

VISUAL ASPECTS OF COMPUTER AIDED CONTROL SYSTEMS DESIGN

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Christopher Andreas Ganz

Dipl. El.-Ing. ETH
born April 25, 1963
citizen of Meggen and Luzern,
Switzerland

accepted on the recommendation of
Prof. Dr. W. Schauffelberger, examiner
Prof. Dr. J. Gutknecht, co-examiner



Richard Strauss

Acknowledgments

In the round of people who helped me in my pursuit for this thesis, I first want to express my gratitude to Prof. Schaufelberger for his support during this work. I also want to thank Prof. Gutknecht for his willingness to co-referee this thesis. I also feel much obliged to Prof. Mansour, who gave me the opportunity to work at the Automatic Control Lab.

Another word of thank goes to Dr. Maier of ABB for the organization of the funds which financed parts of this thesis. He also introduced me to the field of software engineering, another reason why he appears on this page.

I would also like to include my colleagues at the Automatic Control Laboratory in my thanks for their continuous support. Peter Kolb deserves special mention. He is the co-author of the Leporello package presented in the second part of this thesis, I am very much indebted to him for his work. He and Martin Rickli also contributed to the fruitful discussions in the beginning of this project. Not always as fruitful, but mostly entertaining were the discussions with Janusz Milek, Markus Kottmann and Christoph Eck.

This page would not be complete without the mention of many personal friends, who always did a good job in distracting my mind from the problems of this thesis. I would especially like to mention my friends of Byfv Cquoxx (sic!), Beat and Barbara Huber and Denise Graf.

Finally, I want to thank my parents for all their support during my long education. It is to them, this thesis is dedicated.

Leer - Vide - Empty

Contents

Abstract	1
Kurzfassung	2
1. Introduction	5
1.1. Pictures	5
1.2. Visual Engineering	7
1.3. Graphics in Control Engineering	8
1.4. Structured Visual Control Systems Design	10
1.5. Scope, Contributions and Organization	11
1.5.1. Scope	11
1.5.2. Contributions	12
1.5.3. Organization	12
2. Visual Aspects of Software in Automatic Control	15
2.1. Scientific Visualization	16
2.1.1. One-Dimensional Displays	17
2.1.2. Two-Dimensional Plots	17
2.1.3. Three-Dimensional Plots	19
2.1.4. The Use of the Fourth Dimension	20
2.1.5. Summary	22
2.2. Visual Programming	23
2.2.1. Control Flow Descriptions	26
2.2.2. Data Flow Descriptions	30
2.2.3. Summary	34
2.3. Tools	35
2.3.1. SystemSpecs	35
2.3.2. Simulink	38
2.3.3. BlockSim	40
2.3.4. GenT	42
2.3.5. LabVIEW	45
2.4. Text or Pictures?	48
2.4.1. Visual Languages Discussion	51
2.4.2. Summary	53
3. Leporello: Visual Representation of the Design Process	55
3.1. The Leporello Project	56
3.2. The Design Cycle	58
3.2.1. Action Tree Basics	58

3.2.2. Tree-structured Data Bases	60
3.2.3. Graph Theory	61
3.3. Basic Data Structures in Leporello	62
3.3.1. The Object Oriented Approach	63
3.3.2. Control Data Objects	63
3.3.3. Parameter Lists	66
3.3.4. Algorithms	67
3.3.5. Algorithm Execution	69
4. The Action Tree: Functionality and Object Definition	71
4.1. Algorithm Node Execution	72
4.1.1. Selecting and Executing an Algorithm	72
4.1.2. Execution of an Algorithm on Different CDC Nodes	78
4.1.3. Creation of Multi-object Nodes	81
4.1.4. Application of an Algorithm to a Multi-object Node	83
4.2. Managing Multi-object Nodes	90
4.2.1. Extract a Subset Node from a Multi-object Node	90
4.2.2. Merge Subset Nodes	92
4.3. Managing Tree Complexity	93
4.3.1. Copy and Paste Operations	93
4.3.2. Delete Nodes	95
4.3.3. Mark Deleted Subtree	95
4.4. Automatic Subtree Execution	96
4.4.1. Make Subtree Consistent	97
4.4.2. Hierarchical Algorithm	99
4.5. Special Tasks	104
4.5.1. Parameter Sweep	105
4.5.2. Parameter Optimization	105
4.6. Hierarchical Systems	107
4.6.1. Hierarchical Systems Class Definition	107
4.6.2. Block Diagram Editor and Action Tree Operations	107
5. Application	111
5.1. Plant and Hardware Description	112
5.1.1. Electronic Laboratory Balance	112
5.1.2. Computer Hardware	113
5.2. Working with Leporello	114
5.2.1. Leporello User Interface	114
5.2.2. Balance Measurements and Interactive Controller Design	116
5.2.3. Identification of a Continuous-time Model	121
5.2.4. State Feedback Controller Design	123
5.2.5. Discrete-time ARX-model Identification	125
5.2.6. Remarks	130

6. Conclusions and Further Work	131
6.1. Conclusions	131
6.1.1. Scientific Visualization	131
6.1.2. Visual Languages	132
6.1.3. Visual Control Systems Design in Leporello	132
6.2. Further Work	133
6.2.1. Leporello Concept Extensions	133
6.2.2. Future Leporello Development	134
Appendix	
A. List of available Leporello algorithms	135
B. Action tree node states and pictures	139
C. Object syntax and naming conventions	141
D. Managing Dynamical Data Structures	143
D.1. Variable Size Data Structures	143
D.2. Garbage collection	145
Glossary	147
Bibliography	151
Software references	154
Index	155

Abstract

The use of a computer to solve control engineering problems is state of the art in today's automatic control community. Numerous software packages have been developed in recent years. A few of them are commercially available today and some are even regarded as a de-facto standard in computer aided control systems design (CACSD).

The advantage of CACSD software is the large number and the quality of the provided algorithms. However, developments concentrated on numerical rather than on user interface aspects. Until very recently, most of these packages required textually entered commands and provided rudimentary two-dimensional line graphs.

Recent developments in computer science have removed the limitations of older systems. Data and software visualization are applied to various fields of science and engineering, and most computers can be operated using a graphical user interface.

In this thesis, we explore the application of graphical methods in computer aided control systems design. With increasing emphasis we will discuss the following aspects of visual interfaces:

- Data visualization: Data resulting from either experiments or algorithm execution (*i.e.* computer simulations) are mostly displayed in two dimensional graphs. The use of more complex, higher dimensional displays is discussed. Examples of three dimensional displays, moving data visualization and interactive manipulation of algorithms at runtime are presented.
- Visual programming: The technique to show the interaction of sub-systems in block diagrams is well known in control engineering. Their similarity to visual programming languages helps us to not only display data visually, but also to design computer programs or controller implementations using visual methods. Available visual programming languages are presented in control applications.
- Graphical user interface to the control systems design cycle: As we have seen above, the emphasis of today's CACSD packages is in the availability of sophisticated algorithms. Little or no emphasis has been put on the support of the design cycle as a whole. With the 'Leporello' package we present a new software tool, which visualizes all the work done during a design session. All algorithms which were used to reach a successful controller implementation are stored and displayed in the 'action tree', together with their parameters and results. Intermediate data items are collected consistently in control data objects. The possibility to use external packages for the implementation of algorithms allows a fully graphical user interface for control systems engineering using the whole range of algorithms provided by other applications.

The methods and tools presented are used to solve one of the example plants from the student laboratory. By using the 'Leporello' tool we demonstrate its use and range of application.

Kurzfassung

Die Verwendung von Computern zur Lösung von regelungstechnischen Problemen ist heutzutage Stand der Technik. Mehrere Softwarepakete wurden in letzter Zeit entwickelt. Einige von ihnen sind heute kommerziell erhältlich, und davon haben sich einige zu einem faktischen Standard im Computer-unterstützten Reglerentwurf entwickelt.

Die Vorteile der heute verfügbaren Programmpakete liegen hauptsächlich in der grossen Vielfalt und Qualität der unterstützten Algorithmen. Bei deren Entwicklung wurde jedoch dem Entwurf der Benutzerschnittstelle wenig Beachtung geschenkt. Die Möglichkeiten dieser Pakete erschöpften sich bis vor kurzem in alphanumerischen Befehlseingaben und zweidimensionalen Liniengraphen.

Neuste Entwicklungen im Bereich der Computertechnik haben die technischen Einschränkungen früherer Systeme beseitigt. Daten- und Softwarevisualisierung hat in vielen Gebieten Einzug gehalten, die graphische Benutzerschnittstelle hat sich zur Bedienung von Computern durchgesetzt.

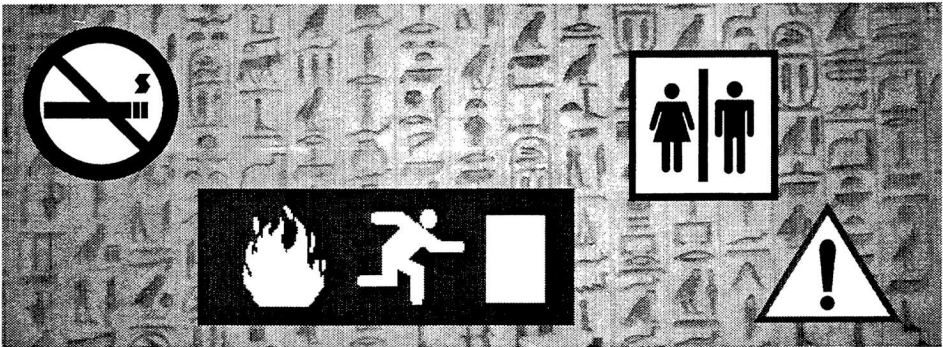
In dieser Arbeit werden die Möglichkeiten der Anwendung graphischer Methoden im Bereich des rechnerunterstützten Reglerentwurfes untersucht. Mit zunehmender Gewichtung werden die folgenden Bereiche diskutiert:

- Datenvisualisierung: Messwerte oder Resultate von Algorithmen (z.B. Simulationsresultate) werden heute meist in zweidimensionalen Graphen dargestellt. Durch die Verwendung höherer Dimensionen kann jedoch anfallenden Datensätzen mehr Information entnommen werden. Dreidimensionale oder animierte Darstellungen sowie die interaktive Beeinflussung von Algorithmen zur Laufzeit werden anhand von Beispielen gezeigt.
- Graphische Programmierung: Die Verwendung von Blockdiagrammen zur Darstellung der Relationen zwischen Teilsystemen ist eine dem Regelungstechniker vertraute Methode. Die Ähnlichkeit dieser Diagramme zu graphischen Programmiersprachen legt die Anwendung solcher Sprachen zum Entwurf von regelungstechnischen Algorithmen und Reglerimplementationen nahe. Verfügbare Umgebungen zur graphischen Programmierung werden jeweils mit Beispielen aus der Regelungstechnik vorgestellt.
- Graphische Benutzerschnittstelle für einen integralen Entwurfsprozess: Wie weiter oben gezeigt, liegt das Schwergewicht bei den heute erhältlichen Software-Paketen auf der Implementation ausgereifter Algorithmen. Wenig oder gar keine Beachtung wird dabei der Unterstützung des gesamten Entwurfszyklus geschenkt. Das in dieser Arbeit vorgestellte Programmpaket 'Leporello' visualisiert diesen Entwurfsprozess. Sämtliche zur Lösung eines Problems verwendeten Algorithmen werden zusammen mit den Algorithmusparametern und den resultierenden Daten gespeichert und im 'Action Tree' (Arbeitsbaum) dargestellt. Anfallende Daten werden jeweils in regelungstechnischen Datenobjekten zusammengefasst und konsis-

tent verwaltet. Die Möglichkeit, von 'Leporello' auf externe Programmpakete zuzugreifen, verwirklicht eine rein graphische Benutzeroberfläche zur Lösung regelungstechnischer Probleme unter Verwendung der Algorithmenvielfalt der externen Pakete.

Anhand eines einfachen Beispiels aus der Regelungstechnik werden die vorgestellten Methoden demonstriert und die Einsatzmöglichkeiten von 'Leporello' anschaulich dargestellt.

Leer - Vide - Empty



1.1. Pictures

The world today is influenced by pictures. Wherever attention needs to be drawn to, images make the deepest impressions in the quickest way. Graphics are used to visualize complex relationships, and movies are made from every book worth it. Even the works of Swiss author Friedrich Dürrenmatt are published in comic strip form.

This is basically due to the fact, that real-time image processing is one of the great strengths of the human brain. We are able to recognize a familiar face under difficult conditions, a task that machine vision is still unable to do.

On the other hand, reading text requires additional mental activities: the characters displayed on the 'text picture' need to be recognized and interpreted in words and sentences. The language read then has to be transformed into imagination, very often we there and then create pictures and scenes from text. A situation which illustrates this effect is somebody watching a movie after reading the book. The scenery previously imagined is replaced by the impressions of the pictures on screen. We then real-

ize that the text left a large range of possible interpretations, whereas the movie scenery shows exactly one scene from that range, leaving no uncertain areas to the imagination of the viewer. Many people do not like their fantasy being destroyed by pictures, in technical specifications however we want to transmit ideas and concepts as clearly as possible without leaving any part of it to the imagination of the reader.

Another situation where a diagram is clearly superior to a textual description is a map (like the one in Fig. 1.1). Imagine the situation when you have to give directions to somebody looking for a specific place. Your textual instructions reflect your personal view of the route, and only your route. A detour on the way or a missed intersection leaves the reader without help. The diagram of the city (commonly known as a map) however does not force the viewer to look at the route the way you describe it. She may choose whatever angle she likes to look at the map and is free to interpret the signs on the paper to her own liking. She is free to count intersections or go by route number signs.

This shows one great advantage of pictures: We access them randomly, *i.e.* the eyes sweep them freely and fix important or marked points instantly. Access to text however is sequential. If we start reading in the middle of the page we hardly ever get the context of the text. By choosing special fonts or pagination for titles or highlighted words, we are nevertheless able to draw the reader's attention to the points we want, but by doing this, we force him to switch to 'pictorial mode'. His eyes do not capture



Figure 1.1: Two-dimensional graphical display of landscape data, *i.e.*, a map.

Reproduced with permission of the Swiss Federal Office of Topography of 1.Nov.1994

the important words, but the area of the page which is darker than the rest and surrounded by a larger border of white. He then focuses on this point and reads the key-words indicated there.

Wherever a message has to be taken up quickly, *i.e.*, a warning or an alarm, we use pictorial shapes instead of words. Traffic signs, for example, transmit their message without text being read¹.

If messages have to be read by people speaking different languages, pictures are also an appropriate means to display information. Directions at airports are mostly indicated by icon-type signs which are similar all over the world. This allows us to quickly find the departure hall in Kuala Lumpur or the toilets in Reykjavík.

These signs mostly contain a simplified picture which is associated with the place they are directing us to. A piece of luggage directs us to the baggage claim, where we hopefully find objects of the type displayed, the sign with a bus leads us to a place where busses stop. In some cases however, we would like to explain some abstract function using a picture. Since there is no physical object to be displayed, we have to display an object which is associated with the abstract situation we would like to describe. If a sign shows a man running towards a door with flames behind him, we interpret the flames as 'emergency' and the man running toward the door as 'exit'. Without actually showing it, the displayed concept inherits some properties or functionality of the object used as its metaphor. If the cursor in a graphical user interface of a computer turns into a hand, we automatically assume that we can now touch something on screen (buttons, etc.) or move things around. Well chosen graphical representation of otherwise abstract concepts can so transmit a complex behavior in a very compact form which is easily understood.

The strong visual orientation of the human mind allows us to conclude that people retrieve information from pictures much faster than from written text. Graphical relations, easily recognized shapes and objects are identified at a much higher rate than from a textual description. In the following, we will see whether the use of pictures and graphical representations in control engineering helps designing and understanding software used for control systems design.

1.2. Visual Engineering

Since the appearance of the graphical user interface on computers, the importance of graphics in computer science has been steadily increasing. Workstations present windows with high resolution color images, and even movies are being seen on computer screens. Publishing has moved from the typesetter to the desktop and with this move has changed the whole printing business. To design, manipulate and improve pictures

1. This applies to European signs, most of the American signs contain their message written in plain text

on screen is no problem with today's computer power. Computer-developed movies are being produced today, and even virtual actors are under development.

But the engineers and programmers who designed all these graphical applications did this mostly by writing pages and pages of text, programs written in a programming language. The tools they create are very rarely available to ease their own job. They are not the only ones. Computer tools in many other fields of engineering require typed text, although graphics mostly do play an important role in these domains.

Since the early days of engineering, graphics have been used to specify and analyze systems of all kinds. Leonardo da Vinci left a countless number of plans and drawings of his inventions [8]. While his textual descriptions on the drawings are hardly legible today, the sketches give an excellent insight into the principles of his machines. Even today it is much easier for a mechanical engineer to specify the details of his design in plans and drawings, rather than to describe them in words to the people in charge of the manufacturing. The electrical engineer draws elements and connections of a logic board; a textual list describing the interconnections does not show the board layout clearly enough. Even when designing or explaining concepts and ideas an engineer uses graphics, and many lunch time discussions in an R&D department end up with someone sketching out his ideas on a napkin.

1.3. Graphics in Control Engineering

A control engineer is no exception herein. He uses block diagrams to specify control systems, and his analysis is very often based on the graphical interpretations of curves and signal displays (Bode-, root locus-diagrams).

Since the computer entered engineering sciences, these working methods have changed more and more. Algorithms used in computer aided control systems engineering (CACSD) shifted from graphical interpretations of curves to numerically difficult matrix manipulations not feasible in the days of paper and pencil calculations. This development required control problems to be formulated by means of computer statements, mostly in Fortran. Frequently used algorithms were collected in libraries (IMSL, Eispack, Linpack, RASP [13], etc.). Later, these libraries were combined and given a user interface to create software packages closer to the needs of CACSD (Ctrl-C, Matlab [51]). Such packages are still among the most widely used. They allow interactive analysis and design of control systems, and their built-in support of matrix manipulations and large libraries of well-tested algorithms ease control engineering very much. Still, even in those packages, programming of new algorithms or data manipulations require textual input of command lines, and graphical data analysis is mostly limited to a simple two-dimensional plot window. Graphical design of control systems has only lately been introduced by simulation packages equipped with a block diagram editor. The 3D or color capabilities are slowly being added, but when it comes to the implementation of either algorithms or real-time control systems, most

control engineers become the programmers of the old days and start writing computer software, where 'writing' is still using textual commands of a programming language.

One reason for this is found in the history of software engineering. The user interface of the early computers were hexadecimal LED displays with a keyboard which included just a few keys more than were necessary to type hexadecimal numbers. After a period of time where punched cards or teletype terminals were the main means of computer input, the CRT brought the possibility to see a whole page of text at a glance. Software engineering then evolved more and more towards developing programming languages, which could still be entered on an alphanumeric terminal. The idea of operating a computer by entering commands on a keyboard was established and is still accepted even in so-called graphical user interfaces which mainly consist of a window displaying the commands entered on that same keyboard.

This is not necessarily so. Already in 1977, Smith presented in his Ph.D. thesis a prototype of a visual language system (PYGMALION, [33]) which allowed software engineering by drawing interconnected boxes on screen by means of a graphics input device (mouse). Many systems and approaches have been presented since then (FUGA [4], Balsa [6], Pict [12], and a survey of several approaches in [1] and [7]), but none is rivalling any of the popular textual programming languages.

With the improving graphics capabilities of today's computer systems and the systems to come, the computer aided use of visual methods becomes more and more feasible. Using graphics and diagrams instead of text to visualize complex ideas and relationships is an attitude common to most engineers. There have been developments in software engineering which try to display the mostly complex behavior of a computer program in diagrams. Different aspects are visualized depending on the viewpoint of the programmer. These diagrams are then turned automatically into executable code, which justifies the term 'visual (graphical) programming languages'. Block diagrams of some sort are used very often to show either control or data relationships.

Direct code generation from computer aided design sketches or diagrams removes one source of errors: the translation of the drawings into code 'by hand'. Concept changes or improvements can be made directly in the design diagrams, the code is adapted automatically. The software design cycle is sped up significantly.

The control engineer's experience in using block diagrams suggests the use of this sort of visual programming languages when designing software in control engineering, but might also be a starting point for the development of new visual languages, based on methods used in automatic control.

1.4. Structured Visual Control Systems Design

In the course of this thesis we will illuminate another aspect of graphics in control engineering. To fully design a control system for a given plant, several problems have to be solved:

- Measurements: Data sets have to be acquired from the plant to get an idea about its behavior.
- Modeling: If the physics of the plant is known, a model has to be developed.
- Identification: The parameters of the model have to be estimated.
- Controller design: An appropriate controller structure has to be chosen. Its parameters are then optimized to guarantee the required design parameters.
- Simulation: The controller is to be tested on a plant simulation to prove its behavior.
- Implementation: The controller is implemented in hardware or on a real-time system and tested with the real plant.

Although few, there are some applications available which do provide visual methods to approach the mentioned design steps. However, little or no importance has been put on the design process as a whole.

Designing a control system is very rarely done in one single step. Many methods and algorithms need to be first selected and then applied correctly to the appropriate data sets. In current tools, this is mostly done by typing command lines on a computer keyboard. Parameters are set by the user and results are returned by the chosen algorithms, mostly in form of named matrices. These results are evaluated with respect to some design criterion. If this criterion judges the solution to be not satisfactory, some part of the design has to be repeated, with either different algorithms or different parameters. Previous results are revisited and in the end compared to each other in order to find the best design. The amount of data produced during a design session is immense, and people rarely have the discipline to name their results in a consistent manner in order to still be able to distinguish them after a few hours of experimenting. In [36], Ravn and Szymkat look at this design cycle under the aspect of a software engineering perspective. One of their conclusions is that there are currently no CACSD packages available which support the designer in the iterating design task.

The *Leporello* package is addressed to that problem. *Leporello* was designed to bring order into data and algorithms used in the control systems design cycle, the order which in other tools has to be maintained by personal discipline of the user.

Special emphasis was put on a fully graphical user interface. The advantages of visual data and program representation should not only be used in single algorithms as this is done with other visual methods in software engineering. During the control system analysis and design iteration, the user builds the data history graph interactively on screen. The abstract data objects (systems, signals, etc.) are displayed together with the algorithms which produced them, and the algorithms which were applied to them. The resulting tree structure (*action tree*) is one of the central parts of *Leporello*.

This tree is not only the visualization, but the user interface of the package. Parts of it can be automatically reproduced, which may be regarded as a sort of visual programming by example. The Leporello package as it exists today allows a fully graphical interactive controller design, which contains a whole range of different visualization techniques.

1.5. Scope, Contributions and Organization

1.5.1. Scope

In this thesis we will take a look at visual methods found in both software and control engineering. Useful techniques are evaluated by asking the following questions:

- What visual methods are used in automatic control?
- Where do we need software in automatic control?
- What visual methods are used in software design?

