *IEE Colloquium on 'Open Systems: The Way Forward in Computer-Aided Control Engineering'*, 15 December 1992.

[35] G. Grübel. ANDECS and CACE 'Framework Reference Models'. *OPEN-CACSD: Newsletter of the IFAC/IEEE-CSS Working Group on Guidelines for Open CACSD Software, Swansea*, 1(11), February 1993.

[36] G. Grübel. The ANDECS-CACE Framework A_RSYST for Integrated Analysis and Design of Controlled Systems. In *Proc. IEEE/IFAC Joint Symposium on Computer-Aided Control Systems Design, Tucson, AZ*, 1994.

[37] G. Grübel. The ANDECS CACE Framework. *IEEE Control Systems*, 15(2), April 1995.

[38] P. Van Den Hamer and K. Lepoeter. Managing Design Data: The Five Dimensions of CAD Frameworks, Configuration Management and Product Data Management. *Proceedings of the IEEE*, 84(1):42–56, January 1996.

[39] K.A. Harbisson-Briggs, H.D. Joos, and J.M. Maciejowski. EXPRESS Data Definitions for Control Engineering. In *Proc. 12th IFAC World Congress, Sydney*, 18-23 July 1993.

[40] S. Hope. Information Repository Services. In *IEE Colloquium on 'Open Systems: The Way Forward in Computer-Aided Control Engineering'*, 15 December 1992.

[41] S. Hope, P.J. Fleming, and J.G. Wolff. Object-Oriented Database Support for Computer-Aided Control System Design . In H. A. Barker, editor, *IFAC/IMACS Symposium on Computer-Aided Design in Control Systems, Swansea*, pages 200–204. IFAC, Pergammon Press, July 1991.

[42] *Intoduction to Ψlab, V 2.1.1.* Institut National de Recherche en Informatique et Automatique, Le Chesnay Cedex, INRIA, France, 1995.

[43] *ISID Module for Xmath.* Integrated Systems Inc., 3260 Jay Street, Santa Clara, CA 95054, 1994.

[44] M. Jamshidi and C.J. Herget, editors. *Computer-Aided Control Systems Engineering.* North-Holland Elsevier Science Publishers, Amsterdam, 1985.

[45] H.D. Joos and M. Otter. Control Engineering Data Structures for Concurrent Engineering. In *IFAC/IMACS Symposium on Computer-Aided Design in Control Systems, Swansea*, pages 107–112, July 1991.

[46] R.H. Katz and T.J. Lehman. Database Support for Versions and Alternatives of Large Design Files. *IEEE Transactions on Software Engineering*, 10(2), March 1984.

[47] W. Kent. Panel: An Overview of the Versioning Problem. *ACM SIGMOD Record*, 18(2), November 1989.

[48] W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[49] P. Kolb, C.A. Ganz, and M. Rickli. An Object-Oriented Kernel for Control Systems Engineering. In *12th IFAC World Congress, Sydney*, volume 6, pages 443–446, 1993.

[50] A. Kröner, P. Holl, W. Marquardt, and E.D. Gilles. DIVA—An Open Architecture for Dynamic Simulation. *Computers & Chemical Engineering*, 14(11):1289–1295, 1990.

[51] J.N. Little, A. Emami-Naeini, and S.N. Bangert. CTRL-C and Matrix Environments for the Computer-Aided Design of Control Systems. In M. Jamshidi and C. J. Herget, editors, *Computer-Aided Control Systems Engineering.* Elsevier Science Publishers, Amsterdam, 1985.

[52] L. Ljung and T. Glad. *Modeling of Dynamic Systems.* Prentice Hall Information and System Sciences Series, 1994.

[53] A.G.J. MacFarlane, G. Grübel, and J. Ackermann. Future Design Environments for Control Engineering. *Automatica*, pages 165–176, 1989.

[54] J.M. Maciejowski. A Database Approach to Computer-Aided Control System Design. In *Proceedings of the American Control Conference, San Francisco*, June 1983.

[55] J.M. Maciejowski. A Core Data Model for Computer-Aided Control Engineering. Technical Report CUED/F-CAMS/TR.257, Cambridge University Engineering Department, March 1985.

[56] J.M. Maciejowski. Data Structures and Software Tools for the Computer-Aided Design of Control Systems : A Survey. *Proc. IFAC Symposium on CADCS, Beijing*, August 1988.

[57] J.M. Maciejowski. *Multivariable Feedback Design.* Addison-Wesley Publishers, 1989.

[58] J.M. Maciejowski. Software standards in the control systems community. In *Symposium on Computer-Aided Control Systems Design, Napa*, 1992.