Fig. 1.2 shows the following (numbered) areas where we will be looking for answers:

1. Visual methods used in automatic control which might of interest in software engineering.
2. Visual software design methods which may give additional insight into control algorithm structure and behavior.
3. Visual methods used in both engineering fields.

All applications and examples using visual languages for their implementation are aimed at solving singular problems during the control systems design cycle. The Leporello package presented in this thesis is located in the area labelled '4' in Fig. 1.2. By presenting a visual representation of the design process as a whole, we extend

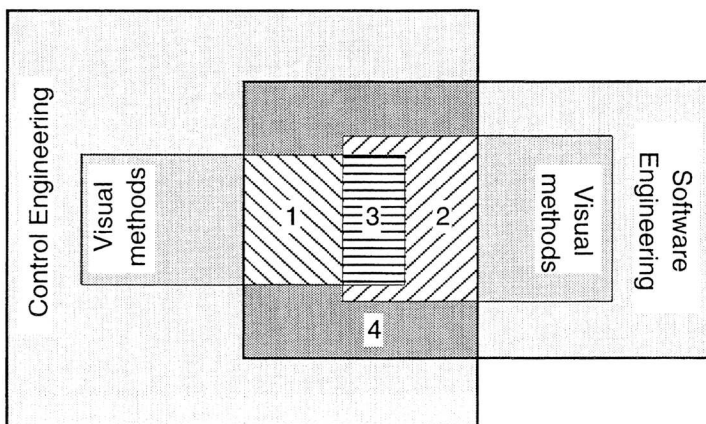


Figure 1.2: Scope of this thesis

areas 1, 2 and 3 to cover some parts of area 4 which has not yet been a target for graphical display (The sizes of the rectangles are not intended to reflect any quantitative or qualitative judgements).

1.5.2. Contributions

1. The successful use of the concepts of scientific visualization and visual languages in automatic control applications is shown in Chapter 2. The examples presented show an interesting future of these disciplines in CACSD.

Furthermore, the main contributions of the Leporello package presented in Chapters 3 and 4 are:

2. The storage of the complete computer aided design process in a tree shaped data structure, the action tree, similar to an engineering data base. All information necessary to create any stored result are contained to the extent, that any step can be re-executed automatically. The automatic task repetition is one of the great advantages of the Leporello package.
3. A new user interface is proposed for a CACSD package. The display of the action tree on screen gives the user full overview over the whole design project. In addition to the presentation of the results, the action tree display provides structured access to all tasks necessary for a complete controller design, including measurements, identification, design, and implementation.

1.5.3. Organization

The thesis is divided into the following chapters:

Chapter 2: We will show several approaches to software visualization. Data visualization is covered as well as the graphical display of programs. We will present some software packages currently available which are suited to solve selected control problems. We end this chapter with a brief discussion of the use of visual languages in general and in control engineering in particular.

Chapter 3: This chapter shows the main ideas behind the Leporello package. Its visualization of the design cycle is presented as well as some basic object oriented techniques which were used for the implementation.

Chapter 4: The Leporello action tree and its main features are in the focus of this chapter. Several tasks needed in control systems design are available in the tree, each one is presented shortly. We will give an overview over implementation details and we will show how the task is visualized in the tree picture.

Chapter 5: Leporello was used to solve a controller implementation problem. Together with the steps taken in the action tree we will give some examples of user interface details.

Conclusions: In this chapter we will briefly review some of the advantages of visual methods in automatic control as well as the advantages gained in the Leporello pack-

age. We do not consider its implementation to be complete, we will therefore indicate a few directions of further work. Some propositions show missing features which only need little additional programming effort, other propositions open wide fields of concepts which are still to be developed.

Leer - Vide - Empty

It is commonly undisputed that visual techniques play an increasing role in computer science. We have seen in the introduction that graphical representations do have several advantages over sequential text in many areas of engineering. What we are now looking for is a picture oriented view on some selected problems from software engineering, especially from software engineering in automatic control.

First we have to resolve a possible ambiguity in the term '*software visualization*' itself. Is it the visual representation of software programs or is it the visualization of general data by use of software? Both interpretations are possible, and both are worth being looked at. We will divide our investigations into the following two parts:

- *Scientific visualization*: Visualization of any sort of scientific data by computer software. This includes anything from two dimensional line plots to animated virtual reality environments.
- *Visual programming*: Design of computer programs by drawing diagrams on screen. These diagrams show different aspects of a program and visualize its functionality and structure. Furthermore, executable code is automatically created from the graphical display. Visual languages should allow easier programming and better understanding of complex software systems.

Mixed forms of these two areas are possible. If we look at a computer program as being a special kind of data, scientific visualization includes visual programming. On the other hand, programs which create visualizations of data could well be programmed visually (e.g. Example 3). We will therefore try to investigate how software can be used to create pictures from raw data, and how pictures can be used to create software.

Another question will help us to approach the problem in order to support control engineering best: where does a control engineer use computers, now and in the future? The search for visual methods in these areas of software engineering will therefore produce the results the most usable in CACSD.

	on-line/real-time	off-line
analysis	measurement	signal and system analysis
design	controller implementation	controller design and simulation

Table 1: Software use in control engineering

Computer use in automatic control in general may be divided into the areas shown in Table 1. The alert reader might argue that there are many more tasks to be done for a successful control system design. Modelling and identification for example are not mentioned in this table. These tasks, which all consist of transforming or generating control system representations, are included in the 'design' row of Table 1. We will mention this again in more detail in Chapter 3.2.

If we compare Table 1 to our previous distinction of software visualization, we see that there is a correspondence between scientific visualization and analysis on the one hand, and visual programming and design on the other hand. The quality of a controller design or a system identification is very often judged by looking at simulation results or measured signals which are compared to each other. The control systems design cycle on the other hand mostly consists of the application of algorithms, or in the case of the controller implementation of real-time code to be executed. In both situations the control engineer is forced to write his own code to achieve these results, at least if there are no libraries available which provide the necessary algorithms. For control systems design this is very often the case with modern CACSD tools; the implementation mostly has to be done 'by hand', by implementing the control law in C or some other programming language.

We will therefore put special emphasis on control engineering requirements in the course of the discussion of both scientific visualization and visual programming. There might be other approaches and methods in these fields – if their application in control engineering is not feasible, they merit very little mention in the chapters to come.

2.1. Scientific Visualization

One area where computers are most useful in science is visualization. With today's computer power we are able to simulate a wide variety of physical phenomena. Simulations often replace costly experiments and are even able to show data otherwise not accessible by sensors. These results contain a lot of information which, if not displayed otherwise, is concealed in long lists of numbers which are hard to interpret. Scientific visualization (the graphical display of data by computers) is needed to show these results on the computer screen in order to reveal as much information as possible.

The amount of information revealed may be regarded as the information transmitted to the reader (or viewer) by a picture. Miller [29] uses the terms known from information theory to calculate the channel capacity of different human senses by psychological experiments. He there shows that an increase in stimuli dimension augments the channel capacity, therefore increases the information perceived. Two-dimensional pictures are capable of providing more information than one dimensional data.

In recent years, these pictures have become more and more sophisticated. Simulations from meteorology or fluid dynamics (just to name two examples) are displayed in images which can hardly be distinguished from photographs. Virtual reality even goes one step further and simulates the real world in order to deliberately fool the viewer into believing it. In future visualizations we may come to the point where we will be able to walk through parameter space and explore simulation results.

2.1.1. One-Dimensional Displays

Today's visualization methods used in control engineering are still quite far from the aforementioned ideas. In many areas, the dimension of a visual display is simply one. In these visualizations, the value of a monitored variable is shown by the length of a bar, or the angle of a needle. The advantage of these instruments is the quick qualitative information they provide, not only about the depicted variable, but also about its rate of change. Whenever these informations are to be read by an operator at a quick glance, one dimensional displays are widely used and still unsurpassed.

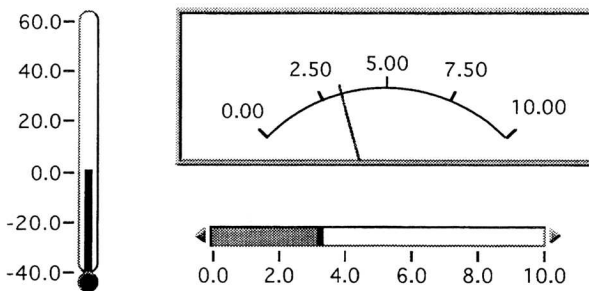


Figure 2.1: One dimensional data visualization devices (examples from LabVIEW)

2.1.2. Two-Dimensional Plots

In pre- and early computer days control engineers used two dimensional diagrams of all sorts for control systems analysis and controller design. For most of these diagrams (Bode, root locus, etc.) approximative methods are available which allow their qualitative drawing on a sheet of paper with very little calculation effort, but also with limited precision. Other results (measurements, simulations) were available in form of voltages and could therefore be displayed on an oscilloscope or an x-y plotter. Like

diagrams in most other fields of engineering, these drawings were limited to two-dimensional line graphs.

Early CACSD packages were able to calculate those diagrams more precisely. However, graphics output was limited to the capabilities of an alphanumeric display. Advanced graphics of today's desktop computers allow more precise drawings with additional features. Automated axis scaling and labelling, multiple plots, and different line styles and colors are just a few of them which are considered state-of-the-art in control design packages.

However, since most of these packages are command-driven, all graph features as well as the plots themselves are created and accessed by textually entered commands. A simple Bode diagram displayed in Matlab requires the piece of code shown in Example 1 (excerpt from the Bode script from the Matlab control toolbox).

Example 1: Bode plot in Matlab 3.5



```
subplot(211)
semilogx(w,20*log10(mag),w,zeros(1,length(w)),'w:')
grid
xlabel('Frequency (rad/sec)'), ylabel('Gain dB')
subplot(212)
semilogx(w,phase,w,phase180*ones(1,length(w)),'w:')
xlabel('Frequency (rad/sec)'), ylabel('Phase deg')
grid
subplot(111)
```



Each change of a plot parameter (everything from grid display to the text of an axis label) requires full re-entering of the lines of Example 1, or, if this piece of code is contained in a script, a change in the script file and its re-execution. Furthermore, if we want to read exact values from the display, we have to do additional programming which requires extensive Matlab scripting experience.

We started looking at two-dimensional plots with the traditional, curve based visual tools of control systems engineering. Now that we see how these graphs are brought to screen, we realize that they can mainly be used to look at, not to work with. In order to restore their 'tool' character we need easier (interactive) access to graph features.

Common tasks in plot windows are:

- read an exact value from a curve
- enlarge parts of the graph (zoom)
- change axis scaling
- show/hide plots
- show/hide a grid
- change curve line styles and colors

All these modifications should be accessible directly in the graph.

Example 2: Interactive signal plot tool in LabVIEW



The plot tool displayed in Fig. 2.2 was programmed in LabVIEW and is used to display curves from within the Leporello application. It is an example of an “intelligent” plot device described above. Axis scaling is changed interactively using the controls to the left of the graph frame. The buttons just below allow zooming into and out of the graph. The probe in the center of the graph (+) may either be moved by dragging it, or its position may be entered in the display below the graph. Axis ranges are changed by either dragging the probe outside the graph, or by selecting the axis labels and typing the new bounds directly.

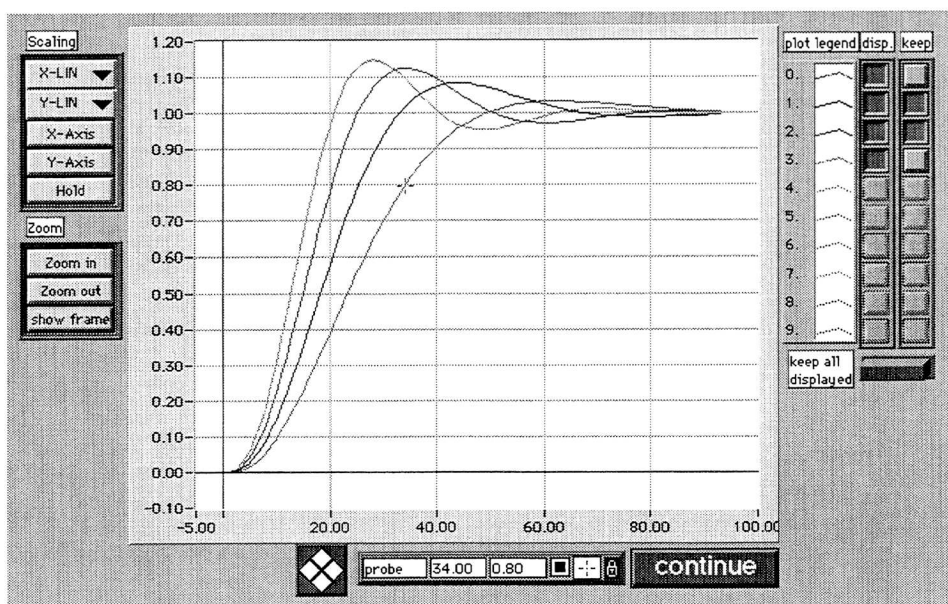


Figure 2.2: LabVIEW tool to display two dimensional plots in Leporello



2.1.3. Three-Dimensional Plots

Although some planar graphs lack the ease of use of other features of graphical user interfaces, these plots are nevertheless integrated components of today’s CACSD packages. On our way to extract more information from control data, we now have a look at higher dimensional plots.

The step into three dimensions is the most obvious. There are many situations where a third dimension proves useful. An evident case is given when data sets are themselves three dimensional. Examples are trajectories of three state variables or the variation of three parameters with respect to the fourth variable. Plots of this kind are one dimensional lines in three dimensional space. Other parameter variations produce two

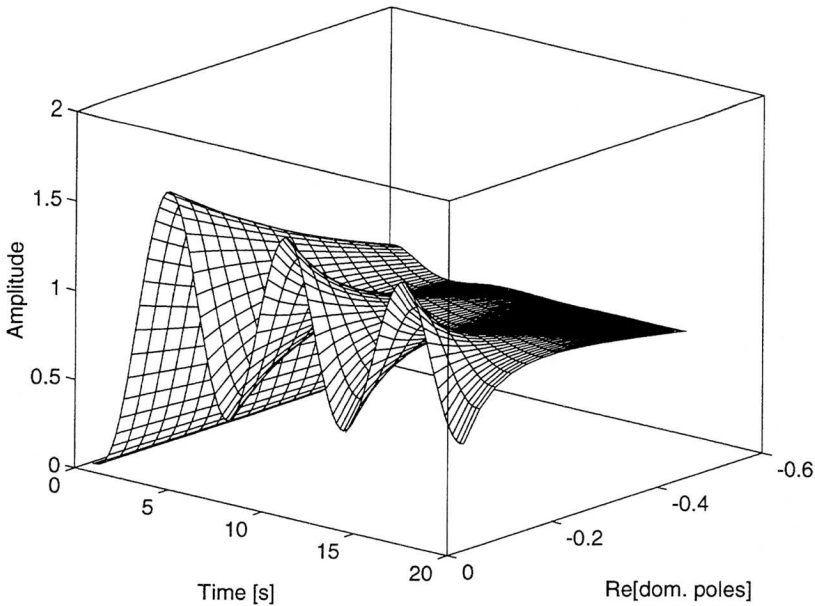


Figure 2.3: Mesh plot of step response vs. pole location produced in Matlab 4.1

dimensional planes. An example of this kind is a graph where simulation results are plotted in two dimensions, the real part of the dominant pole location is used for the third dimension (Fig. 2.3).

The complexity of three dimensional shapes is unlimited. Stability regions for systems with parameter uncertainties may be candidates for display as well as geometrically complex robot arm trajectories. However, these displays are still rarely seen in automatic control.

CACSD packages do have limited capabilities for three dimensional plots (the example in Fig. 2.3 was produced in Matlab 4.1). These limitations are similar to the ones previously mentioned for two dimensional graphs, although interactive manipulations are even more important for spacial displays. Choosing the right viewing angle to look at the picture should be possible by dragging the picture itself into the correct location instead of typing numbers which indicate the viewing direction. Some mathematical packages (Maple, Mathematica, and in its latest version Matlab) include tools to handle these more complex plots.

2.1.4. The Use of the Fourth Dimension

Difficulties increase if our imagination has to deal with dimensions exceeding three. Spatial objects are usually displayed by just projecting their third dimension on the two-dimensional drawing plane. To include a fourth dimension in a picture needs more abstract concepts. A few of these ideas will be shown here.

In some cases it is possible to use a color scheme to indicate a fourth variable on a three dimensional object. Examples are temperature, pressure, or similar measurements on a surface in 3D-space. The same idea is sometimes used to display the altitude of a three dimensional surface (*i.e.* a landscape) on a planar map.

Dynamical pictures: Another method to show four dimensions is the use of time in screen drawings, *i.e.*, to display moving objects. This approach is obvious if the fourth dimension to be displayed is the time itself. If we want to display the transient movement of a membrane in three dimensional space over time, a movie of the membrane is certainly giving the most appropriate information about its behavior.

Apart from time, other variables may be mapped onto the time instances of a movie. Such a movie shows how the displayed shapes move or behave as a given parameter changes. The surface in Fig. 2.3 could be animated by moving the imaginary parts of the poles in addition to the real parts. We could so watch the change of the oscillation frequency in the moving surface.

The information contained in the static surface could also have been displayed as a movie of a two dimensional step response graph which changes with pole locations. In both the two- as well as the three dimensional moving images the parameter variations do not necessarily have to be predefined. The user could as well modify them interactively on screen and watch the effect of the alterations on the picture displayed. With the appropriate interactive controls (sliders, knobs, etc.) the impacts of continuous parameter changes are intuitively visible on screen.

Animated Pictures: Very often, simulations are used to investigate the movement of some physical objects in space. A bouncing ball or the famous ejector seat simulation are just two popular examples of this sort. If the results of such simulations are just displayed as two dimensional plots, their interpretation requires some imagination to see the correspondence between curves and the real world system. The behavior of this system is perceived much easier, if we display its picture and animate it using the simulation results. Instead of just looking at several curves we could actually see a robot arm move in space, or a pilot being ejected from a plane.

Interactive algorithms: In automatic control, dynamic pictures are not only an excellent tool for system and signal analysis, they are also well suited as a user interface for other algorithms. Many of the algorithms in use today are easily expanded to display the behavior of some internal parameters as the calculations proceed. Instead of just being presented with an inappropriate result at the end of lengthy calculations, the user is able to detect algorithm misbehavior earlier, and to take measures against it. As an example, the state of an optimization algorithm could be displayed in parameter space.

The measures mentioned are either stopping the algorithm and starting it again with different settings, or interactive parameter tuning by the user. This is closely related to the algorithms previously mentioned, which allow interactive parameter adaptations.

Example 3: Interactive Lead-Lag controller design ▽

The LabVIEW instrument in Fig. 2.4 shows an example of an interactive algorithm. It was used with the Leporello example described in Chapter 5. The Bode diagram of the plant is displayed together with the diagram of the open loop controlled system. The controller parameters on the left of the instrument can be manipulated interactively. The resulting controlled system and its Bode diagram are instantly calculated and displayed. The user can so monitor the influence of the controller parameter changes on the Bode diagram. In addition, critical frequency, amplitude margin and gain margin are calculated, displayed numerically and marked in the diagram with a vertical bar.

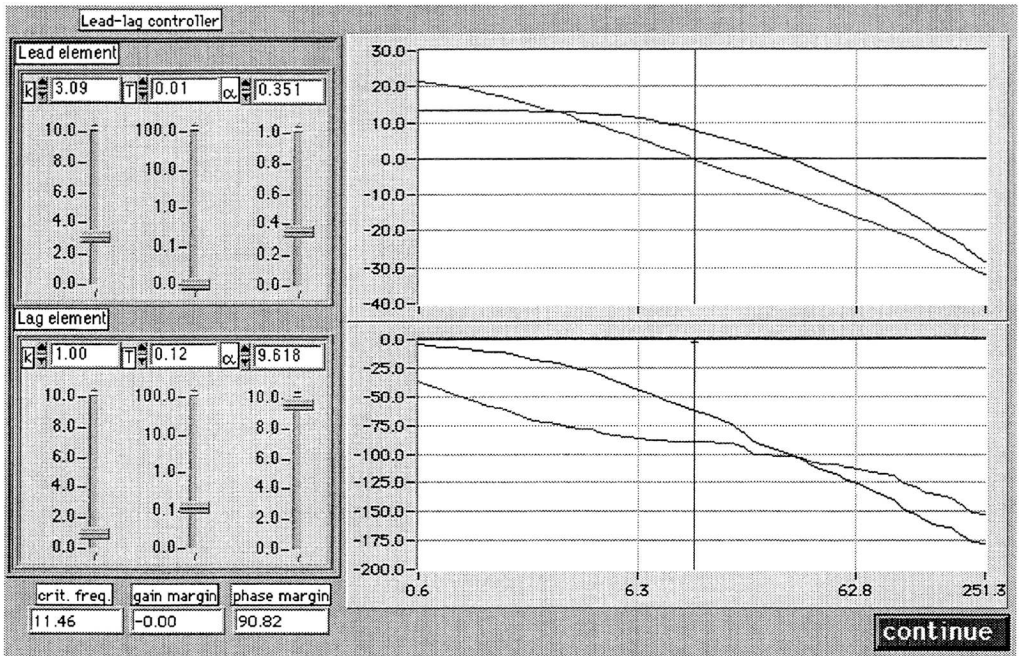


Figure 2.4: Interactive lead-lag controller design in the Bode diagram △

2.1.5. Summary

The methods presented in this chapter all served to present the information contained in scientific data in an intriguing manner. By choosing the appropriate display, the solution of a given problem is often found by just looking at the resulting graphics. These visual methods to analyze systems or signals have always played an important role in control engineering. With today's computer power we not only have the means to display the necessary pictures, but also to work with them. Including more dimen-

sions in pictures increases the amount of information displayed, and in addition often reveals data relationships which are not perceived otherwise.

Dynamic pictures and interactive algorithms present a new way of working with a computer. The user is involved more directly in algorithm execution and has therefore greater influence on its behavior. Experimenting with parameter variations becomes easier and allows quicker evaluation of alternative solutions. Without question, these methods will enhance computer usage in automatic control.

2.2. Visual Programming

From the visualization of numerical data sets we now come to the pictorial display of a program itself. Like traditional, textual programming languages, visual languages are used to create executable code. A visual program is designed on screen using graphical elements (typically lines, icons and boxes) by use of a graphical input device (mostly a mouse). Graphical programming paradigms are very often based on traditional visual representations of software or on visual methods traditionally used in the domain where the program is going to be used. On our quest for visual language applications in automatic control we will therefore not only look for language implementations, but also for visual methods in general which might serve as paradigms for a visual language. These paradigms usually display one single aspect of an implementation. However, very often a visual language emerged from one of these diagrams and added views on parts of the program which are not displayed by the initial method.

In some way, visual programming may be regarded as a special case of scientific visualization, with the following differences:

- The data displayed are computer programs. Compared with the relatively unstructured simple numerical data sets displayed in the examples of Chapter 2.1, a program is of much higher complexity. Although it is available in computer memory like any other kind of data, the information it contains is much more intricate. Depending on the purpose of the program, different aspects may be important to be visualized.
- The data set, in this case the program, is not only visualized by, but created from the graphics. The data sets mentioned earlier were produced by another source (measurements, simulations, etc.) and displayed only by visual software. Besides displaying diagrams of an existing program, visual programming languages are designed to create an executable program and to support the programmer in this task.

The basic effect of visual programming however is almost the same: by using graphics we would like make use of our image processing capabilities and get more information from the software itself. The functionality of a piece of code should be presented

more clearly; its behavior should be seen from the program's graphical representation. Furthermore, similarities to concepts or diagrams from the programmer's own domain should ease program development and understanding.

Syntactical formatting: Increasing the display dimension to transmit more information holds also in the case of visual programming. Conventional textual programming languages are basically one dimensional, a character stream like any other piece of text. To get an idea of what a program is doing is close to impossible if it is displayed as unformatted text. An unstructured code segment is shown in Example 4.

Example 4: Unstructured stream of code:

```
s1; if c1 then s2; repeat s3 until c2 else if c3 then while
c4 do s4 end else if c5 then s5; end; s6; end;
```

The use of syntactical formatting increases code readability drastically. Program lines of the same syntactical unit (IF, REPEAT, FOR statements, etc.) start at the same indentation level. This is done either by hand or by using a so called pretty-printer, a tool which parses the source code and inserts line breaks and tabulators where necessary. The code in Example 4 is much easier to read after its formatting, which is done in the example continuation.

Example 4 (continued): Syntactically formatted code

```
S1;
IF C1 THEN
  S2;
  REPEAT
    S3
  UNTIL C2
ELSE
  IF C3 THEN
    WHILE C4 DO
      S4
    END
  ELSE IF C5 THEN
    S5;
  END;
  S6;
END;
```

By doing this, we introduce a higher dimension to our otherwise linear textual display. Although the plain text itself contains all the information necessary for the computer to execute the example, we add extra graphical information to the layout in order to transmit this information not only to the computer, but also to the human who wants to work on this program. Even though the program does spread over the two dimensions of a page, we do not consider it an example of fully two-dimensional programming. In addition to the one dimensional text flow, the position of the formatting characters make for part of another (fractional) dimension. Since it is more than one-

dimensional but not two-dimensional, the definition of dimension 1.5 for this representation could be thought of.

Higher dimensional programs: At the time the popular programming languages were developed, one dimension was adequate. A computer program mostly executed sequentially, along the path specified by the compiled code. The sequence of statements laid out in the program text was always traversed in an order only depending on a limited number of conditions, *i.e.*, given a limited number of information (program input), program flow could fully be traced from the source code only. During user interaction, for example, all other processing was stopped and resumed after the user finished the input.

Today, many popular software design techniques abandon the formerly linear program flow. Although a single processor still executes programs sequentially, the order of execution cannot be traced from the source code. One example where this is obvious is real-time software with parallel processes. Program execution mostly depends on the inter-reaction of concurrent processes, which in turn depend on external events such as interrupts or time events. Timing of these events may be crucial to program execution.

Another very popular paradigm is object oriented programming¹. Program flow in an object oriented environment is not specified by a sequence of procedures and functions, but by objects exchanging messages and the receiver of a message reacting on it. This reaction may initiate messages to other objects; a whole message passing scheme is established at runtime. If messages are not only created by objects but by external events (user interaction, interrupts), we get the same software complexity as with parallel processes. Today's graphical user interfaces (Macintosh OS among others) are all based on an event processing loop, which reacts on any user interaction (mouse movement, keyboard events, etc.). In the Leporello project, in addition to the user interaction event handling, asynchronous message passing is implemented in the communication links to externally accessed numerical packages. With these packages located on remote computers, in the current state of development Leporello supports up to four interacting processors (if we simply reduce the user to an external processor). This produces situations where parallel processes and object oriented design are mixed (see Chapter 4.1, "*Algorithm Node Execution*", on page 4.1.).

Higher dimensional programming languages: The inherent parallelism of the aforementioned examples is very difficult to show in textual program descriptions. Visual languages, which make use of dimensions two or higher, present themselves as an appropriate means to portray these concepts. The additional dimensions offer the possibility to display various views on a program which are not easily to be seen in a textual description. A natural depiction of parallel processes for example could be in parallel columns next to each other. For object relations, there are several aspects to be

1. In the introduction to the Leporello project (Chapter 3.3.1) we will give an overview over the most important concepts of object oriented software design.

visualized: message passing schemes could be shown in a graph, subclass relations or object hierarchies are best represented in trees (cf. Fig. 3.4 on page 64 or Fig. 4.2 on page 78).

In the history of visual programming the following *software aspects* have proven to be candidates for visualization¹:

- Control flow, showing the sequence of operations
- Data flow, showing what happens to data sets
- Data structure, showing how data sets are specified
- Topology, showing how parts of the program interact.

The decision which part of a software concept to emphasize in a given graphical programming environment is governed by its designated range of application, which in our case is automatic control. In Table 1, "*Software use in control engineering*", on page 16 we have seen that there are primarily two circumstances where a control engineer not only uses, but also writes software: One is the programming of new algorithms necessary for control systems design (a situation more common in research environments than in everyday work in industrial development) and the other one is the implementation of digital control systems on a real-time target computer system, where control laws have to be turned into software. In the latter case, drawing block diagrams for controller specification is one of the traditional visual methods in automatic control. The use of a visual language in this situation is evident; we will show some examples in the chapters to come.

We will now take a closer look at control flow and data flow descriptions, the more important aspects of automatic control software.

2.2.1. Control Flow Descriptions

The emphasis in control flow descriptions is on the sequence of operations in a program. The diagram shows the conditions necessary to execute each statement. Basically this is just following a linear sequence. More complicated situations may also include branches, loops and possibly parts which indicate parallel execution. Control flow descriptions are an appropriate visualization in situations, where the emphasis is placed on an *agent*. Instructions directing a robot to complete a desired task are an example which is readily visualized in a diagram which indicates the sequence of movements and possible events calling for reactions (sensor signals, etc.).

Nassi-Shneiderman diagrams: A very early, and also very popular description is given in [32] by Nassi and Shneiderman, who propose a box notation of structured programs. Fig. 2.5 shows the Nassi-Shneiderman diagram of Example 4.

We see that block descriptions of this sort have little analogy to methods traditionally found in control engineering. This limits their applicability. The use of "box-and-line"-

1. Please note that these are topics which are important for software development. Visualization of data at runtime was presented in the previous chapter and is not considered a subject of visual programming.

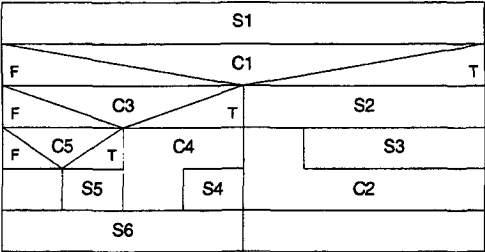


Figure 2.5: Nassi-Shneiderman diagram corresponding to Example 4

notations is more common in this field. A survey of possible software specification schemes, which basically use boxes, lines, or both, is found in [46]. Most of them have been proposed as graphical software specification tools and not as visual languages. Some of them do show similarities to the block diagram description popular in automatic control; we will have a closer look at selected examples later.

Flowchart descriptions: One notation dating from the times of assembly language is the flowchart description. Analogous to assembly language, its only control structure is a branch statement. In Fig. 2.6, Example 4 is laid out in a flowchart. The flow of the program can easily be traced; it is obvious which statement depends on which selection statement or is contained in which loop.

Although this sort of diagram is very popular for sketching software design details on paper, its use as a visual language is limited. Compared to high level programming languages, it does not support their concepts of data abstraction and their more disci-

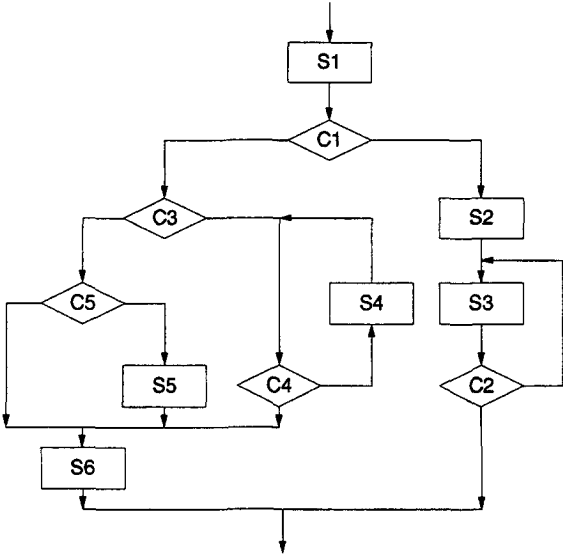


Figure 2.6: Flowchart description of Example 4

plined control structures (FOR-, WHILE-loops). However, extended flowcharts may be a possible control flow visualization in a language supporting different views on programs (see the GenT approach described in Chapter 2.3.4).

If we look for control flow diagrams in control applications, we see that especially in the field of discrete event dynamic systems (DEDS) there are some methods being used which illustrate this program aspect.

State diagrams: One of these descriptions is the state diagram. A directed graph is used to depict system behavior. Discrete system states are connected by directed branches which define state transitions. A state transition is triggered by the corresponding event. The action executed as a reaction to this event is either assigned to the branch or to the new state reached, depending on the chosen topology.

Example 5: Automatic garage door controller



To illustrate control flow specifications we will use the example of an automatic garage door. It consists of a door, a motor, an 'open' and a 'closed' sensor. The user can issue open or close commands, the door stops automatically when reaching the open or closed position. In the discussion of the example we will not handle irregular events (open command when the door is open, closed signal when the door is opening, obstacles, etc.).

In Fig. 2.7, we see the state diagram of this door. It shows the commands the motor receives upon user request or sensor signals.

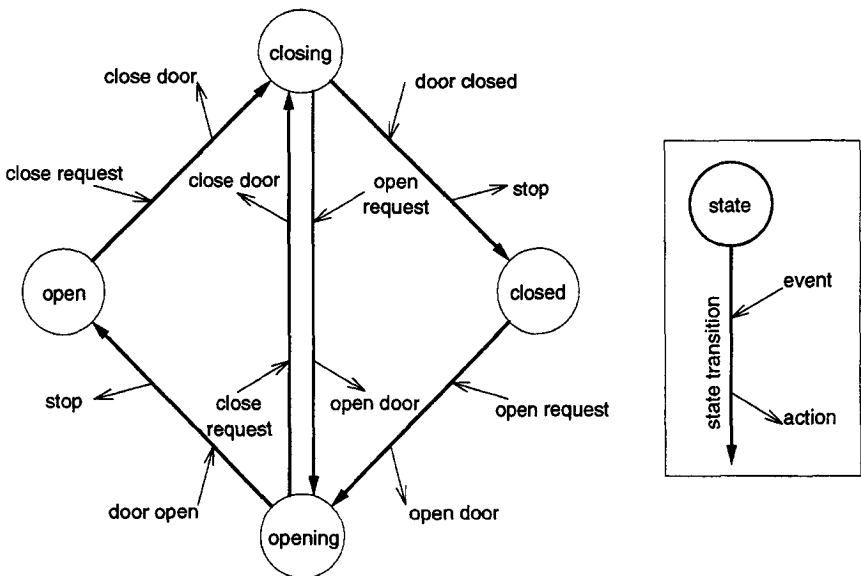


Figure 2.7: State diagram of an automatic door



Since a pure state diagram tends towards very high complexity (exploding number of system states), it is not likely to be used to fully describe software. Nevertheless there are some visual languages which have their roots in these diagrams (e.g. [6], [16]).

Petri nets: Another method widely used in DEDS is the Petri net. Its appearance is similar to the state diagram, its operation however is quite different. The elements of Petri nets are places (s-elements, circles), transitions (t-elements, bars or rectangles) and tokens (bullets). Places are connected to transitions by directed branches (arcs) and vice versa, connections cannot exist between elements of the same types. The system state is indicated by the distribution of tokens, called marking. State changes occur when tokens are moved to other places by firing transitions. The transition then removes one token from each input place and sends one token to each output place. A transition is allowed to fire if tokens are available on all its input places.

Example 5 (continued): Petri net description of the garage door ▽

The state diagram of the garage door example in Fig. 2.7 is transformed almost identically to the Petri net in Fig. 2.8. The state of the door is defined by the one token positioned on the 'open' place; the token (and therefore the control of the door) flows clockwise around the main circle, triggered by the external sensor and request events (grey places).

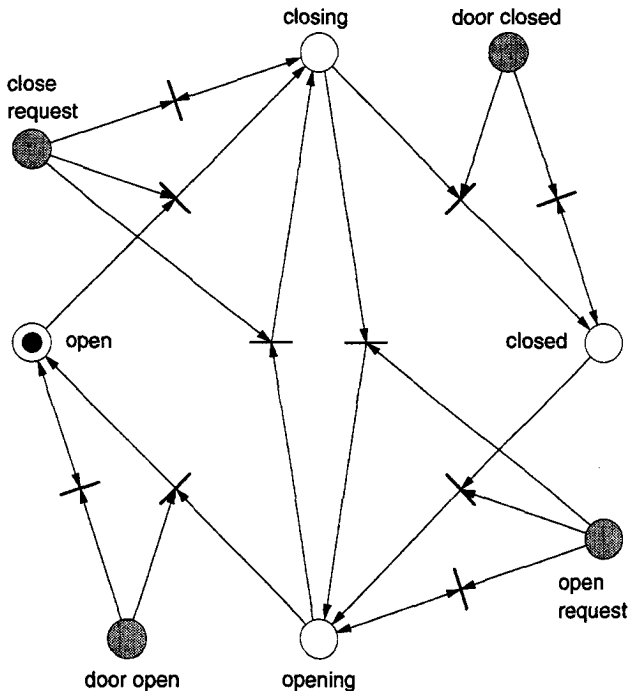


Figure 2.8: Petri net of the automatic door of Example 5

If we compare Petri nets to state diagrams, we find a number of analogies: the places of a Petri net may be compared to the states of a state diagram, the transitions are similar to events. The difference is in the definition of the system state: the state of the Petri net is given by the marking of the net, *i.e.*, if several tokens are circulating all the positions of these tokens form the current state. In the state diagram, the state of the system is given by one position in the net. If one token only is available in a Petri net, both descriptions are quite similar.

The possibility of multiple tokens circulating independently is the great advantage of the Petri net over the state diagram. It allows the display of the control flow of multiple processes, each described by its own token. This makes the Petri net one of the preferred tools for real-time software specification and analysis.

Both the state diagram and the Petri net can not be used as a visual language or to represent uniquely a piece of code, since they do not provide enough information. Data relations and manipulations are not shown at all. Besides their names, the state diagram does not show anything about the actions triggered by the state transitions. Additional views on the displayed code are necessary. However, in applications which emphasize discrete states of one or more agents these diagrams may not only be used to specify the control flow of a program, but also to animate it at runtime. Tokens moving in a Petri net show clearly where a discrete supervisory control algorithm is waiting on process interaction or is running in an endless loop.

2.2.2. Data Flow Descriptions

One of the software aspects missing in control flow diagrams, data relations and manipulations, is emphasized by data flow diagrams. The examples presented in Chapter 2.2.1 did show quite clearly which statement was executed at what time or under which conditions. They did not show any effects of the statements other than a general 'state change' of the system displayed. In particular, data objects or elements and their manipulations did not show in any of the views.

The garage door example only displayed the discrete state of the door (closed, opening, etc.). We did not have any quantitative information about the door, like its current opening angle or its velocity. In order to simulate the continuous door movement, we could introduce a counter variable and the increase or decrease operation which is executed repeatedly until one of the limits is reached.

The data flow diagrams we want to take a look at are "box-and-line" diagrams like the ones displaying control flow. Nodes of a data flow graph represent operations; data elements are passed on the connecting directed branches from the output of one operation to the input of another. An operation node is ready to be executed as soon as it has received data on all its inputs, it passes its results to all the operations connected to its outputs, which in turn may become executable. Data objects are so passed through a network of operations.

This description of the principles of data flow graphs reminds us of the firing mechanism of Petri net transitions. And in fact, there are Petri nets which may not only be regarded as control flow graphs. Their extensions contain elements which allows their interpretation as data processing networks.

High level Petri nets [9]: In addition to the specification of ordinary Petri nets, the elements of these extended nets can contain attributes and commands, which influence transition firing and introduce attribute manipulation code. Tokens not only indicate the current system state by the place which contains them, they can also hold data records and transport them through the net. Transitions and arcs receive two additional features:

- By assigning a condition to a transition or one of its input arcs, it does not only need tokens on each of the input places in order to be enabled for firing, the tokens on these places must also fulfill these additional conditions.
- By assigning an action to a transition or one of its output arcs, the data records of a token passing this transition or the assigned arc are changed.

A high level Petri net which makes use of these extensions may literally be regarded as a data flow diagram, since the tokens loaded with data records are 'flowing' along the arcs through places and transitions.

Example 5 (continued): High level Petri net of the garage door ▽

Let us review the automatic door example (Example 5). We mentioned the missing information about its current door angle. All we knew was the time instance when the door touched one of the sensors. If we define the door angle to be an attribute of the token which is reflecting the state of the door, we can monitor the increasing or decreasing value of the door position and detect its open or closed state. If we make use of the timing facilities available, the simulation is already a little closer to reality.

The extended net shown in Fig. 2.9 contains these changes. It was created using SystemSpecs [54], which accounts for the square transition nodes. The door state token moves from its "door open" place to the "door closing" position as soon as the "close door" event occurs. Contrary to the standard net displayed in Fig. 2.8, it does not wait for the sensor event to be signalled. Both transitions containing the condition label could now be reached from the current state, their condition statements ($x > 0$ and $x \leq 0$) specify exactly which one is to fire next. The token still containing the value 10 is passed as variable x to the upper transition; $x-1$ is returned to the "door closing" place. The token oscillates between place and transition until its value has reached zero. The lower transition then receives the token and passes it on to the "door closed" place.

We will later discuss the SystemSpecs visual language and its mixed control and data flow aspects.

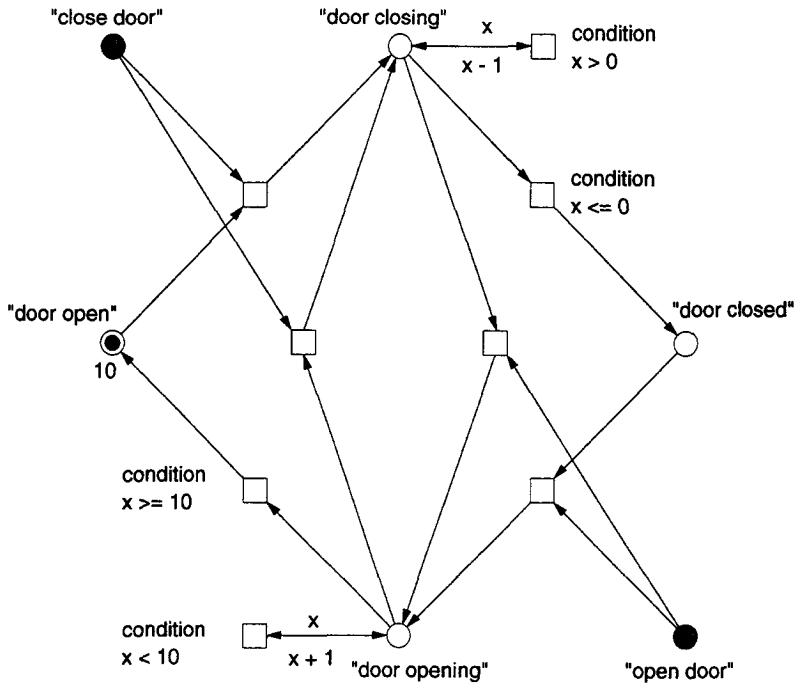


Figure 2.9: High level Petri net of the automatic door with modelled door position

△

Block diagrams: Another form of data flow descriptions very popular in control engineering is the block diagram. As it is the case with Petri nets, this method of visualizing control problems has been used long before the appearance of computers in the field. However, limited graphical capabilities and computer power banned it from the electronic desktop until recently. Even the standard textual simulation package ACSL [47] now ships with a block diagram editor in its latest version.

General block diagrams display a system whose main components are represented by boxes; their relationship is indicated by connecting lines. In automatic control, these relationships are directed signal flow paths. The direction of these paths is well defined. Each one has its source and one or more recipients. The data types passed along these connections are numerical values, signals. They are passed on directed branches between systems which, by using the input values and some internal states, produce their output values.

So far, this corresponds to the general description of a data flow diagram. The difference is in the very specific structure of the control flow in block diagrams. The control systems represented are not designed to handle single numerical values. The variables passed are continuous or discrete signals, *i.e.*, a whole series of numerical values are sequentially processed. The main (and in most tools the only) control flow feature is therefore the loop which repeatedly evaluates the diagram.

Example 6: Primary control of a synchronous generator ▽

As an example when discussing block diagram structures, we will use the primary controller of a motor-generator group at the power network simulation plant in use at the Automatic Control Lab. Both frequency and active power are used to generate the torque control signal. By changing the k_1/k_2 ratio we are able to influence which of the values (frequency or power) is to be favored. Depending on the type of the turbine the frequency is allowed to vary in a smaller or larger interval. The block diagram of this controller is shown in Fig. 2.10. We will re-use this controller structure in the next chapter as an example for controller implementation.

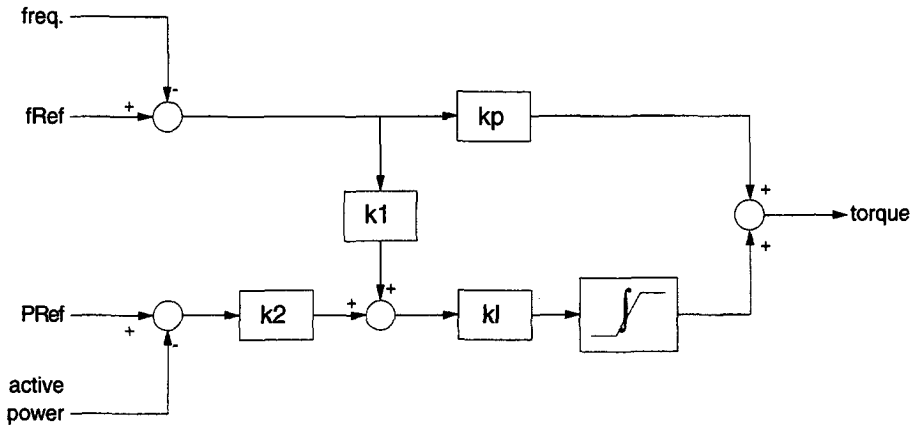


Figure 2.10: Controller structure used for primary control in the power network simulation plant

The execution order of a series of discrete blocks follows the forward data path, which does not cause major difficulties (as long as all sampling times involved share a common divider). However, we must keep in mind that feedback loops within our model do add an additional delay of one sampling period, *i.e.*, the value calculated in one pass is only available at the input of the loop in the next pass.