[59] J.M. Maciejowski. The Role of EXPRESS in the CACE Environment. In *IEE Colloquium on 'Open Systems: The Way Forward in Computer-Aided Control Engineering'*, 15 December 1992.

[60] J.M. Maciejowski. Robustness of Multivariable Smith Predictors. *Journal of Process Control*, 4(1):29–32, 1994.

[61] J.M. Maciejowski. CACSD, GUIs and the Multivariable Frequency Domain Toolbox. In *IEE Colloquium on Advances in Computer-Aided Control System Design*, March 1996.

[62] J.M. Maciejowski and A.G.J. MacFarlane. CLADP: The Cambridge Linear Analysis and Design Programs. *Control Systems Magazine*, December 1982.

[63] J.M. Maciejowski and M. Szymkat. Containers—A Step Towards Objects with Matlab. In S.E. Mattsson, J.O. Gray, and F.E. Cellier, editors, *IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, Tucson, AZ*, 1994.

[64] J.M. Maciejowski and C.Y. Tan. DB-Prolog Environments for Computer-Aided Control Engineering. In D. A. Linkens, editor, *CAD for Control Systems.* Marcel Dekker Inc., April 1992.

[65] J.M. Maciejowski and J.H. Taylor. A Report on the 1993 IFAC World Congress Standards and Guidelines Session. In *Proc. IEEE/IFAC Joint Symposium on Computer-Aided Control Systems Design, Tucson, AZ*, March 1994.

[66] H. Mason. STEP: Overview and Fundamental Principles. Technical report, ISO TC184/SC4/WGPMAG, 1991.

[67] *MATLAB User Guide.* The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760, 1994.

[68] *$Matrix_x$ Products Family.* Integrated Systems Inc., 3260 Jay Street, Santa Clara, CA 95054, 1994.

[69] S.E. Mattsson and M. Andersson. Omola—An Object-oriented Modeling Language. In M. Jamshidi and C.J. Herget, editors, *Recent Advances in Computer-Aided Control Systems Engineering.* Elsevier Science Publishers, 1993.

[70] S.E. Mattsson, M. Andersson, and K.J. Åström. Object-Oriented Modeling and Simulation. In D.A. Linkens, editor, *CAD for Control Systems.* Marcel Dekker, Inc., 1993.

[71] A.D. McGettrick. *ALGOL 68—A First and Second Course.* Cambridge University Press, Cambridge, 1978.

[72] N. Munro. ECSTASY—A Control System CAD Environment. In *Proc. 11th IFAC World Congress on Automatic Control, Tallinn, Estonia*, 13-17 August 1990.

[73] N. Munro and C.P. Jobling. ECSTASY: A Control System CAD Environment. In D.A. Linkens, editor, *CAD for Control Systems.* Marcel Dekker, Inc., 1993.

[74] L.W. Nagel. SPICE2: A Computer Program to Simulate Semiconductor Circuits. Technical Report ERL-M520, Electronics Research Laboratory, University of California, Berkeley, 1975.

[75] *The Nonlinear Control Design Toolbox.* The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760, 1994.

[76] B.A. Oggunaike and W.H. Ray. Multivariable controller design for linear systems having multiple time delays. *AIChEJ*, 25:1043–1057, 1979.

[77] P. Van Overschee. *Subspace Identification.* PhD thesis, Katholieke Universiteit Leeuven, Belgium, 1994.

[78] C.C. Pantelides. SpeedUp—Recent Advances in Process Simulation. *Computers & Chemical Engineering*, 12(7):745–755, 1988.

[79] P. Phaal. *An Object Oriented Environment for Control System Design.* PhD thesis, Cambridge University Engineering Department, 1987.

[80] R.S. Pressman. *Software Engineering—A Practitioner's Approach.* McGraw-Hill, 1992.

[81] D.M. Prett and M. Morari. The Shell Process Control Workshop, Boston, 1987.

[82] A. Ralston and E.D. Reilly, editors. *Encyclopedia of Computer Science*. Chapman & Hall, London, 1993.

[83] O. Ravn and M. Szymkat. The Evolution of CACSD Tools—A Software Engineering Perspective. In *IFAC/IMACS Symposium on Computer-Aided Design in Control Systems, Swansea*, 1991.

[84] O. Ravn and M. Szymkat. Requirements for user interaction support in future cace environments. In *Proc. IEEE/IFAC Joint Symposium on Computer-Aided Control Systems Design, Tucson, AZ*, March 1994.

[85] C.M. Rimvall. *Man-machine Interfaces and Implementation Issues in CACSD*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 1986.

[86] C.M. Rimvall and M. Wette. Towards Standards for CACE Command Syntax and Graphical Interfaces. In *Proc. 12th IFAC World Congress, Sydney*, 18-23 July 1993.