More problems arise if continuous or mixed systems are modelled as differential equations and put together in one block diagram. Signals between sub-systems are now continuous. Since digital computers always operate on quantized time instances, the straight forward evaluation of the blocks involved is not possible. In this case, the control flow does not necessarily follow the signal paths. In most simulation packages, sub-systems are collected and sorted to form an integral description of the system as a whole. The simulation of either the compound system or the series of sub-systems is controlled by an integration algorithm which takes care of the correct evaluation of the system description. This includes the proper calculation of feedback loops.

To achieve this, integration algorithms require the simulated systems to supply some of the following additional information:

- continuous systems: state derivatives, dX/dT .
- discrete systems: discrete states, $X(n+1)$.
- discrete systems: next time interval for update
- system outputs

During one integration step, system evaluation does not follow continuous time. Some integration algorithms first calculate the boundaries of one time interval and then compute values in between (e.g. the Runge-Kutta45 algorithm).

This rather complex implicit flow of control is specific to simulation languages. It does not correspond to the natural signal flow in the simulated system, it is therefore hidden from the user¹. Other system inherent control structures (state dependent events, switching events, etc.) are rarely seen in simulation languages in general, and not available at all in tools which provide a graphical block diagram. Although ACSL does support timed and state events in its textual implementation, these features are not available in its graphical user interface.

Limitations: There are several problems in automatic control which can not be solved using pure data flow descriptions. They all require some sort of control flow specifications. One example is an optimization algorithm which optimizes some system response: The system to be tuned is simulated repeatedly. After each simulation, the results are compared to the desired output and some system parameters are adjusted. This loop is difficult to display in a pure data flow environment. Other problems where control flow structures are required are exception handling in real-time applications or user input and output.

2.2.3. Summary

Most of the visual methods presented have been used as basic paradigm for the implementation of a visual language. However, in order to serve as a 'grown up' visual language which allows complete design of a general program used to solve automatic control problems, these methods need to be extended. This results either in mixed structures, where several software aspects are combined in one diagram (see the SystemSpecs and LabVIEW examples in the next chapter), or in different views on the same program, each displaying one specific aspect (the GenT example).

On the other hand, the use of a well known visual paradigm as the starting point for a visual language implementation eases its learning by people who are familiar with this visualization. These people often developed a thorough comprehension of systems modelled using this method. They can then adapt this knowledge to get a better insight into the domain specific software produced by these tools. Furthermore, the direct creation of executable code from the graphical design sketches removes one

1. The perception of dynamical systems and processes is known to be a difficult task for humans.

source of error and speeds design adaptation. Minor changes in a concept can be applied to the drawings directly, and automatic code generation keeps the whole implementation consistent with the changed concepts.

2.3. Tools

In the run of this chapter, we will discuss some of the software packages providing features which may in one or the other respect be regarded as visual programming. Some of them are applications only used in control engineering. Some others are developed to solve problems from related fields of engineering but may as well be used to handle control problems. Each of the packages presented here has particular features worth looking at. The examples are picked from a larger number of similar applications, they all were available at the Automatic Control Lab at the time of writing.

For each package, we will discuss the following aspects:

<i>Type</i>	Visual method the package is based upon
<i>Origin</i>	Original domain of the package
<i>Data flow</i>	Data flow handling
<i>Control flow</i>	Control flow specification methods
<i>Data types</i>	Data types supported, data type specification
<i>Data visualization</i>	Graphical display of data at runtime
<i>Program animation</i>	Possibilities to monitor program behavior at runtime
<i>Hierarchical diagrams</i>	Ability to break programs into parts
<i>Code analysis</i>	Methods available to analyze the diagram (completeness, correctness, etc.)
<i>Code generation</i>	Availability of code generation facilities
<i>Code extension</i>	Externally compiled code inclusion
<i>Real-time capability</i>	Real-time execution features
<i>Similar tools</i>	Related tools with similar features
<i>Example</i>	Solution of a simple automatic control problem
<i>Discussion</i>	Remarks about the usability of the tool in automatic control

The chapter is closed with a discussion about the advantages of visual programming.

2.3.1. SystemSpecs

Type: SystemSpecs is a tool which allows graphical specification and animated simulation of high level Petri nets [54].

Origin: Petri net simulation.

Data flow: As described when we introduced high level Petri nets (cf. Chapter 2.2.2), data flow is visualized by tokens holding data records. These tokens are passed through the net, thereby observing transition firing rules. Data elements are manipulated by transition inscriptions. These inscriptions are small pieces of textually programmed code written in *SpecsLingua*, a Pascal-like programming language.

Control flow: Basically, control flow is visualized by the net itself. The current state of the program is reflected in the token distribution. In addition, transitions and branches can be inscribed with additional firing rules, depending not only on the net topology, but also on the data values of the input tokens. A Specs net which makes use of extended firing condition was shown in Fig. 2.9.

Data types: The type of the data records assigned to tokens can be chosen from either the standard data types available in *SpecsLingua* (boolean, integer, real, etc.) or they can be structured and combined by declaring enumeration types, arrays or records. These structures are defined by writing *SpecsLingua* scripts. Visual specification of data types is not available.

Data visualization: Specs contains a special element class for user interaction (data visualization and user input) at runtime: the I/O elements. They are displayed as transitions which either produce or consume tokens depending on whether they have been designed as input or output I/O element. At runtime, their I/O window can be opened, where output I/O elements display the values of the tokens consumed and input I/O elements process user input. Once the user operates a control input element (presses a button or drags a slider) a token is released which carries the value the I/O element was set to (true or false with buttons, numerical values with other basic elements or records with several I/O elements contained in the same window). Examples of Specs I/O transitions and windows are displayed in Fig. 2.11.

Program animation: Visualization of program execution in Petri nets is obvious: tokens are animated and move along the graph branches. They display the current value of their data inscription. Fig. 2.11 shows two tokens currently in motion. Transitions whose firing conditions are satisfied are marked. For reasons of speed the animation can be turned off.

Hierarchical diagrams: A connected group of places and transitions can be coarsened into one element. Depending on whether the net was cut around places or around transitions the new hierarchical element is turned into a channel (sub-net contained in a place) or into an agency (sub-net contained in a transition.). Channels and agencies are indicated by grey net elements (see Fig. 2.9).

Code extension: By making channels or agencies reusable, Specs allows the creation of user-defined libraries, which can be used in nets other than the one where they were created. The link to the original specification is retained, changes in the original net affect all the reused nets.

In addition to graphically programmed libraries, there are textually programmed libraries which contain SpecsLingua scripts. Functions defined there can be used in any transition inscription.

Code analysis: In order to be simulated, a given net needs to be translated first. This is the moment when the net consistency is checked (correct data type passing and syntax check of inscriptions).

Furthermore, SPECS provides Petri net analysis tools which do not check the net 'syntax', but the net topology. Conflicts and invariants can be calculated and displayed, transitions can be monitored and token statistics can be collected and displayed.

Code generation: The full version of SystemSpecs allows code generation for faster simulation and prototype implementation. In addition to C source code, there is an option to create code which can be executed on a signal processor.

Code extension: SystemSpecs scripts can call externally compiled C code.

Real-time capability: In real-time software development, Petri nets already are a popular tool. Specs therefore provides a useful tool not only for real-time software simulation and analysis, but also for the implementation. The necessary timer access is available, the inherent parallelism of a Petri net does not require special language constructs to specify parallel processes. However, the speed requirements of a real time system can only be reached with the compiled code.

Similar tools: none known

Example: The garage door controller was shown in Fig. 2.9.

Discussion: Being a Petri net simulation tool, SystemSpecs has its strong points if a given problem is easily expressed with Petri nets. Direct code generation from the net

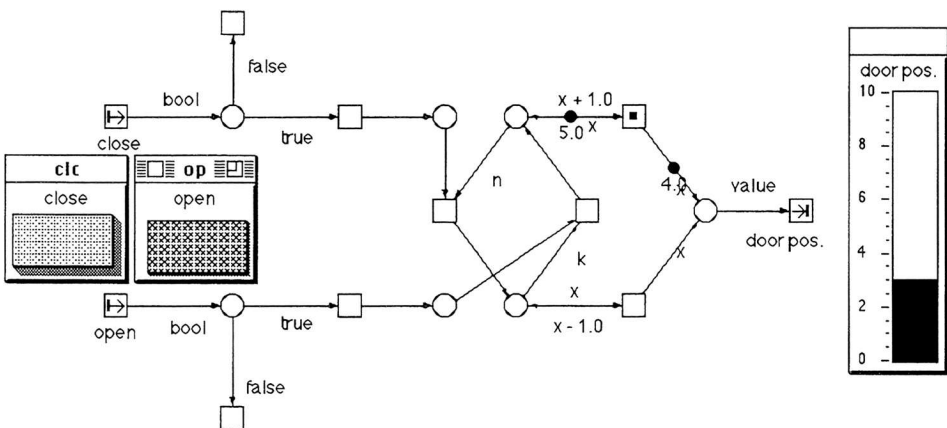


Figure 2.11: SPECS program animation: I/O transitions, windows and animated tokens

specification allows rapid prototyping and interactive debugging using the animated net. Problems which are not naturally modelled with Petri nets can be solved with SystemSpecs. However, these solutions tend to be very complicated. A discrete transfer function simulation implemented to experiment with SystemSpecs's data flow features filled a whole page. The same simulation in a more suitable environment (Simulink or LabVIEW) can be realized much easier.

2.3.2. Simulink

Type: Simulink is a Matlab extension, which provides drawing and simulation facilities to create and execute continuous and discrete block diagrams. It may not be regarded as a general purpose visual language. We present it to show the similarity of the more general languages to tools well known in automatic control. It was chosen from a series of packages which provide similar capabilities [53].

Origin: Simulation

Data flow: We have already discussed the data flow properties of a block diagram. Simulink does not provide any further data flow elements. The blocks used to build a Simulink diagram are taken from libraries, which contain most of the commonly used control systems structures, including non-linear, discrete or continuous parts as well as data visualization and data exchange blocks (take data from or return data to Matlab).

Control flow: The control flow in a Simulink model is controlled by the chosen integration algorithm. Any control flow mechanisms going beyond that are not provided. This causes difficulties with simulations which depend on the correct handling of state events (zero-crossing of variables, etc.). For example a system which contains friction cannot be simulated correctly using Simulink.

Data types: Being a Matlab-based tool, Simulink only provides two-dimensional matrix structures.

Data visualization: Some of the Simulink library blocks support data visualization features. The simulation can be monitored on scopes or Matlab graph displays which show the current state of the simulation variables (the scope in Fig. 2.12 is continuously updated at runtime).

Program animation: Simulink does not provide any process monitoring features which display anything else than the contents of variables.

Hierarchical diagrams: Any part of a Simulink diagram can be collected in a single subsystem and included in a user defined library. By creating empty blocks which are completed later, bottom-up development is also possible.

Code analysis: A Simulink block diagram can be analyzed under several aspects of control system analysis. Algorithms available are conversion into a linear system or steady state analysis. In addition to system analysis, the diagram is checked for syn-

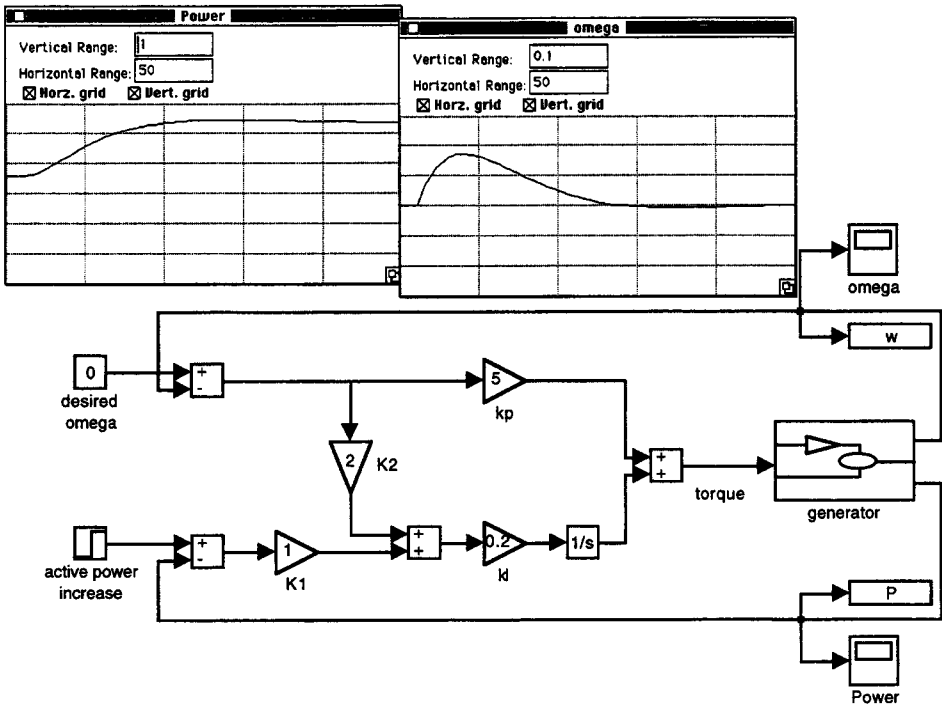


Figure 2.12: Simulink block diagram and scope output

tactical inconsistencies. However, certain error situations (missing Matlab variables, etc.) are only detected at runtime.

Code generation: A C-code generator is shipped with the latest version of Simulink on UNIX workstations. The compiled code is much faster than the Simulink simulation. It can also be ported to another target system, for example a signal processor based real-time system. In conjunction with such an external system, Simulink can be used as a prototype development environment for real-time digital control systems.

Code extension: The same rules apply for Simulink as they do for Matlab. It is possible to define parts of a block diagram by either writing Matlab script files or MEX-files. The latter are externally programmed subroutines which can be incorporated into a Matlab environment. They not only allow data passing between Matlab and the external code, they can also access Matlab functions from within C-code (The Leporello communication extension to Matlab depends on that feature).

Real-time capability: The generation of C-code from a Simulink model allows its execution in real-time. The compiled code contains the appropriate routines to assure correct timing, which are not available from within the block diagram editor. The necessary analog input and output routines can be included in the real-time code as well as in the block diagram by programming MEX-files.

Similar tools: ACSL/Graphic modeler, Easy5, MatrixX

Example: The example in Fig. 2.12 shows the simulink implementation of the generator controller from Example 6. Simulation results are displayed in the scope windows and are returned to Matlab in variable 'w' and 'P', where they can be processed further. The simulation can either be controlled interactively by specifying the integration algorithm and a simulation duration in the Simulink model window, or by a textual command entered on the Matlab command line.

Discussion: In its native domain (simulation), Simulink provides easy to use facilities to quickly design and test a control system. Its incorporation in Matlab does allow additional control flow programming, but only in text scripts. Its C-code generator also permits its use as a rapid prototyping tool for controller implementation. Any algorithms other than simulation are not feasible.

2.3.3. BlockSim

Type: Another block diagram drawing and simulation tool is BlockSim, developed at the Automatic Control Lab by Kolb and Rickli [20], [48].

Origin: Simulation and controller implementation in control education.

Data flow: As in Simulink, block diagrams may be drawn on screen. Its library of available blocks is limited to a few standard elements (signal generators, system descriptions and on PC analog I/O modules).

Control flow: The sequence of execution of the blocks connected in the block diagram is not derived from the diagram itself as this is done in other simulation tools. The user has to define the sequence before the first simulation. This is done on screen by clicking the blocks in the order the user wishes them to be executed.

A special feature is available because BlockSim does not require all blocks on screen to be contained in a sequence: Different block diagrams can exist independently in the same file. By assigning sequences which only contain blocks of one or the other diagram, simulations can easily be compared without switching files (This feature may also be regarded as a weak point: the completeness of a sequence cannot be checked and provides another source of errors).

Data types, data visualization, program animation and hierarchical diagrams: Being a public domain educational tool, BlockSim does not provide any of these features.

Code analysis and generation: Both features are not available.

Real-time capability: The implementation on IBM PC compatible computers does provide real-time features. A timer, analog input and output blocks are available. In addition to the simulation done for a limited number of samples, controller implementation may be tested indefinitely by using the infinite loop feature which allows real-time simulation until user interaction.

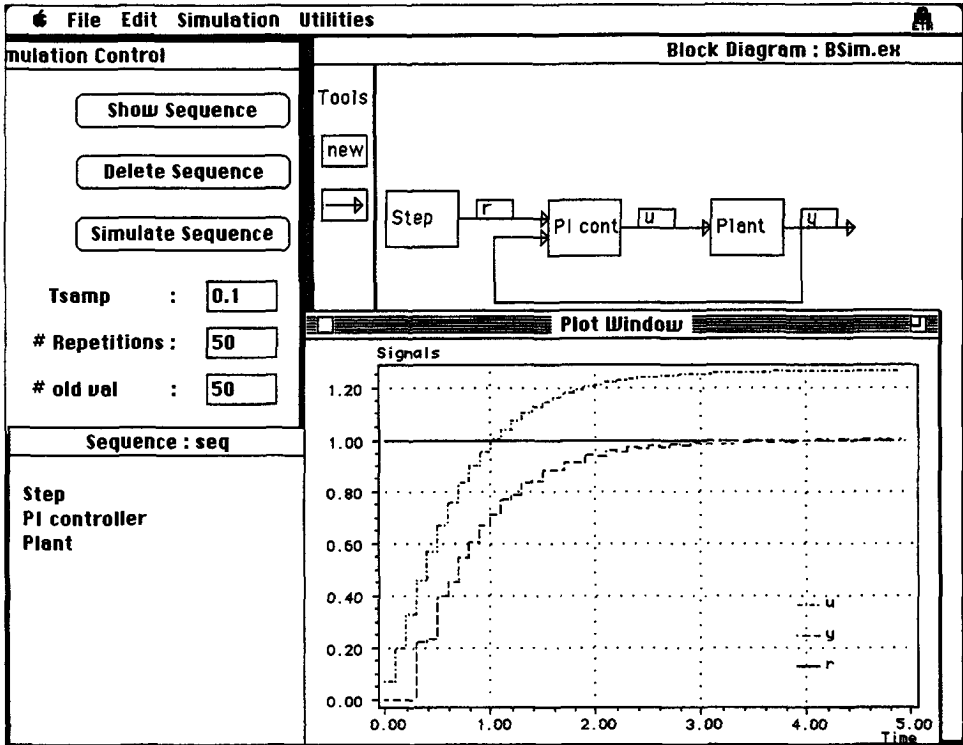


Figure 2.13: BlockSim example of a simple control system. It shows simulation control window, the block diagram, the simulation sequence and the plot window

Code extension: New blocks can be created only if the BlockSim source code is available. Like subclasses in object oriented frameworks, each block available has a number of necessary procedures which are called by the simulator. Any block added in Modula-2 requires a full rebuild of the package.

Similar tools: Education tools with similar features are rare.

Example: Fig. 2.13 shows a small example where a plant has been simulated with the corresponding PI-controller. The three blocks contained are evaluated in the order indicated by the simulation sequence (Fig. 2.13: window 'Sequence: seq'). The signals labelled in the block diagram are plotted in the plot window after the simulation.

Discussion: BlockSim was developed to teach the behavior of discrete systems. It is based on FPU, a package which provided the same functionality as BlockSim. Its user interface however was menu driven and not very friendly.

The capabilities of BlockSim are quite limited compared to commercial tools of that sort. However, it was one of the first visual block diagram editors available. We mention it because in addition to the block diagram defining the data flow, the control flow is specified by the user. Although this is a negative aspect when it comes to judging its

user friendliness, this feature is particularly interesting in our discussion of visual tools.

BlockSim has its range of applicability. If the control problem can be solved using only the built in blocks, this is done very quickly.

2.3.4. GenT

Type: GenT presents a fully graphical user interface to software development. Every aspect mentioned in Chapter 2.2 is displayed graphically: control and data flow of a program are presented in form of flow charts, data types are displayed as hierarchical iconic lists, and topological views are available in library windows. In addition to other general purpose languages, GenT contains a large set of structures necessary for real-time programming.

The GenT prototype used in this project was developed at the ABB corporate research center [40], [49].

Origin: General purpose software engineering tool. Its designated application domain is software development for automation.

Data flow: Similar to the block diagrams in Simulink and BlockSim, the data flow of a program (*i.e.* module or procedure) is drawn in form of a data flow graph. Like in the other data flow diagrams discussed earlier, the results of a given block are passed along its output connection to the inputs of the blocks it is linked to. The instance when this happens, however, is not specified in the data flow graph (*i.e.* the execution of a block does not depend on the availability of all data elements at its inputs). Similar to BlockSim, the control flow is specified in another diagram which will be discussed later. GenT blocks are either hierarchical procedures available in GenT

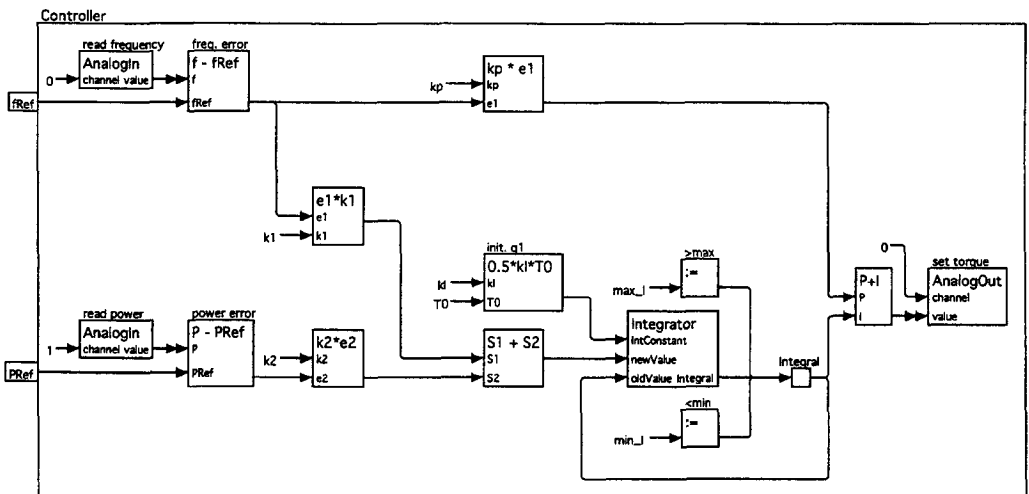


Figure 2.14: GenT data flow diagram of the primary controller of Example 6

libraries, textually programmed libraries, expressions or local variables (a construction normally not available in data flow only languages). Blocks may be connected without any topological restrictions, feedback is allowed. The connections are tested on type compatibility, if an output of a block is wired to an input of another block with different data type, an error is signalled immediately.

Control flow: A second diagram is needed to be set up for each program module, the control flow diagram. This diagram is basically a flowchart, which indicates on which conditions and in which sequence the operation blocks defined in the data flow graph are executed. Each operation block in the data flow has its corresponding description in the control flow, where it must appear at least once. In addition to these data manipulation statements, GenT provides a number of elements which define the control flow.

Among the basic statements are case structures, which branch on given conditions. In addition to plain flowcharts, GenT's case statements allow different conditions to be tested in the same call. By wiring the output of a control flow block to the input of a previously called block, GenT allows programming of loops.

In addition to these basic elements already known from flowcharts, GenT supports a few structures which particularly ease software design for real-time applications.

- Parallel flows: Different paths of the control flow may be defined to execute in parallel.

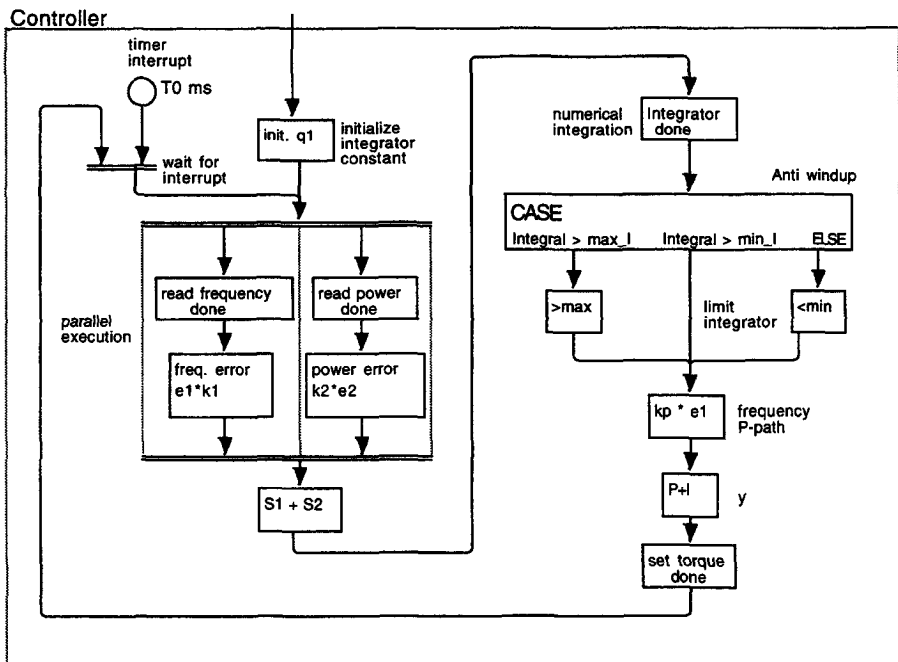


Figure 2.15: GenT control flow diagram of the primary controller example

- Interrupt handling: Interrupt signals may be used to trigger a program part previously assigned to wait for this signal.
- Synchronization: A synchronization mechanism is available to synchronize processes which are executing in parallel.

Data types: GenT allows all data types known from high level programming languages. They are specified in a dialog window and later displayed in a list form, which resembles the syntactically formatted type definitions in high level languages.

Data visualization: Unlike SystemSpecs or Simulink, GenT does not allow code execution directly from the diagram, it always requires code generation and compilation in order to run a program. Visualization tools for runtime data object instances have to be included in the generated code, they are not available directly from the tool itself.

Program animation: The same holds for program animation. Once the program is in a state to be run, it does no longer have any connection to GenT. Neither of them knows anything about the state of the other, therefore GenT is not capable of monitoring the behavior of the created program.

Hierarchical diagrams: Like in high level textual languages, libraries of user defined procedures and data types may be created. The GenT design direction is bottom up, single procedures are connected to form a more complex hierarchy. Parts of a complex procedure may not be selected and turned into subroutines as this is possible in SystemSpecs and Simulink.

Code analysis: GenT allows code analysis with respect to syntactical correctness. Type checks are done immediately as the user tries to improperly connect two elements which differ in data type. More inconsistencies (unwired blocks, undefined procedure code, etc.) are detected once the user chooses to either check module completeness or generates code.

Code generation: In order to run a GenT program, the user has to generate code. Pascal, C, and Modula-2 may be chosen as target languages. The generated code is executed independently from GenT after it is properly compiled and linked on the target system. This system need not necessarily be the one GenT is running on.

Code extension: Interface modules from other languages may be incorporated in GenT libraries. The external definitions are parsed and turned into graphical displays like the definitions specified from within GenT.

Real-time capability: Real-time execution of programs generated by GenT is the main emphasis of this package. Since high level language code is generated, the target system need not be known at the time the program is designed.

Similar tools: none known

Example: The primary controller presented in Example 6 has been implemented in GenT, its data flow graph is shown in Fig. 2.14. The controller is defined as a device, a stand-alone process, which communicates only via read-only variables (reference val-

ues fRef and PRef) with its environment. The diagram contains one variable (Integrator), several expressions and function calls. The similarity to the plain block diagram is obvious, only the integrator had to be replaced by a numerical algorithm.

Some of the control structures are contained in the control flow graph of the primary controller example (Fig. 2.15). The control loop is triggered by a timer interrupt, the control flow of the loop is therefore synchronized with the interrupt. The analog-in converters may require some delay, reading the current values from them is executed in parallel. Finally, some anti-windup measures are taken, the integrator is bounded depending on its value. These bounds are set in a CASE statement.

Discussion: The features offered by GenT make it a preferred tool for real-time implementations. Programming in different views needs some time to get used to. Once familiar with its concepts, GenT diagrams reveal some properties of the program being designed, which are otherwise not visible. Superfluous program parts or too complicated structures are easily seen on screen. If the GenT diagrams could also be used to run animated prototype implementations before code generation, this would certainly allow even more insight into program behavior.

Unfortunately, the unstable state of the GenT implementation did not allow its testing on a larger scale.

2.3.5. LabVIEW

Type: LabVIEW provides a data flow language (called 'G') to specify the functionality of a graphical user interface with a data flow graph. Each program (or procedure) called 'virtual instrument' (VI) consists of a user interface part (front panel) and of a diagram part [50].

Origin: Software instrumentation. The main user interface elements were designed to display oscilloscopes and other measurement equipment.

Data flow: Each LabVIEW virtual instrument is programmed by drawing its data flow graph. Elements of the graph are other VIs (hierarchical structures) or built-in operations like expressions, record or array element access. Similar to the high level Petri net, data tokens are passed between the data flow nodes. Different than in the Petri net, these tokens cannot flow in loops. A data flow path cannot be dependent on one of its results. This result is not available when the loop is executed for the first time. In the Petri net, this problem is solved by the initial marking (initial conditions), which provides the data elements to be taken for the first execution. G however introduces structured data flow (presented in [19]), which mixes plain data flow with control structures.

Control flow: Control flow in a G diagram is defined by special elements. Each of them is accessed like a data flow element, i.e., it is entered as soon as all its input connections are evaluated. Connections leaving the control flow element become active the

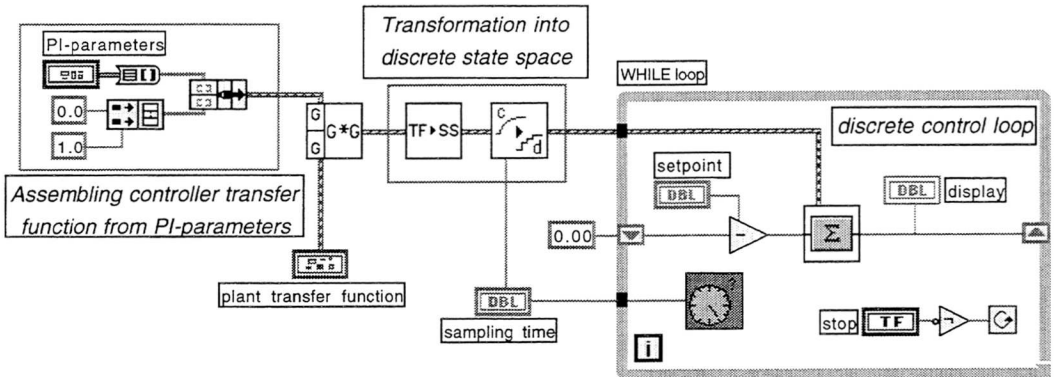



Figure 2.16: LabVIEW diagram of a discrete, timed simulation of a plant and a PI-controller

moment its execution is completed. Some of the G control flow elements are shown in Fig. 2.16: The control loop is implemented in a WHILE-loop.

The WHILE-loop also shows the implementation of feedback loops: shift registers () are used to store the state variables between the loop executions.

Data types: LabVIEW allows the definition of structured data types known from textual high-level programming languages. Arrays of any dimension and any element type are possible as well as record structures (clusters in G) of arbitrary complexity. Data types are specified visually by combining elements of the required types on the VI front panel. On the exemplary front panel shown in Fig. 2.17, the PI-controller parameters are collected in a single cluster.

Data visualization: Front panel data elements can be accessed throughout diagram execution, therefore providing animated data capabilities. A variety of interactive data

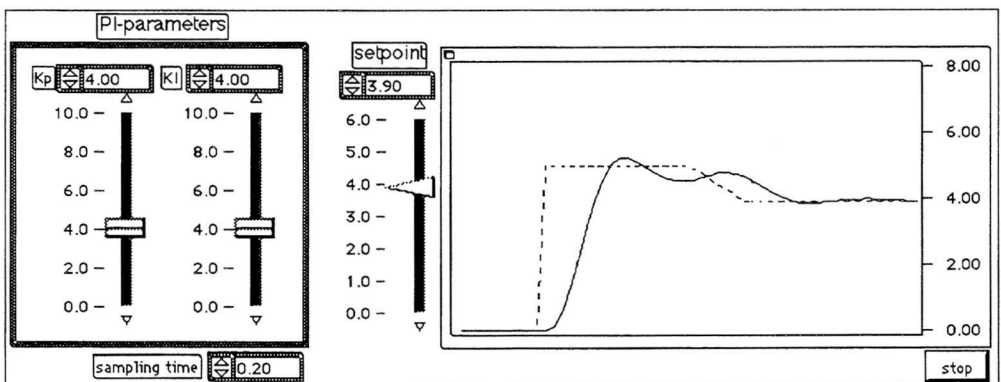


Figure 2.17: Front panel of the diagram shown in Fig. 2.16

manipulation elements are available, ranging from simple buttons to complex two-dimensional graphs. All these elements can be customized by replacing parts of them with pictures. This allows a user interface which very much visualizes the actions of the program. Similarities to processes known from the real world can be shown. In addition, data passed to a sub-VI can be watched by opening its own front panel.

Program animation: The execution of a VI diagram can be monitored by switching to highlighted execution mode. Data elements are then displayed as tokens similar to the Petri net animation. Their current values can be traced by assigning probes to data branches.

Hierarchical diagrams: In G there is no distinction between main program and subroutines. Any VI can be turned into a callable sub-VI. This is done by assigning some or all of the user interface elements on the front panel to be included in the VI's connector (the parameter list in a high level language). If the VI is later used as a sub-VI in another diagram, these connections need to be wired.

The sub-VI is still executable as top level VI. It can therefore still be operated from its own front panel, which allows development and debugging of smaller parts of a complex hierarchical VI setup.

Code analysis: LabVIEW provides on-line syntax checking of its diagrams. Missing or invalid connections are dashed. The error is explained upon request or before the VI is compiled.

Code generation: LabVIEW does not provide code generation in a high level programming language. However, its VIs are portable, and utilities are available on the designated target to create a stand-alone application from any diagram. These applications do not require the LabVIEW development system to be available. There also exists a signal processor library which allows execution of library routines on a signal processor.

Code extension: Program parts written in C can be incorporated and accessed from a diagram through a *code interface node*.

Real-time capability: The inherent parallelism of data flow graphs is supported. Timed execution is possible as well as analog input and output facilities. With the latest version, process synchronization by semaphores has been introduced.

Similar tools: There are several tools available today which provide graphical programming of user interface instrumentation (Workbench [55] *et al.*). None has yet reached the capabilities of LabVIEW.

Example: An example VI is shown in Fig. 2.16 (VI diagram) and Fig. 2.17 (VI front panel). It simulates a continuous plant with a PI-controller. Both systems are connected and converted into the discrete state space form. The control loop is timed appropriately and the simulation results of the closed loop system are displayed on

screen. The user can experiment on-line with different set-point changes; repeated execution of the VI also allows changes of the PI-controller parameters.

Discussion: LabVIEW has proven to be the most mature graphical programming package of the ones being discussed. A wide range of control problems was successfully solved. The very advanced user interface suggests its use as a teaching tool, which allows easy experimenting with a physical process. LabVIEW has been used in our Lab by several students in their semester projects, requiring only a short training time. These projects showed the advantageous use of LabVIEW in applications which are based on data flow principles.

2.4. Text or Pictures?

In addition to the examples mentioned in the previous chapters, several other problems from software design in automatic control were solved using one of the visual languages mentioned. After the discussion of those languages, we are now about to ask: Where did we get? Did we find the ultimate tool for software design in automatic control? Apart from the fact that the field of automatic control is too wide to be covered by one tool only, the answer is still: No. Did we gain anything by using visual languages at all? To this question, the answer is yes: there are situations where the use of a visual language is beneficial for the control engineer who is about to write software for the solution of a problem. Other situations are still better approached using traditional textual tools.

What are these situations? To get an answer to this question, have a brief look at Fig. 2.18a (try not yet to look at b): Only few readers probably recognize the function displayed by the LabVIEW program at first sight. In the formula node in Fig. 2.18b however, the function is easily identified: the two lines of C-like code calculate the roots of a quadratic equation. Its textual description is much closer to our idea of this

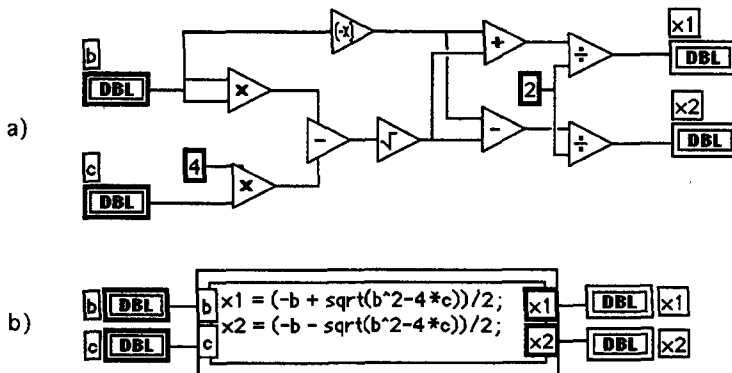


Figure 2.18: LabVIEW data flow diagram and formula node of the same equation

formula than the graphical data flow representation. The use of graphics in this example is questionable.

Example 7: Adaptive controllers in LabVIEW

During a semester project, students at the Automatic Control Lab tested the use of LabVIEW for the implementation of adaptive controllers [38]. They chose the self-tuning controller structure displayed in Fig. 2.20. The controller was a deadbeat controller, the identification was done using the recursive least squares algorithm which calculated estimated system parameters $\hat{\Theta}$ according to the following equations (for a thorough discussion of these or related equations please refer to Milek's overview in [28]. They are shown here to give an impression of formula visualization, understanding is not necessarily required):

$$\begin{aligned}\hat{\Theta}(k+1) &= \hat{\Theta}(k) + \gamma(k) [y(k+1) - \varphi'(k+1) \hat{\Theta}(k)] \\ \gamma(k) &= \frac{P(k) \varphi(k+1)}{\varphi'(k+1) P(k) \varphi(k+1) + 1} \\ P(k+1) &= \mu [I - \gamma(k) \varphi'(k+1)] P(k)\end{aligned}$$

These formulas were programmed in LabVIEW which resulted in the diagram shown in Fig. 2.19. It is quite obvious that the graphics do not contribute to the understanding of the algorithm.

The top-level control loop shown in Fig. 2.21 however shows its relation to the structure of the diagram in Fig. 2.20. The center of the diagram shows the control loop (plant, controller, identification, and controller design). Due to the LabVIEW implementation of feedback, the controller design is placed on the right of the identification instead of the left. In addition, the diagram contains other control blocks necessary for

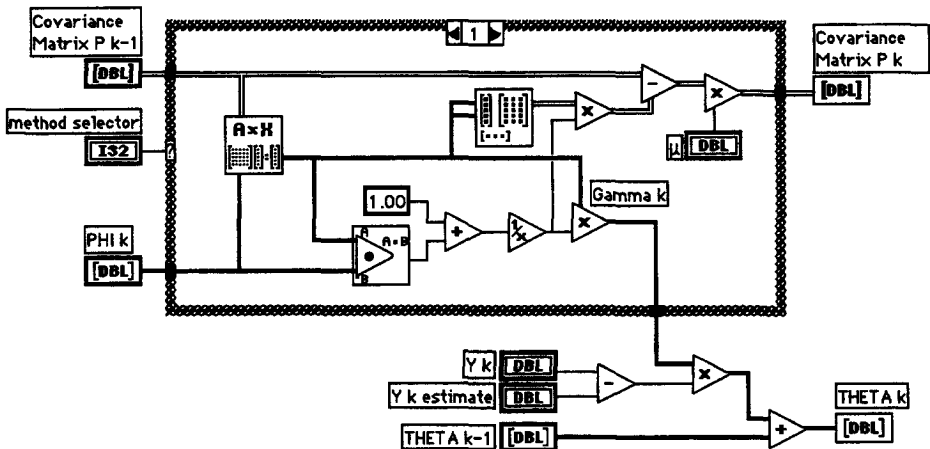


Figure 2.19: Recursive identification update programmed in LabVIEW

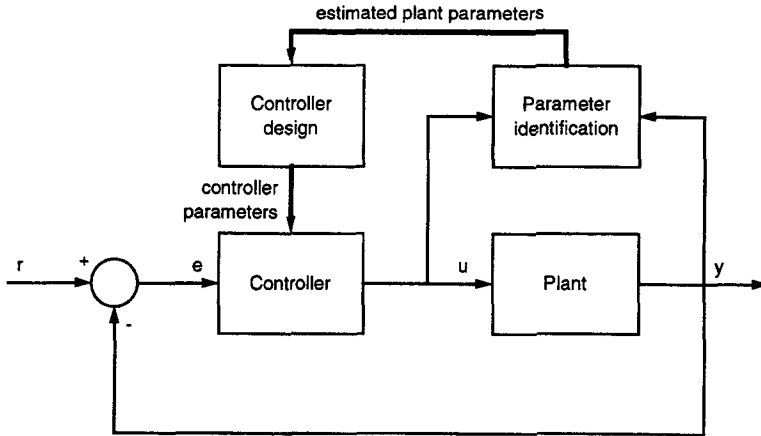


Figure 2.20: Adaptive controller scheme

the simulation. Limiter, noise generator, and random binary source are not shown in Fig. 2.20, they are shown, however, in the LabVIEW VI.

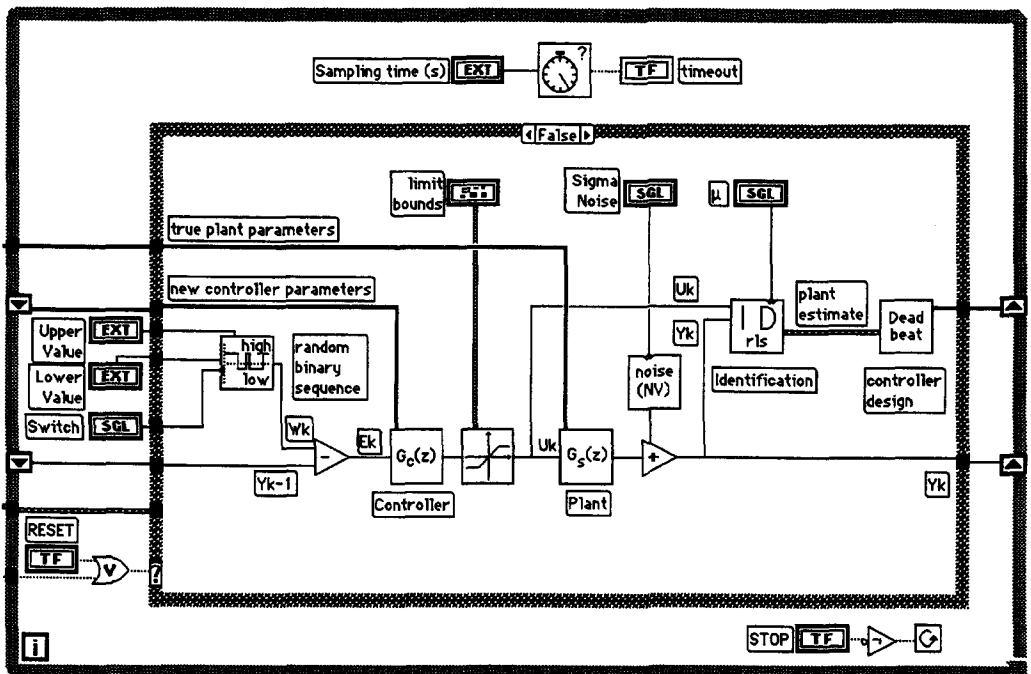


Figure 2.21: Adaptive controller loop programmed in LabVIEW

From the quadratic equation example and the adaptive controller implementation we get an indication about where to use textual program specifications, and where to switch to graphical programming languages.

Wherever our program already is defined by a text, conventional languages are best suited to solve the problem. Mathematical formulas mostly find their implementation in a direct conversion to textual program code, be it directly in one of the high level programming languages, or via specification in a mathematical package (Maple, Mathematica, Matlab). Since most controller design¹ algorithms today are expressed in pages of formulas, the numerical details of a 'control toolbox' are best implemented using one of these textual languages or mathematical packages. These packages contain large libraries of mathematical algorithms (matrix arithmetic, Fourier series, etc.), which eases algorithm implementation².

An obvious domain for the application of visual language is given when parts of the design was done using visual methods. If the controller design is done by drawing a block diagram, the use of a block diagram editor with simulation and code generation facilities is preferred.

Another example where an advantage of visual languages is shown is the adaptive control loop in Fig. 2.21. Data flow between complex sub-systems or control flow behavior in a larger context are easier perceived in a graphical form. At a higher level of abstraction, where function blocks containing implementation details and library functions are connected on screen, visual languages show program structure and other details of the implementation more clearly than textual languages do.

Between numerical details and overall system behavior there are more levels of abstraction, where the advantage of graphical or textual display needs to be judged anew with each implementation problem. In the optimal implementation of a complex application, both textual as well as graphical parts are probably used. The following discussion is intended to show some advantages of one or the other specification. It also indicates some weak points of visual languages which, once removed, could increase the usefulness of graphical programming.

2.4.1. Visual Languages Discussion

Development system: If we want to compare textual and visual languages, we cannot restrict the discussion to the language definition alone. It is also the qualities of the development system as a whole we need to look at. The independence from a fixed development system is one great advantage of textual languages. A textual program can be written in any text editor, on any computer system which need not necessarily be the target system. In contrast to this, visual languages are always closely linked (or bound) to the development system, and therefore to the supported computer hard-

-
1. Once again, we include all algorithms used in the design cycle (identification, design, etc.)
 2. Following this principle, the RASP control library written in FORTRAN was incorporated into LabVIEW in order to access stable numerical algorithms from within a visual language

ware. This development system not only contains the graphical editor, but also the code generator, and very often some sort of a user interface design tool.