[87] M. Rimvall and F.E. Cellier. A Structural Approach to CACSD. In M. Jamshidi and C. J. Herget, editors, *Computer-Aided Control Systems Engineering*. Elsevier Science Publishers, Amsterdam, 1985.

[88] H.H. Rosenbrock. On the Design of Linear Multivariable Control Systems. In *IFAC, Session 1, Paper 1A*, 1966.

[89] A.B. Saifuddin, J.M. Maciejowski, and M. Szymkat. Analysis of Multivariable Smith Predictors Using Matlab Containers. In *IEE Colloquium on Applied Control Techniques using Matlab, Digest No: 1995/014, London*, January 1995.

[90] A.B. Saifuddin, M. Szymkat, and J.M. Maciejowski. Computational Chains for CACSD Using Matlab Containers. *To appear in the Proceedings of IFAC/IEEE Symposium on Computer-aided Control Systems Design, Michigan*, September 1996.

[91] A.S.M. Sajeev and A.J. Hurst. Programming Persistence in χ. *IEEE Computer*, 25(9):57–66, September 1992.

[92] D. Schenck. Information Modelling Language EXPRESS. Technical report, ISO TC184/SC4/WG1, 1988.

[93] *The System Identification Toolbox*. The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760, 1995.

[94] *Simulink User's Guide*. The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760, 1994.

[95] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. klema, and C.B. Moler. *Matrix Eigensystem Routines—Eispack Guide*. Springer-Verlag, Berlin, 1976.

[96] H.A. Spang. The Federated Computer-Aided Control Design System. In M. Jamshidi and C. J. Herget, editors, *Computer-Aided Control Systems Engineering.* Elsevier Science Publishers, Amsterdam, 1985.

[97] J.C. Strauss. The SCi Continuous System Simulation Language (CSSL). *Simulation*, 9:281–303, 1967.

[98] *SystemBuild.* Integrated Systems Inc., 3260 Jay Street, Santa Clara, CA 95054, 1994.

[99] M. Szymkat and J.M. Maciejowski. Time Delay Toolbox for Matlab. In S.E. Mattsson, J.O. Gray, and F.E. Cellier, editors, *Proceedings, IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, Tucson, AZ*, 1994.

[100] M. Szymkat, J.M. Maciejowski, and A.B. Saifuddin. Matlab Containers for Control Engineering: Concepts and Applications. *To be submitted to IEEE Transactions on Control Systems Technology*, 1996.

[101] C.Y. Tan. A Logic Programming based CACSD Environment. Technical Report CUED/F-INFENG/TR46, Cambridge University Engineering Department, 1990.

[102] C.Y. Tan and J.M. Maciejowski. DB-Prolog: A Database Programming Environment for Computer Aided Control Engineering. In *Proceedings of the Workshop in Expert Systems, AI and Software Engineering Application, in association with the International Logic Programming Symposium, San Diego, California, USA*, 28-31 October 1991.

[103] J.H. Taylor, D.K. Frederick, C.M. Rimvall, and H.A. Sutherland. Computer-Aided Control Engineering Environments: Architecture, User Interface, Database Management, and Expert Aiding. In *Proc. 11th IFAC World Congress on Automatic Control, Tallinn, Estonia*, 13-17 August 1990.

[104] J.H. Taylor and G. Grübel. Database Management Guidlines for Computer-Aided Control Engineering. In *Proc. 12th IFAC World Congress, Sydney*, 18-23 July 1993.

[105] J.H. Taylor, K.H. Nieh, and P.A. Mroz. A Database Management Scheme for Computer-Aided Control Engineering. In *American Control Conference, Atlanta*, June 1988.

[106] J.H. Taylor, M. Rimvall, and H.A. Sutherland. Future Developments in Modern Environments for CADCS. In *IEEE Symposium on Computer-Aided Control Systems Design, Napa*, 17-19 March 1992.

[107] J.H. Taylor, M. Rimvall, and H.A. Sutherland. Computer-Aided Control Engineering Environments. In D.A. Linkens, editor, *CAD for Control Systems.* Marcel Dekker, Inc., 1993.

[108] R. Walker, C. Gregory Jr., and S. Shah. Matrix-X, A Data Analysis, System Identification, Control Design and Simulation Package. *IEEE Control Systems Magazine*, 2(4):30–37, 1982.

[109] M. Wette. Casey: A Computer-Aided Engineering System. In *IEEE Symposium on Computer-Aided Control Systems Design, Napa*, 17-19 March 1992.

[110] *Xmath Basics.* Integrated Systems Inc., 3260 Jay Street, Santa Clara, CA 95054, 1994.

[111] S.B. Zdonik and D. Maier (editors). *Readings in Object-oriented Database Systems.* Morgan Kaufmann Publishers Inc., 1990.