This close linkage between the language and its development system does imply some of the advantages of visual languages mentioned below. However, if some feature is not available in a visual development tool, there is no alternative tool which does provide the desired feature. With the large number of available text editors on different platforms, there is a chance to find an alternate editor which suits the needs of the programmer.

Better feedback during programming: Since the language specifications are known to the visual development system, it can continuously monitor program correctness. Program development using graphics passes less inconsistent states than textual programming. A WHILE-loop entered in a LabVIEW diagram does not cause any inconsistencies. If the same loop is typed in a textual language (e.g. Pascal), the program is inconsistent from the 'W' at the beginning of the loop until the programmer enters the ';' after the final 'END'.

The remaining programming errors in the visual language can be flagged immediately. LabVIEW draws dashed connections in case of data type conflicts, GenT displays an alert when the faulty connection is wired.

Code analysis: The analysis features available in Specs even allow more error detection. By using Petri net analysis algorithms, some conclusions about program correctness are possible. By using graph analysis algorithms a graphical program can be searched for infinite loops or other faulty parts. Simulink detects algebraic loops in block diagrams, and the data flow topology in LabVIEW does not allow un-initialized variables to be used.

Debugging by program animation: Animated visual programs show their behavior more clearly than this is possible with textual debuggers. Token animation in LabVIEW and SystemSpecs give a good impression about program execution and misbehavior.

In addition to the flowing tokens, LabVIEW allows independent testing of any of its sub-VIs. Each VI (subroutine) provides a front panel (user interface). Values passed in the flow of program execution can be monitored on the open front panel, or the VI can be operated independently from its calling program and tested by operating it from its own front panel.

Layout: The layout of a textual program does not cause any problems. It is usually done by pretty-printer tools which add the necessary formatting characters. Tools of this sort are not available in any of the visual languages used.

The layout of a visual program can illustrate the program behavior. This is not guaranteed by the program just being graphical, this quality has to be designed by the programmer. This task is not trivial. According to our experience, most students are aesthetically challenged using graphical programming tools. All the advantages of visual languages are lost if a program is represented by a screen full of badly placed

objects and wires. An automatic layout and routing facility could help much in order to improve this.

Subroutines, graphical and textual: The requirement for hierarchical diagrams in visual languages is obvious. No serious program development tool can be used in larger projects without libraries of re-usable code. The tools presented all support this feature.

In addition to graphical libraries, the possibility to incorporate textually programmed code extensions should also be provided. We have seen that most numerical algorithms are still more clearly expressed in conventional programming languages. By allowing the inclusion of their textual implementation, we are free to choose the appropriate tool for each part of our project.

Code maintenance: Although the visual languages discussed provide a highly interactive programming environment, code maintenance is sometimes more difficult than in textual environments. A cross reference tool which is available for most textual languages was not found in the graphical implementations. Operations which are very simple in text processors impose difficulties in a graphical editor. A global find or replace operation is not provided. If the number of parameters of a procedure changes, each use of the function needs to be traced 'by hand', the replacement is done by deleting the old function call and connecting the new one. Although LabVIEW provides a 'replace' operation, automatic connections are rarely correct after the replacement. Changes in a complex diagram always need a rearrangement of the diagram, a routing tool mentioned earlier would save much time otherwise used for moving parts of the diagram and repositioning wires.

2.4.2. Summary

Both textual and visual languages have their strong and weak points. Still, visual languages are limited to a narrow range and size of applications, mostly within their native domain. If we take into account the long history of the development of textual languages, there are more general visual development tools to be expected. Together with graph analysis algorithms, graphically developed software could be tested for completeness or topological errors, which increases software stability.

In our opinion, working with visual languages shows the edge of a powerful approach towards computer programming, which is definitely worth to be watched in the years to come.

Leer - Vide - Empty

Leporello

Visual Representation of the Design Process

3

In the previous chapters, the targets of our aim to visualize software were single algorithms. The adaptive controller example (Example 7 on page 49) contained visually programmed identification and controller design algorithms, the interactive design in the bode diagram (Example 3 on page 22) showed the advantage of an animated algorithm user interface, and block diagrams in Simulink (Fig. 2.12) provided a good example of how visual programming has always been present in modelling and simulation.

However, programming algorithms, whether visually or textually, is not a very frequent task in a control engineer's work. Control systems design mainly requires choosing an algorithm on a previously selected set of data, executing it and afterwards judging the results. The examples from Chapters 2.3.4 and 2.3.5 only contained the 'executing' part of that work. Selecting an algorithm or data was not supported at all. In fact, these self-contained algorithms left us in a situation very common a decade ago: the whole design cycle had to be programmed and compiled, data were passed on files, and interactive experimenting with data and algorithms was awesome, if not impossible. If we want to do a full design cycle in LabVIEW alone for example, a feasible way is to add file handling capabilities to each algorithm in the library, and then pass intermediate results on files. Direct passing of parameters between instruments requires additional programming, which results in very inflexible solutions.

We therefore need a tool to combine all these visually programmed (and other) algorithms to allow their convenient selection and application to well-organized data sets. Staying within the concepts of visual programming we look for a tool which satisfies these requirements in a graphical user interface.

This is where we started the Leporello project. An overview over different aspects of the project has been presented in [10], [11] and [21].

3.1. The Leporello Project

It is not only the visual examples we presented which have their drawbacks when it comes to full design support, it is also commercial packages. As an example, we take a look at the weaker points of Matlab, a package which due to its widespread use may be regarded as a de facto standard in computer aided control systems design.

Matlab's popularity is mainly based on its following advantages:

- It contains a large number of algorithms. There are many toolboxes that combine numerical methods from specific fields of electrical engineering. Its particular strengths besides linear algebra lie in signal processing and control systems design.
- All those algorithms can be accessed interactively by entering commands. This makes the package very flexible. Experimenting with different algorithms and parameter sets is very easy.
- Algorithms developed in house can be added easily by writing scripts in the same language the commands are entered.

But since the average user is far from being perfect, some of these advantages may in some cases turn into disadvantages:

- The command line interface also provides a very powerful tool for entering erroneous commands. The user has to remember the exact spelling of algorithm and variable names even to get help about some topics. Misspellings or typing errors result in cryptic error codes, and parameter dimensions are very easily mixed up (row vs. column vector, ordering of polynomial coefficients, etc.).
- Finding the appropriate algorithm which solves a given problem is not very easy because of their large number or their peculiar spelling. Few users know all the details about available features. Algorithms are sorted by name and toolbox, additional information is only available in help texts. Again, the user has to know exactly how to spell a command and how to choose its parameters to execute it.
- As mentioned with the positive points, experimenting with various algorithms and parameter sets is very easy. It is also very easy to completely mix up the results. If the user forgets names of variables, data may be lost or mixed up and misinterpreted. The user's mind is the only place where additional information about data items (algorithm and parameters which produced them) is stored (One exception is the theta-format used in the identification toolbox by L. Ljung. It stores the name of the algorithm which produced it. Further history is not recorded).

In the Leporello package, structuring of data and algorithms is a key feature. The concepts of object oriented programming are providing powerful features for the implementation of the necessary constructions. Special emphasis was put on the following topics (the numbers correspond to the numbered labels in Fig. 3.1):

1. Algorithm structures: Algorithms include the information about their required input data types (*i.e.* whether the algorithm acts on a state space system description or on a discrete transfer function), which parts of the input data are used for the cal-

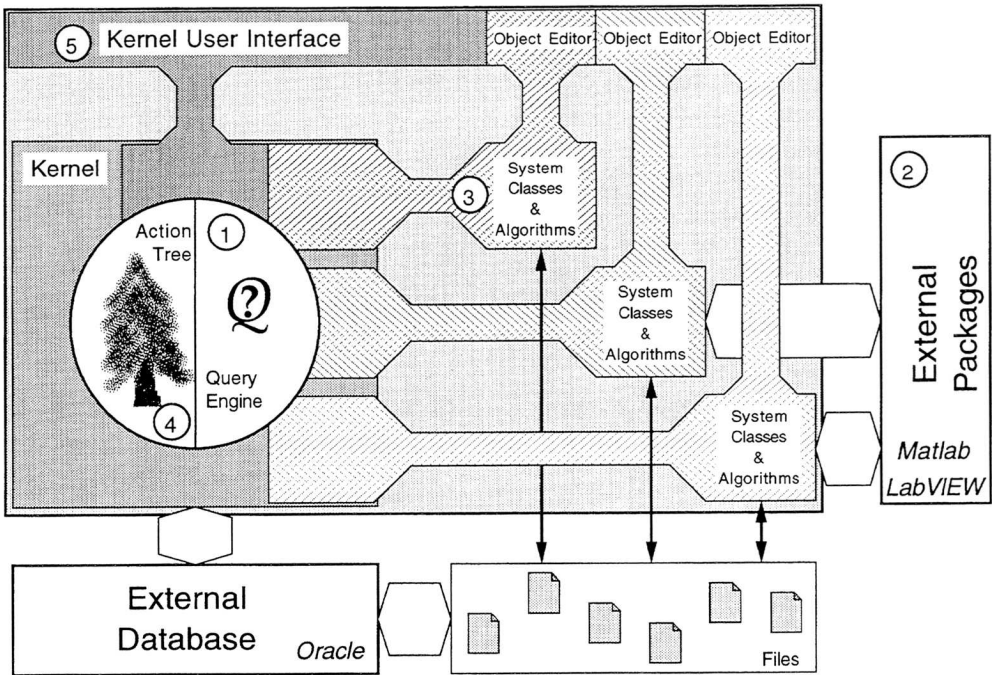


Figure 3.1: Leporello package overview

culations, number and dimensioning of the parameters, and the type of output data they produce. This allows structured algorithm selection and execution.

2. External algorithms: To access the large number of algorithms available in software tools today, Leporello is equipped with communication facilities to execute algorithms externally and transfer parameters and results. Currently, Matlab and LabVIEW are supported as external packages.
3. Object oriented design: External algorithms as well as control data items are mapped onto Leporello objects. These objects are then organized by the kernel and can be accessed through their individual user interfaces. By combining simple data items (matrices, vectors) to control data objects we achieve a higher level of data abstraction.
4. Design history: The history of each data object is kept not only for information purposes, but also to repeat previous work, either on different sets of data, or on the same set with changed parameters. This allows more structured experimenting with modified algorithm settings. Results of different algorithms or experiments can easily be compared to each other.
5. Graphical user interface: The use of a fully graphical user interface reduces the possibility of entering erroneous data. The organization and the structure of algorithms prevent their application on improperly shaped data sets. Their results are always stored in a consistent high level data object.

In the following chapters, we will first give an introduction to the concepts of Leporello. We will then present the main features along with some details of the implementation and some objects involved¹. User interface examples are provided where necessary.

3.2. The Design Cycle

To be able to ease data management and algorithm selection in the design process, we will first have a closer look at this process. We will look for data and control structures which are obvious candidates for visualization.

Control systems design is performed in a design cycle like many other designs in engineering. We always have what we call a design target, the object which is to be designed. In the case of CACSD these are control systems or their elements (signals, systems, etc.). Some work is done on selected data elements, the results are judged by how they match the design goals, and then either some previous step is repeated using different methods or better parameters, or the results are used for further processing. Almost everybody who writes about CACSD (e.g. [34], [37]) shows a diagram of this design cycle. We follow this tradition in Fig. 3.2.

The loop starts with an experiment executed on a physical plant to obtain some measurement data. Modelling and identification algorithms use these to produce a numerical model. We then run design algorithms to get a controller, which is later implemented in some microprocessor system first to be simulated, and then to be tested on the plant. Each step produces a set of control data, signals, systems, or any combination thereof. These are rated, rejected, or stored to be used for further calculations.

3.2.1. Action Tree Basics

The amount of data in the cycle however is not limited to one item per data node shown in Fig. 3.2. Results are not always discarded, but very often kept for documentation, to be compared to other items produced by other algorithms, or just to leave some design options. In Leporello, we will therefore store all intermediate data in *data nodes*. Each of these nodes may be regarded as the root of a tree-like structure. Each time the user returns to this node and applies another algorithm to it, a new branch is created with the new results as its leaf. This may again be regarded as root of further calculations. The loop in Fig. 3.2 is hence decomposed into a tree shown in Fig. 3.3. Data objects are now tree nodes, the algorithms applied are the connecting branches.

1. Certain aspects of Leporello, especially the basic data structures will be mentioned very briefly. These parts have been developed by P. Kolb and are discussed in greater extent in his thesis [22] (which unfortunately is in German) or in short in [21].

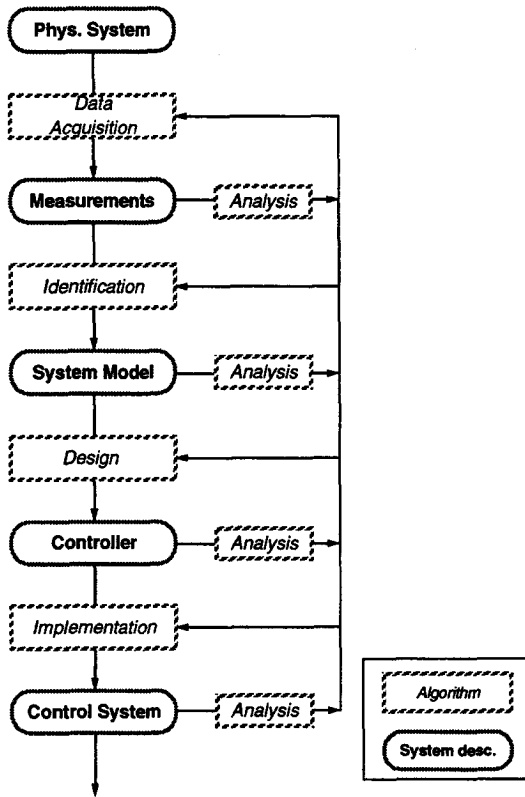


Figure 3.2: Control systems design cycle

Since this tree is an image of the actions taken during the design process, we call it the *action tree*.

The action tree structure holds control objects in its nodes, the information about the algorithms which produced them could be stored in corresponding branch objects. This algorithm information not only contains the name of the algorithm, but also all the information necessary to reproduce the results. Keeping in mind that a visual representation should give an impression about the displayed object, we did not like the idea that all this algorithm data were contained in the branch line only. We therefore introduced *algorithm nodes*. But not only user interface considerations led to this node definition: there are situations when an algorithm node can produce several result data nodes¹. This is more clearly visualized by an algorithm node which has several children than with a branch leading to different data nodes, or with different branches bearing the same algorithm. Since an algorithm cannot result in or be applied to another algorithm and a control data object cannot produce another data object with-

1. The motivation for this approach will be shown later, the reader may keep the picture of algorithm branches in mind to get an idea of the data relations in the tree.

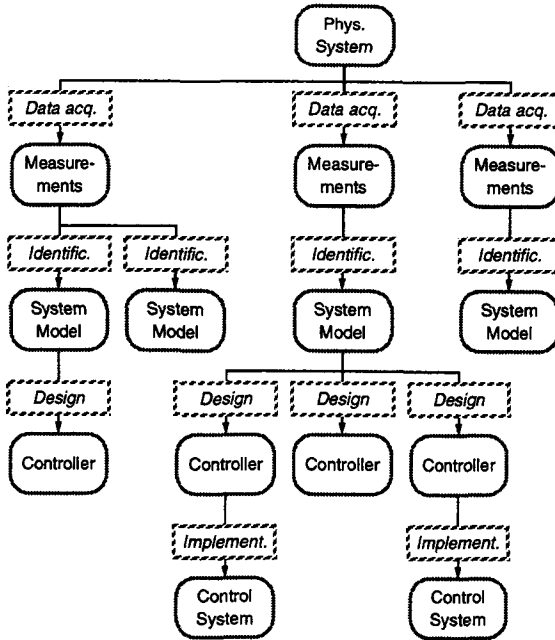


Figure 3.3: Action tree draft

out a transforming algorithm; node types on each tree layer are alternating. A layer containing data nodes is always followed by one containing algorithm nodes.

3.2.2. Tree-structured Data Bases

Storing data in a tree-structured database is not a new idea. Any process which involves returning to previous results and applying changes may be organized in a tree.

Kierulf [18] uses the idea in his database for two dimensional games (chess, Othello, etc.). He stores game variations in trees, where the moves form the branches and the board situations the nodes. He can return to previous situations (take back moves), look for alternatives or replay famous games.

Apple's Macintosh Programmer's Workshop [52] (and other document management tools) contains a database for source code revisions, the *Projector Tool*. Each file has its own revision-tree. Contrary to Kierulf, not every action is stored, but only selected stages of the document designated by the user.

The Andecs package by Gröbel *et. al.* [14], [17] uses a structure similar to the one proposed by Kierulf for their control engineering database. Their 'moves' are not general actions taken, but are formed by three design strata. The resulting tree therefore has only three levels: dynamics model, performance measures, and design.

In these examples of tree structures, nodes mainly contain what was produced. In Leporello we also store how it was produced; together with the data it was applied to and the results, we store all the information necessary to execute an algorithm. This includes its name, parameters, and details on how it can be accessed or executed externally. In addition to just being a database of the design process, the action tree thus becomes a tool to manage algorithm selection and execution. Algorithms can be started and monitored, and by reapplying parts of the tree to other nodes, parameter variations can easily be applied. The tree is created directly on screen by selecting nodes and algorithms to be executed.

Another difference to other tree-structured databases is that they all have their tailored user interfaces and use the tree structure only as a database. It is then accessed like any other database in form of lists and text displays. In Leporello, the picture of the action tree constitutes its main user interface, where the user decides which algorithm to apply next and monitors its results. The display is updated on-line, completed algorithms immediately display new nodes containing their results.

In the following chapters we will develop the object and user interface structure of the tree. Before we continue to the description of the tree elements and the actions on them, we will look for an object oriented data description for control data and algorithms.

3.2.3. Graph Theory

But let us first take a short excursion into graph theory to see that the design process described above does comply to the definitions for tree-shaped graphs.

First, we see that the graph describing the design process is directed: Branches connect either source data nodes with the algorithm node applied to it, or an algorithm node with its resulting data node.

In the IEEE standards dictionary [15] we find the definition of a directed tree (*arborescence*):

An abstract hierarchical structure consisting of nodes connected by branches, in which:

- (a) each branch connects one node to a directly subsidiary node,*
and
- (b) there is a unique node called the root which is not subsidiary to any other node,*
and
- (c) every node besides the root is directly subsidiary to exactly one other node.*

In [42] we find the same requirements in a more analytic description, including the corresponding graph theoretic definitions and proofs. In the following discussion however, we will keep with the more linguistic explanations cited above.

To show that the action tree is in fact a directed tree according to graph theoretic definitions, we need to show three parts:

- i) there is exactly one root node
 - ii) every node has exactly one parent, which is divided into
 - iiia) every data node has exactly one parent
 - iiib) every algorithm node has exactly one parent.
- i) may be assumed without loss of generality. We define the root node to be an algorithm node which produces data objects to which the design process is applied. An algorithm of this sort may be reading from a data file or performing a measurement experiment which returns its results in data vectors. We only allow one generating root node in a tree document. Using data sets from different generating algorithms (on-line measurements and data read from file) requires two separate trees.
- iiia) is also quite obvious. A data item must be the result of exactly one algorithm. The cases where two algorithms produce exactly the same result (symbolically as well as numerically) are very rare. If they exist, we will handle this situation as if the results were different and leave the fact, that they are indeed identical to be remembered by the user¹.
- iiib) is much trickier as it limits the available algorithms to those, who act on one data item alone. This may sound too strong a limitation, but remember that a data item is a highly abstract control data object, *i.e.*, a system or a signal on which the design is focused. Although we restrict algorithms to having one of these items as their input (which may contain any number of sub-objects and parameters), we do allow additional parameters to control algorithm execution which are not taken from the tree, therefore not adding branches, which would destroy the tree structure.
- However, incorporating algorithms which combine two data objects (*e.g.* connect them in a block diagram) is not allowed. We will later show (Chapter 4.6), how these connection algorithms fit into the action tree concepts. For now we do limit the algorithms available for use in the action tree to the ones which act on exactly one source data item. We will see that their number is significant enough to justify the concepts presented in the chapters to come.

3.3. Basic Data Structures in Leporello

As stated in Chapter 3.1, one of the drawbacks of CACSD packages like Matlab is their lack of data structures more complex than two-dimensional matrices. Information about a system given in state space representation is contained in four matrices with interrelated dimensions. The user has no means, other than choosing appropriate

1. Since we want to ease algorithm selection, it is not reasonable to supply two algorithms which do exactly the same thing.

names, to indicate that these four matrices are part of a common structure. It is also impossible to pass a complete system to an algorithm, all four matrices have to be supplied individually. This limitation dates from the time when FORTRAN was still the number one language for scientific computing. In his ADA-based implementation of IMPACT, Rinvall introduces structured data types [37]. He composes a state space system from single matrices and then passes this one system variable to an algorithm. With today's tools and languages available, there is no reason not to follow Rinvall's ideas.

3.3.1. The Object Oriented Approach

Since the times when ADA and Modula-2 became popular, another concept has entered software engineering and has surpassed a popularity reached for example by fuzzy systems in automatic control: *Object oriented programming* (OOP). Although the main ideas of OOP are widely known, we will summarize some key features.

Contrary to a conventional program, a program developed using object oriented concepts consists of interconnected software objects rather than sequential code. The elements of these objects are fields which store data, and methods which operate on these data. The declaration (similar to the type declaration in Pascal or Modula-2) is called the *class*, the instance of a class is called an *object* of this class (Pascal variables).

In a proper object oriented environment, the data fields are referenced directly only for reading, writing is permitted through methods. The program execution is then controlled by messages exchanged between objects (*i.e.* method calls). These messages are initiated by user interaction or other asynchronous events, or as a reaction to other messages.

A very important feature of an object oriented language is inheritance. It allows subclassing of existing object classes. The subclass then inherits the fields and methods of the basic class. It can be extended by adding more fields and methods, or by overwriting methods defined in the superclass, therefore adding special functionality to previously declared tasks. When using a specialized object, its exact composition does not have to be known. A client object just calls the appropriate method contained in the base class, and the object itself knows, how to handle this call, whether it can satisfy the request itself or whether the appropriate method of one of its ancestor classes is the one to be called.

The following object definitions (systems, parameters and algorithms) are only described briefly since these parts are extensively discussed by Kolb in [22].

3.3.2. Control Data Objects

Returning to our initial problem of designing data structures for control systems and algorithms, we see that system descriptions are obvious candidates for an object oriented implementation. If we look at the elements used in a block diagram, we see a set of interconnected system boxes. Each box takes its inputs, does some calculations and

produces the outputs (see Chapter 2.2.2). Details of these calculations may be influenced by parameters. If we apply object oriented concepts to the design of data structures for systems, we get the following structure:

<i>Connections</i>	A system has inputs and outputs, which may also be objects of some 'connection'-class
<i>Parameters</i>	It has a list of parameters which can be accessed by the internal calculations as well as by the user who wants to influence system behavior.
<i>Internal behavior</i>	The calculations which perform the transformation of the inputs to the outputs. This part is subclassed by specialized objects to define systems of various kinds (nonlinear, linear, parametric, nonparametric, etc.)

The advantages and the popularity of object oriented programming have led to several papers describing system objects (e.g. [3]). Some of them are discussed by Maciejowski in [25]. All solutions proposed share this structure of connections, internal representation and parameters¹. In addition, a system object contains elements which ease its administration (name, date created, etc.) or other tasks it is used in.

Omola, presented in [27] by Matsson, puts its emphasis on modelling and simulation. The features shown in the paper guarantee a consistent connection of sub-systems. Connection items therefore include information about SI-units and type of connection (electrical, pipe, etc.).

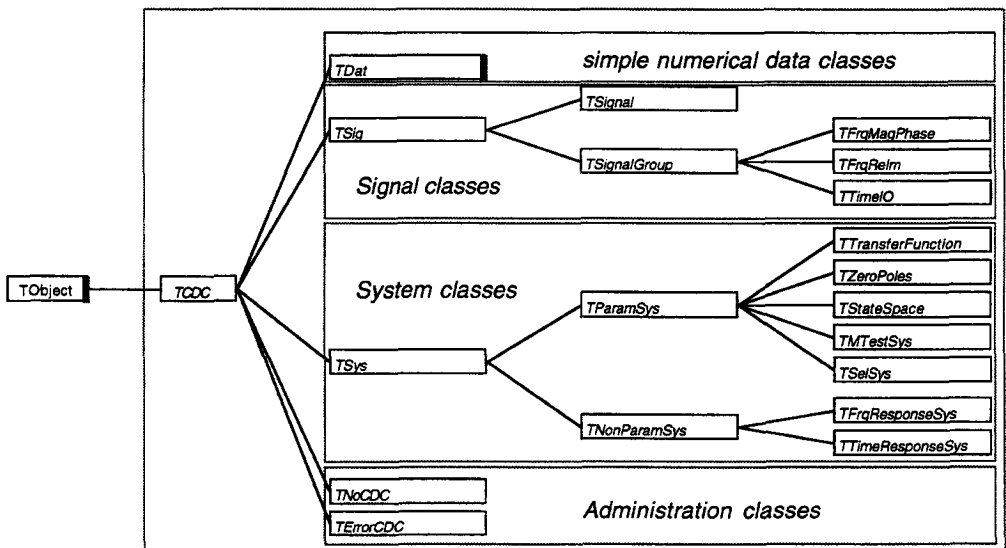


Figure 3.4: Control data class tree

1. One day we might even see a standard for object oriented CACSD structures. It is currently being discussed

The system representation chosen in Leporello is very similar to the one presented by Matsson. However, since our focus is not only on building interconnected systems but mainly on manipulating data items, we do not limit control objects just to systems. Data items required during the control cycle are also signals (measurements and simulation results in the time domain, spectra in the frequency domain) and in some cases numerical elements (polynomials, matrices, etc.). These structures are added to the tree of control data classes (Fig. 3.4). This tree has its root in the abstract *control data class* (CDC), which does not contain any structural information. Since we allow any kind of control elements in the action tree, a tree node is designed to hold basic CDC objects. Due to object oriented concepts, it is then allowed to contain any subclass of the CDC.¹

Control data class²

TCDC = OBJECT(TObject)

Control data objects used in Leporello.

Fields	
fName	Name given by the user
fUsedAs	The purpose the object is used for
fFixedPropList	List of class properties (i.e. linear, parametric, etc.)
fDynamicPropList	List of object properties and numerical attributes (stable, controllable, system order, etc.) depending on parameter values
fRatingList	
fComponentList	List of elements of this object (sub-systems, etc.)
fParameterList	List of parameters and numerical data
Methods	
GetName	Retrieve information from the object
GetParameterList	
HasDynProperty	
GetInfo	Display information window

According to object oriented concepts, these fields are inherited by systems and signals. In addition, a system receives a list of inputs and outputs:

1. Please note that a CDC signifies the control data class, whereas a CDC object denotes an instance of that class. For algorithm execution, we check if the algorithm can be applied to a given data class, but the numerical values then taken for execution are taken from the CDC object of this data class.
2. For the format of class definitions please refer to Appendix C.

System

TSys = OBJECT(TObject)

System class

Fields	
fInputsOutputs	List of connections
Methods	
AddNewInput AddNewOutput	Add inputs or outputs to the system

3.3.3. Parameter Lists

One important part of a CDC object and one of the central classes in Leporello are parameters and parameter lists. A parameter is a numerical element (scalar, vector, matrix) which determines numerical values specific to one object instance.

All parameters of a CDC object are collected in a parameter list. The parameter list of a system in state space representation contains its four matrices, if it is discrete, it also contains a scalar indicating the sampling time. Each parameter in the list is identified by its name. This name is used when the parameter list is displayed on screen, or when an algorithm looks for the parts of a CDC object required for its execution (this will be shown in the next section). Its close relation to numerical control data classes suggests making parameters a technical subclass¹ of the numerical data CDC. Parameter subclasses include all forms of numerical data, from integer, real or complex scalars to matrices of the respective numerical formats. The parameter class tree is shown in Fig. 3.5.

We will later see that parameter lists are also essential to the algorithm execution mechanism.

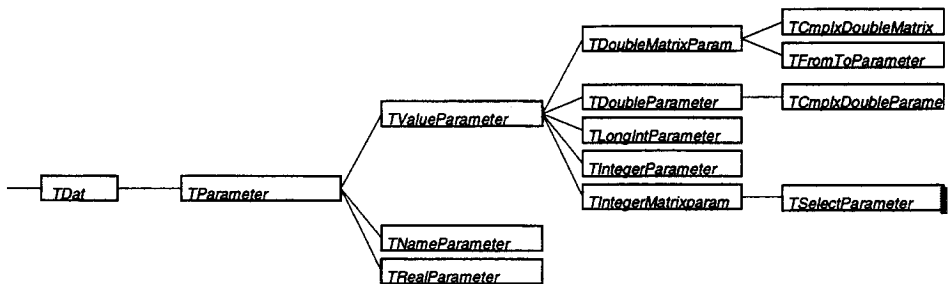


Figure 3.5: CDC and algorithm parameter class tree

1. By the term *technical subclass* we want to indicate, that the parameter was made a CDC subclass only because of the convenient inheritance, not because it can ever be regarded as a CDC itself. It is therefore not included in Fig. 3.4.

3.3.4. Algorithms

Now that we have defined the control data classes we can take a second look at algorithms. When we laid out the structure of the action tree, we met two kinds of algorithms. Their individual behavior is shown in Fig. 3.6.

CDC algorithms The algorithms which form the action tree. They take a CDC object and a parameter list to create another CDC object. The object produced is different from the source object.

Rating algorithms In the design cycle description in Fig. 3.2 these algorithms were contained in the 'analysis' boxes of the cycle. They perform some operations on a CDC object to derive some properties or rating figures. The results of these algorithms are attributes of the source object and are stored in its object properties list.

Since algorithms, as we define them, are closely linked to the source CDC structure, object oriented principles suggest to define them as methods of the CDC.

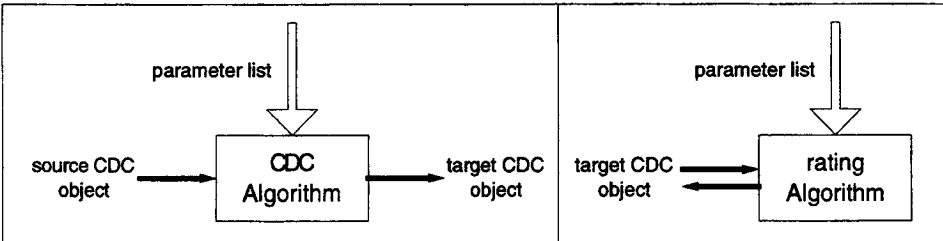


Figure 3.6: Algorithm structures: a) CDC modification and b) rating algorithm

We remember: methods access or modify an object of the class they were defined for. If some functionality is achieved differently in a subclass, the method is overridden; if the subclass does not require a different implementation, it is inherited. Algorithm definitions could benefit from this feature: Each system class for example could have a method to test a system's stability. The user does not need to know which algorithms test stability, the appropriate one is chosen automatically.

There is one severe drawback of this implementation: a method call is determined before compilation. This does not facilitate package extension. New algorithms are easily added to increase the functionality of Matlab, and a package which has to be recompiled to add a new algorithm or to modify an existing one is not considered very useful. Since we are providing external package support, addition of new algorithms should be as easy as in their native packages. This permits skilled users who develop their own algorithms to implement them in some external package and have them available in Leporello with little extra effort.

To allow the algorithm query facility to display a list of algorithms which can be applied to the CDC object selected by the user, it needs the possibility to search for algorithms. If we ask which identification algorithm can be applied to a series of mea-

surements and returns a state space model, searching object classes for appropriate methods is not practicable.

We therefore decide to define algorithm access classes independent from the CDC they are applied to. We distinguish between an *algorithm information object* (in Leporello TAlgHead class: objects which store all the information about an algorithm, also referenced to as the *algorithm head*) and the *algorithm execution object* (TAlg, an object which executes and stores an applied algorithm, *i.e.*, source and target CDC objects, parameters applied etc.). Each available algorithm is listed in one algorithm information object. If an algorithm is applied to several CDC objects, there exist multiple algorithm execution objects, each referring to the same algorithm information object (*i.e.* the same algorithm) but with distinguished parameter lists, source CDC objects and result objects.

Algorithm information objects contain information which is necessary to select an algorithm. Properties of the required source CDC are contained as well as properties of the algorithm itself, its purpose and its classification based on control theoretic viewpoints. Its GenAlgorithm method creates an algorithm executor which is capable of controlling the execution of this algorithm.

Algorithm execution objects contain the information necessary to successfully execute the algorithm. They contain the algorithm's source CDC, the algorithm parameters and the information how the algorithm is executed. After algorithm completion, the resulting CDC is also available in the algorithm executor. In addition to its capability to run an algorithm, the algorithm executor can also be used as an algorithm history object, to store all the information about one specific execution.

Algorithm information object

TAlgHead = OBJECT (TObject)

Object which contains all the information necessary to execute an algorithm.

Fields	
fAlgClassName	Name of the algorithm
fDemandsPropList	Class and required properties of the source CDC object
fProducesPropList	Class and produced properties of the target CDC object
fAlgParamTypes	List of parameters the algorithm requests
Methods	
GetInfo	Displays a window showing the algorithm information
IsExecutable	Returns true if this algorithm can be applied to the supplied CDC object
GenAlgorithm	Creates an algorithm executor object.

The algorithm information of externally accessed algorithms include some additional information about the package they use.

External algorithm information

TextAlgHead = OBJECT (TAlgHead)

Information about externally accessed algorithms.

Fields	
fAppSig	Signature of the package used
fCommandName	Name of the command transmitted to the external package

Algorithm execution object

TAlg = OBJECT (TObject)

Executable algorithm instance object.

Fields	
fAlgHead	Reference to the algorithm information object
fDemandsCDC	Source CDC object the algorithm is applied to
fProducesCDC	Target CDC object which contains the algorithm results
fAlgParams	List of parameters
Methods	
GetInfo	Displays a window showing the algorithm information and parameters
SetDemandsCDC SetParameterList TakeAwayProducesCDC	Access methods to set and retrieve the required input and output objects
ModalAskParameters	Displays a dialog box to ask the parameters
Execute	Executes the algorithm on the current source CDC object using the parameters available.

The external algorithm executor class contains an override of the Execute method which controls the transmission of the parameters and the correct classification and ordering of the results.

3.3.5. Algorithm Execution

Once an algorithm execution object is created and all necessary fields are set (source CDC object and parameter list), an empty target CDC object is created. The algorithm executor then collects the necessary parameters from the source CDC and the parame-

ter list and calls the algorithm code. This is mostly an external Matlab or LabVIEW function, but may also be an internal piece of code. The target CDC parameters are then filled with the algorithm results, and if necessary, with some parameters of the input CDC and the algorithm parameters. If the algorithm does not return a CDC object but a property or a rating, no target CDC object is produced and the result is appended to the appropriate list of the source CDC object.

Communicating with an external package imposes special problems. The server application may be located on the same computer as the Leporello client. Leporello therefore has to release the processor in order to allow processing in the other application. All externally accessed algorithms therefore must be executed asynchronously. We will later see that these restrictions require some special programming techniques. Certain tasks cannot simply be executed in a statement sequence, they must be scheduled.

This is a very short summary of the algorithm execution mechanism in Leporello. For a complete description please refer to [22].

The Action Tree: Functionality and Object Definition

4

With the definition of the objects from Chapter 3.3 we have reached a stage where we can reproduce the basic functionality of Matlab (execute an algorithm on some data). We can now start to describe the key features specific to Leporello. We will concentrate on the tree related elements: data storage and algorithm execution in the tree. Other elements (algorithm query and extensibility) are described, once again, by Kolb in [22].

The tasks supported when working with the tree not only allow storing the design history. Algorithms appearing in the history may be executed repeatedly to observe the influence of parameter variations on the results, or to apply a successful algorithm to another object in the tree. These tasks need to be organized and up to a certain degree automated.

Tasks explained in the forthcoming chapters are as follows:

- Algorithm execution
 - select and execute an algorithm using CDC and algorithm nodes;
 - execute an algorithm node on different CDC nodes;
 - reapply an algorithm repeatedly to the same CDC object with different parameters and create a multi-object node;
 - reapply an algorithm to a multi-object node.
- Managing multi-object nodes (nodes which contain a list of CDC objects resulting from the same algorithm):
 - create subset nodes (split a multi-object node into several nodes);
 - merge subset node (merge two previously split nodes).
- Managing tree complexity:
 - copy and paste operations;
 - delete objects;
 - mark deleted nodes

- Automatic subtree execution:
 - make a subtree consistent (automatic re-execution of partial trees);
 - hierarchical algorithm (create a new algorithm from a branch of algorithms).
- Special automated tasks:
 - parameter sweep (automated parameter variations);
 - optimization (optimize parts of the tree).
- Hierarchical systems

Each task is described in three parts, each focused on of the following aspects:

<i>Task</i>	A short description of the task. The purpose of this feature during control systems design is explained.
<i>Objects</i>	To implement the feature presented the tree object structure possibly has to be extended. Additional or modified objects are introduced where the newly added feature shows the purpose or the use of the object best.
<i>Display</i>	This part shows how the task is represented in the user interface and how it is performed.

4.1. Algorithm Node Execution

4.1.1. Selecting and Executing an Algorithm

Task

The most elementary operation in the tree is the execution of a single algorithm on a single CDC object. We choose a CDC node (a node which contains CDC objects) whose CDC object will become the algorithm's source data object. We then select an algorithm which is executable on this object, *i.e.*, which contains its class information in its 'required CDC' field. To ease this selection, the list of algorithms is filtered in order not to display algorithms which can't be applied. The children of the CDC node are searched for an occurrence of the chosen algorithm, and if none is found (the algorithm has not yet been applied to this node), an algorithm executor is created and is included in a new algorithm node. The algorithm executor then asks for parameters in its parameter window and runs its algorithm. On completion, it creates a new node containing the target CDC object. If we do find an instance of the selected algorithm within the children of the CDC node, the algorithm is reapplied. This procedure is discussed in Chapter 4.1.3.

This is the normal execution sequence of an algorithm which generates CDC objects. As we have seen in Chapter 3.3.5, rating algorithms do not produce a target object. They add their result information (property true or false, rating value) to the corresponding list in the source CDC object.

Objects

The first class we need to define in order to set up the action tree are the tree nodes. Following the definitions we found in graph theory, each node may have an unlimited number of children and is connected to exactly one parent. The tree is fully defined if each node just contains a list (on dynamic lists please refer to Appendix D) of its children. To allow traversing the tree in both directions we also include a reference to its parent.

A tree node also contains a reference to the graphical object displayed on screen which allows faster reflection of node state changes on screen.

Tree node

```
TTreeNode = OBJECT (TLList1)
```

Arrange data in tree format. Each node contains a list of its children. A full tree is accessed by its root.

Fields	
fParentNode	Parent node
fDisplayShape	Object on screen which displays this node
(Since TTreeNode is a subclass of TLList, children can be stored in the object itself. For details please refer to Appendix D)	
Methods	
InsertChild	Tree construction/destruction methods
RemoveChild	
GetLeftBrother	Access methods to reach related nodes
GetRightBrother	
GetParentNode	
GetCommonParent	Methods which analyze node relations
IsAncestor	
IsDescendant	

The action tree resulting from the data relations of the design cycle presented in Chapter 3.2.1 has a special structure. The action tree node types containing CDC or algorithm executors differ in both data contents and functionality. We therefore subclass the general tree node object to declare a CDC and an algorithm node class.

We keep in mind that we want to store the full working history in such detail that we can reproduce part of the tree from the data stored. We therefore need to be able to retrieve a source CDC object, an algorithm executor and parameter lists from the tree

1. TLList is a subclass of the MacApp TList class which implements a general list of objects. The Leporello subclass of this list includes a special object management mechanism to introduce some sort of garbage collection. See Appendix D for details.

and store the target CDC object therein. The basic structures of the CDC and the algorithm node classes follow immediately (they will later be extended as we add more features).

Algorithm node

TAlgATNode = OBJECT (TTreeNode)

Tree node which contains algorithm information. It has two main purposes:

It controls the algorithm executor, *i.e.*, it applies the source CDC object, acquires parameters before execution, starts execution and stores the result correctly in the appropriate CDC node.

It stores the algorithm run information, which is necessary to reproduce the target data from the source data (algorithm executor and parameter lists).

Fields	
fAlgorithm	an algorithm executor object
fCurrentAlgParamSrc	current source of parameters (TParameterSource object, explained later)
fAlgParamList	a list of former parameter sets (we will later see that one set is not sufficient, since we would like to apply the algorithm with several different parameter settings)
Methods	
NodeExecution	Initiates the execution of the algorithm. Tasks included contain source CDC acquiring, algorithm execution and result storage. The procedure has one parameter, the object which is in charge of supplying parameter sets (a parameter source described below).
AlgorithmCompletion	Actions necessary after algorithm completion.
StoreResults	Handles result storage properly, <i>i.e.</i> selects an appropriate target CDC node and stores the results.

Rating algorithms and other algorithms which do not return a result CDC are handled quite similar. Their node structure is the same, but since they do not produce a result object (rating algorithms add their result to the CDC object they are applied to) we do not create a target CDC node either. Algorithms of this sort are the only algorithm leaves of the action tree. All other functions (*e.g.* re-execution) are supported. We will therefore not mention this special algorithm class in the chapters to come.

CDC node

```
TCDCATNode = OBJECT (TTreeNode)
```

Tree node which stores algorithm results and supplies them to the algorithms which are applied to it.

Fields	
fCDCList	List of CDC objects supplied by various executions of the parent algorithm. Like the list of previous parameter lists in the algorithm node, considerations which are discussed later will show that we need a list here instead of just one CDC object stored.
Methods	
StoreResults	Called by the StoreResults procedure of the parent algorithm node. It adds the supplied CDC object to its list.
GetFirstCDC GetNextCDC	These and similar methods may be used to retrieve each CDC object contained in the list.

What rests to be defined is an object which supplies parameters to the algorithm. There might arise several situations where parameter sets for an algorithm have to be composed following varying objectives (one single list entered by the user, a previously applied list, a series of lists, etc.). We therefore define a class which returns parameter lists on request. This class is then subclassed to handle different modes of execution of the algorithm node.

Parameter source

```
TParameterSource = OBJECT (TLListSource)1
```

Object which is responsible for supplying appropriate parameter sets to an algorithm executor.

Fields	
Defined in subclasses	
Methods	
GetFirstParameterSet RequestNextSet MoreSets	Procedures to retrieve the parameter lists from this source

The first specialized subclass of the parameter source is an object, which allows interactive input of one parameter set. A call to the parameter requesting procedures results

1. The TLListSource parent class is a list of TLList objects.

in the display of a window where all parameters of an algorithm can be entered. To achieve this, the `ModalAskParameters` method of the algorithm executor is called.

Parameter input

`TParameterInput = OBJECT (TParameterSource)`

This subclass of the parameter source was designed to ask parameters from the user.

Fields	
<code>fAlg</code>	reference to the algorithm executor whose <code>ModalAskParameters</code> will be called
Methods	
<code>RequestNextSet</code>	Override to display the parameter window.

Algorithm execution on the tree level now looks as follows: The algorithm selection window returns the algorithm information object of the algorithm selected by the user. The children of the selected CDC node are searched for an algorithm of that type. If none is found, a new algorithm node is created which is properly connected to its parent CDC node, *i.e.*, the parent CDC object is made the new algorithm executor's source CDC object.

The command initiated from the user interface (*execute algorithm*) then creates a parameter input source and makes it the current source of the algorithm node. The node then triggers its algorithm executor to start the algorithm as described in Chapter 4.1. On return, it creates a new target node and stores a copy of the algorithm executor's target CDC object therein.

Diagrams of the algorithm execution will be shown once we have described all different node execution modes (in Chapter 4.1.4).

Tree root: As we have seen in the graph theory chapter (Chapter 3.2.3), the action tree root must contain an algorithm which produces a CDC object without requiring an input CDC object. This may be a file access algorithm, some data acquisition procedures or a request to the user to enter any desired CDC object on screen.

To comply to the algorithm definition (CDC objects as input and output objects), we define a special CDC which represents no CDC object (this does not mean 'which does not represent a CDC object', but 'which does represent a CDC object which is not there'). Any algorithm acting on such a `TNoCDC` object may become an action tree root. An empty document always contains an invisible `TNoCDC` object, which is automatically selected to allow the ordinary algorithm selection mechanism.

Display

The elements described in this and the following 'Display' chapters show the reflection of the features described in the action tree display¹. This display is a part of the

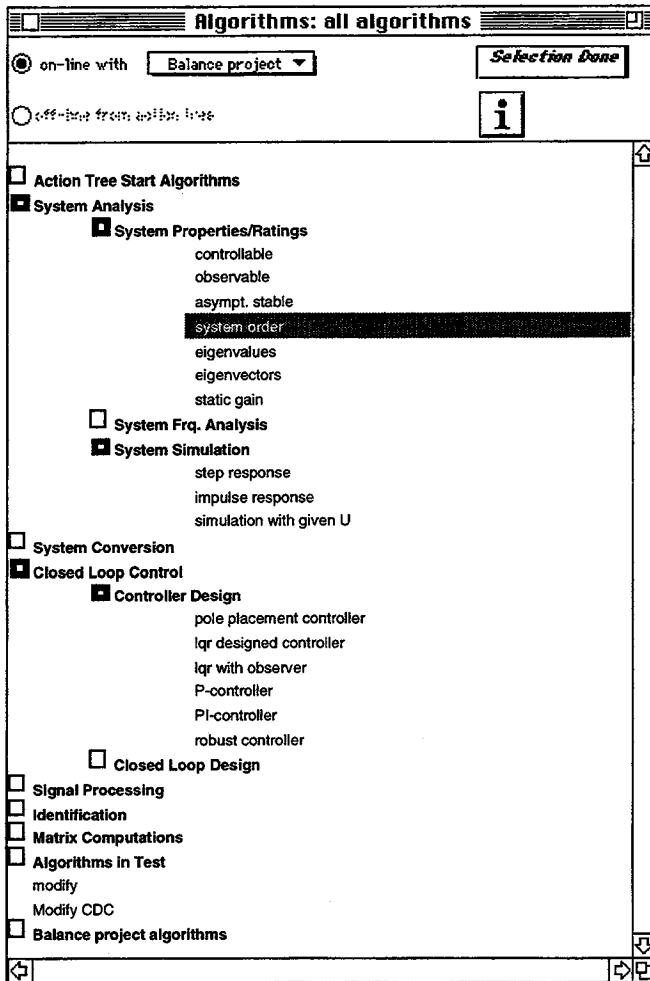


Figure 4.1: Algorithm selection window. The algorithms displayed are applicable to a selected node containing a state space system in file "Balance project"

Leporello document window, which will be presented in detail in Chapter 5.2.1 The buttons on the top row of this window (Fig. 5.4) are also where the commands and actions described are initiated.

The first thing to do when beginning a new tree is selecting the algorithm which creates the first CDC object. This is done in the algorithm selection window shown in Fig. 4.1.

1. For better quality of the drawings, the action tree drawings are all shown in their black and white version. On color displays, action tree nodes vary in color and in shading. Pictures of both versions are contained in Appendix B.

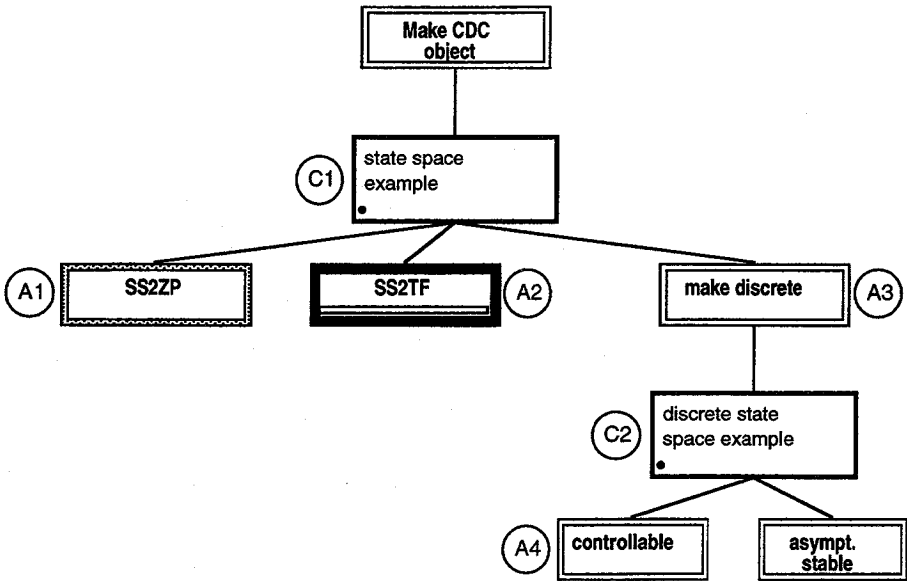


Figure 4.2: Action tree example with different algorithm node states

The selected algorithm appears in an empty algorithm node (Fig. 4.2, node A1) labelled with the name of the algorithm and ready to be executed. The user then starts the algorithm, which asks for parameters in the algorithm parameter window (A detailed description of parameter windows is given by Kolb in [22]). The node is then marked executing (Fig. 4.2, node A2). A horizontal bar (status bar) appears in the node and fills as the algorithm proceeds. Upon completion, the target CDC node is created (Fig. 4.2, node C2). It shows the name of the CDC object it contains. The algorithm node takes the picture of the completed node (Fig. 4.2, node A3).

Both the contents of the algorithm and the CDC node may be viewed after completion by opening an information window which displays all the relevant information available. This window allows changing the name of the CDC object (a sample window is shown in Fig. 5.5 on page 116).

4.1.2. Execution of an Algorithm on Different CDC Nodes

Task

If an algorithm has successfully been applied to a CDC node, we can try it on other nodes. If we are working on a continuous time model and would like to carry on the design in discrete time, we need to apply a discretization algorithm on all nodes we continue to use in further calculations. Using the technique presented in the last chapter, we would have to choose the same sampling algorithm for each of the selected CDC nodes. We would then execute each of them, therefore entering the same param-

eter value to specify the sampling time over and over. This may end up being a very tiresome task.

Since for the desired algorithm an executor is contained in an algorithm node after its first application, we would like to choose it not from the algorithm list, but directly from the tree. We can thus mark an existing algorithm node for re-execution on a selected CDC node.

If the selected CDC object conforms to the requested input CDC of the algorithm, it can be applied directly. We proceed as if the algorithm information came from the algorithm list, and create a new algorithm node if necessary. Since the original algorithm node already contains valid parameters, we are asked whether to re-use them or whether we want to supply new ones.

We see that the discretization example is sped up significantly by this feature. If, as we will see in the next chapter, an algorithm node contains parameters from several executions, re-application of algorithms will become very handy.

Objects

Re-applying an algorithm on a different source CDC object varies in two aspects from the standard case: the algorithm is not chosen from the algorithm selection window, and there are previous parameters available.

The first point requires us to test the applicability of the algorithm to the selected CDC prior to performing the task. Since the algorithm selection window only displays algorithms which can be applied to the selected CDC object, this was not necessary in the previous task. Algorithm node allocation then proceeds the same way as described there. The algorithm information object is taken from the algorithm node which is marked for re-execution instead from the algorithm selection window, but this does not make any difference.

Using the parameters available from previous executions does not impose any limitations, they may be applied to the new algorithm node for execution. To achieve this most easily, we extend two of the classes presented earlier:

The `TParameterSource` class is given the ability to not only supply parameter sets, but also to store them. Its methods are modified as follows:

Parameter Source

(continued)

Extended object definition to contain item storage methods.

<i>Methods</i>	
GetFirstParameterSet	The parameter retrieval methods all return the corresponding elements in the list. A counter is introduced to point to the current parameter set in the list.
RequestNextSet	
MoreSets	
InsertSet	Stores a parameter set in the list.

Now that the parameter source is defined as a more specialized list of parameter sets, we may easily take an object of type `TParameterSource` instead of a simple list object to store the previous parameter sets of the algorithm node. We therefore have a parameter source which upon request will return all the parameters used with the algorithm we would like to reapply. We may simply use its `fAlgParamList` and supply it as the current parameter source to the `NodeExecution` method of the new algorithm node.

Display

In order to be re-applied, an algorithm (Fig. 4.3, node A1) has to be marked. This is done either by command-clicking on the algorithm node, or by pressing the corresponding toolbar button. The node mark is displayed as a line on the left side of the algorithm node box (red on a color display). A marked algorithm may be applied to a selected CDC node if it can be executed on this CDC object (node C2).

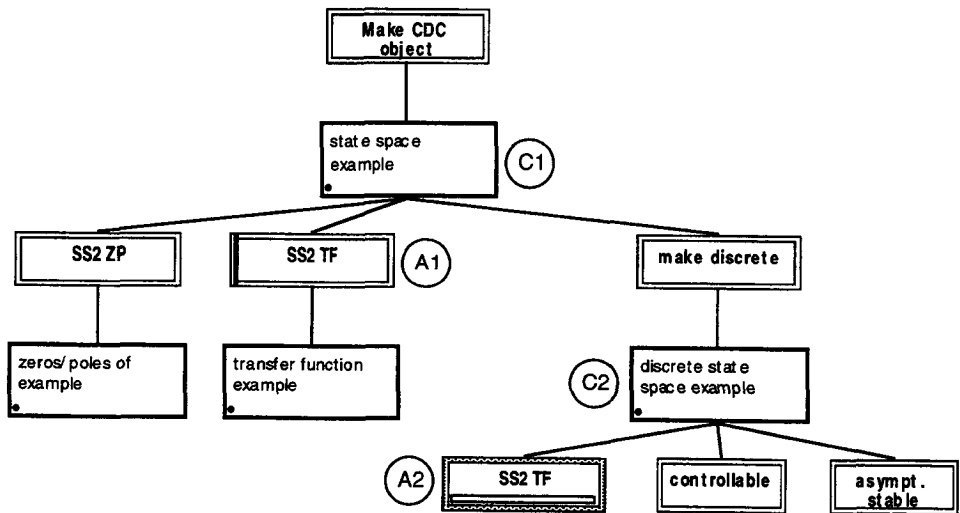


Figure 4.3: Algorithm scheduled for re-execution

The selected CDC node receives a new algorithm child (if the marked algorithm has previously been applied, its node is selected). If the user chooses to reapply the parameters of the marked algorithm, they are connected to the newly created node (node A2). This is reflected by showing the empty status bar in this node. This also indicates that the node is now inconsistent and needs to be re-executed. As this is done, the status bar fills and vanishes as soon as the node reaches consistency (all source CDC objects have been processed with the new parameter source).

4.1.3. Creation of Multi-object Nodes

Task

When an algorithm is executed, choosing appropriate parameters is not always an easy task. There are several situations where algorithm parameters have to be varied to get adequate results.

Wherever the design step involves setting some weighting parameters, this choice often requires engineering experience. It may be necessary to apply several design parameters to achieve satisfying results. One example of this kind is a controller design using the LQR method. Selecting weighting matrices Q and R is left to the user's experience.

Other methods explicitly require parameter variations during the design process. In system identification the order of the model is not known beforehand. Engineering skills are necessary to choose a range of possible orders. An identification algorithm is executed for each order and the resulting models are compared afterwards. This procedure is clearly shown by Ljung in his example of Chapter 17.2 in [23]. Similar situations exist with model reduction or in some cases when choosing the order of a controller.

If we store each algorithm result in a separate node, the use of parameter variations rapidly leads to an explosion of the tree. Although the action tree was initially designed to organize experiment results, a large number of nodes may also be a source of user confusion.

Multi-object nodes: We consider parameter variations to be a very useful feature of Leporello, which deserves additional attention. Therefore the CDC nodes of the action tree are extended. The results of an algorithm applied repeatedly to the same source CDC object are collected in a single node: a *multi-object node*. All CDC objects in this node share the same structure (object class, name, etc.), but differ in their parameter lists. They may be regarded as particular numerical variations of the same CDC object.

Objects

Parameter variations may be achieved in two different ways: we may either apply several parameter sources to an algorithm node, each providing a single parameter list, or we may use a parameter source which returns a whole series of parameter lists.

Both ways do not require special precautions. Upon node execution, the algorithm node retrieves the first parameter set from the applied source and continues to process each consecutive set. After processing, these parameter lists are stored in the node's `fAlgParamList` field. This loop is repeated each time the user supplies a new parameter source (*i.e.* enters a new set of parameters, etc.).

We already prepared the object structure of the CDC node to contain a list of CDC objects. However, this may lead to consistency problems. We have seen that all CDC objects in a node share their structural information. If, for example, the user changes

the name of a CDC object contained in the node, we need to change all the names in the list. We therefore split up the CDC object list into one current CDC object containing the name and other administrative information, and a list of its varying parts (parameters, object properties, ratings). This new construct needs special management to always keep the current CDC in a coherent state. This is done in a special class. The occurrence of a list of parameter sets in this object suggests the use of a subclass of `TParameterSource`. The supply of a whole list of CDC objects to an algorithm presented in the next chapter justifies this decision.

CDC object source

`TCDCSource = OBJECT (TParameterSource)`

Stores and supplies a set of CDC objects all sharing the same structural information.

<i>Fields</i>	
<code>fCurrentCDC</code>	Current CDC object. It contains the structure information
<code>fDynamicProperties</code>	List of tested object properties of each CDC object
<code>fRatings</code>	List of ratings applied to each CDC object
(the parameter lists are handled by the inherited methods and fields)	
<i>Methods</i>	
<code>GetFirstCDC</code> <code>GetNextCDC</code> <code>InsertCDC</code>	Methods to store and retrieve CDC objects similar to the corresponding methods of the parameter source.

The first CDC object to be stored in the list supplies the structural information. Its parameters, properties and rating lists are the first items in the corresponding source lists. Any subsequent CDC objects lose their structure information¹, their object attributes are copied to the source lists.

Display

All multi-object operations are shown in Fig. 4.7. Node A2 shows an algorithm which has been executed repeatedly on the same CDC node (node C1). This is reflected in the multi-object bar on the right of the algorithm node A2 (green on a color display). To stress the idea of multiple objects being fed from the algorithm to the CDC node, the branch connecting the target CDC node is wider than one leading to a single CDC node. This target node C2 also receives a multi object marker on its right side.

1. We will later see that this information must be the same as the one in any previously stored CDC object in order to share the same node.

4.1.4. Application of an Algorithm to a Multi-object Node

Task

In some cases we would not only like to see the effects of an algorithm applied with varied parameters, but also the impact of a single parameter setting on a series of CDC objects. We have seen that the decision about a parameter selection may be postponed until we see its consequences. If the decision cannot be made even then, we would like to apply further algorithms to the whole bunch of CDC objects.

Consider the following example: We have tried an identification algorithm within a certain order range and would then like to design a controller for the model found. Rating functions tell us that the identification gives satisfying results with model orders 3 and 4. It may be reasonable to complete the whole controller design with both models and decide only afterwards whether to take the controller derived from the third or fourth order model for implementation.

Multi-object subtrees: In Leporello this task is achieved by applying an algorithm to one of the multi-object nodes developed in Chapter 4.1.3. The results of this action satisfy the conditions for building a multi-object node (all CDC objects have the same structure and result from the execution of the same algorithm on the same CDC node). The application of an algorithm to a collection of CDC objects therefore describes a second way to form multi object nodes. With this structure we can build whole subtrees containing multi-object nodes (multi-object subtrees). This gives us the ability to compare alternative solutions in a large range.

Multiple algorithm children: The possibility of building multi-object subtrees adds a requirement to a multi-object node. If we want to apply an algorithm to a whole list of CDC objects, each of them must conform to the algorithm's requirements, *i.e.*, it must be of the object class which is contained in the algorithm information object's required CDC field. Therefore, all CDC objects contained in a multi-object node must share the same class and class properties. If an algorithm produces CDC objects of several classes, we have to store objects with different class properties in different multi-object nodes. The algorithm node then has more than one child.

We now see the motivation for defining algorithm nodes in the tree: if we had defined the algorithms to be stored in the tree branches, multiple leaves for one single branch would not be possible. With algorithms defined as nodes this is feasible.

Limitations: The flexibility gained by the introduction of multi-object nodes may also be a source of problems. If we repeatedly vary the parameters of an algorithm applied to a multi-object node, the number of CDC objects in the resulting node explodes. Even in the highly structured environment of Leporello the vast amount of data may become confusing. This requires us to limit the flexibility of multi-object subtrees. In order to apply parameter variations to a multi-object node, the user first is asked to decompose the node. This task will be the next feature described in Chapter 4.2.1.

Consistency: Managing multiple objects in tree leafs does not cause great difficulties in maintaining consistency. Problems may arise if we modify parameters of an algorithm node already contained within a subtree (more algorithms have been applied to the result CDC nodes). In that case, the target node of the modified algorithm node contains an additional CDC object to which all children algorithms of this CDC node have not yet been applied. All these algorithms (and their descendants) have to be re-executed in order to have a consistent subtree (an automated task to do this will be presented later).

Target CDC node: Again, this may cause consistency problems. Consider the case where parameter variations have been previously applied to an algorithm node in the subtree of the inconsistent node. Rendering its parent (source CDC node) consistent implies adding another CDC object, therefore making it a multi CDC node. We now have the situation, where a parameter variation has been applied to a multi-object node; the limiting condition imposed previously is violated. We conclude that the parent CDC node of a multi parameter algorithm node should not be used as target CDC node of its parent algorithm. Automatic subtree re-execution will therefore not store any results in a node which was previously involved in a parameter variation operation.

Twin CDC node: To allow these variations nevertheless, we add a new child to the parent algorithm mentioned. It contains the same CDC structure information as the first target node (we will therefore call it twin node). Contrary to the original target, it does not have any children but is a multi-object node which contains the results of all but the first execution of its parent (which stays is the original node). This node may then be used like any other node in the tree, *i.e.*, it may also be merged (see Chapter 4.2.2) with the original node. Although the result is the same as if we had kept the original target, this explicit node operation forces the user to be aware of the increasing problem complexity.

Objects

The definition of the CDC source class in the Chapter 4.1.3 proves to be useful, now that we want to apply an algorithm node to all the CDC objects contained in its parent node. The CDC source was equipped with the appropriate methods to retrieve the whole series of CDC objects contained. These methods (`GetFirstCDC`, `GetNextCDC`) replace all object attribute information (parameters, object properties and ratings) of the structure CDC object by the corresponding items from the appropriate lists. Access through these methods always provides a consistent current CDC object.

The algorithm node parameter loop presented earlier is carried out for each CDC object retrieved from the source CDC node. The whole state transition diagram of an algorithm node is shown in Fig. 4.4. The diagram illustrates the algorithm node behavior when one parameter source is supplied to the `NodeExecution` method. Each parameter set it returns is applied to each of the CDC object contained in the parent.

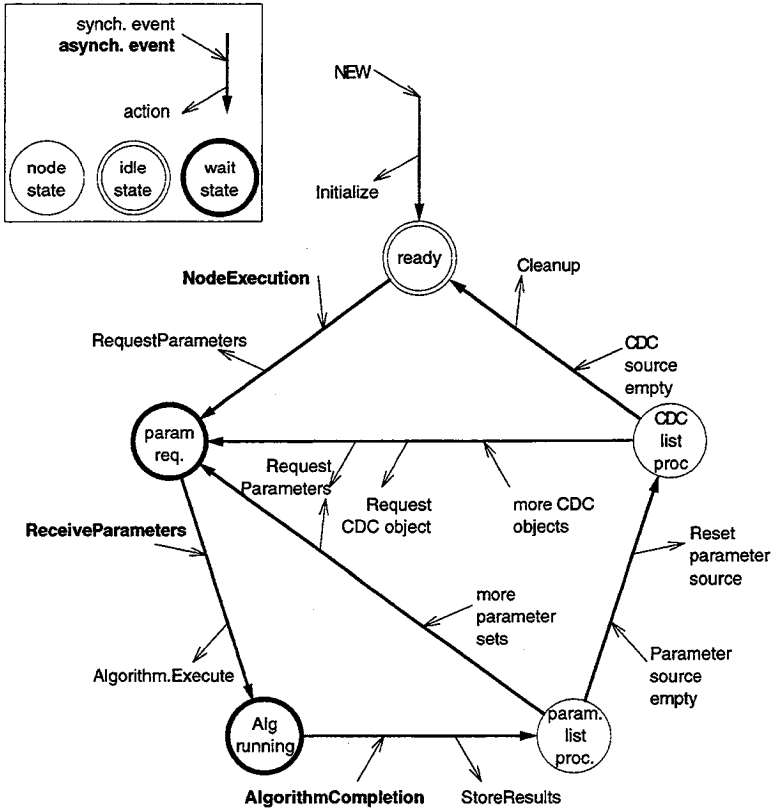


Figure 4.4: State transition diagram of an algorithm node

The algorithm node is ready after its creation. Algorithm execution is initiated by a call to the `NodeExecution` method to which a parameter source is supplied. This source is then asked for parameters and the node waits asynchronously for the parameter list to be processed (state `param req.`). As soon as the parameters are available, the algorithm executor is triggered for the algorithm execution; the node is again put on hold until the asynchronous termination of the algorithm (state `Alg running`). Completion of the algorithm executor triggers storage of the results which includes the allocation of a new node. Depending on whether more parameters are available, the next parameter set is processed, or the parameter source is reset and the next CDC object is requested. These nested loops continue until all parameter lists supplied by the source are applied to all CDC objects given by the parent node. Temporary objects used for the execution cycle are cleaned up afterwards; the node is again in its ready state to receive further execution requests.

CDC object history references: In the context of the action tree, the history of a data node can easily be traced. However, details of the origin of a single CDC object contained in

a multi CDC node may still be lost. The algorithm which produced the object is obvious, but exactly which of its grandparent multi node's elements, or which of the parameter sets of its parent algorithm node was used, is not visible. We therefore store a reference to each the parent and the grandparent item in addition to the result CDC object.

The objects which store these links are mainly lists of integer numbers, indicating the list index of the referenced object in the respective node list of the parent or the grandparent. In addition to the index list, the index object also contains an entry which points to the original list.

With a single method override we can add another feature to this reference list, which might prove useful. Retrieving an object from a simple list as it is implemented in MacApp (TList.At) is shown in Fig. 4.5a. The address of the list element is calculated from its index and the base address of the object, then dereferenced, and the resulting object is returned. With our index list, we can easily implement a masked list just by overriding the dereferencing method. The functionality of the new method is shown in Fig. 4.5b: Instead of using the supplied index for the address calculation, we use the index list element. Since each list operation accesses this dereferencing method, we do not see that the objects retrieved are no longer in a sequential list but in arbitrary order.

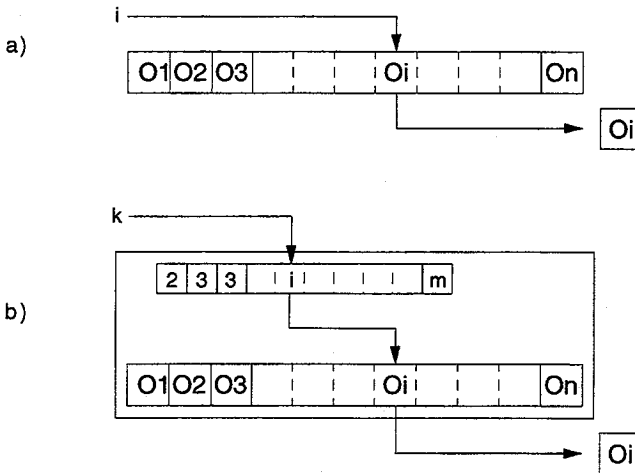


Figure 4.5: Leporello list access: a) direct access in MacApp, b) indirect list access

Subclasses of this indirectly referenced list could sort a given list without changing the order of the original list, or they could access a subset of the list without actually creating a new list and copying all the elements in the subset.

Indirectly referenced list

`TDb1IndexParamSource = OBJECT (TParameterSource)`

Parameter source which re-maps access of another source. Elements of the mapped source are remapped onto the local index scheme.

Fields	
<code>fIndexList</code>	List of the indices
<code>fParameterSource</code>	Reference to the list which contains the original objects
Methods	
<code>At</code>	Override to implement the indirect index calculation
<code>InsertIndex</code>	Methods to add or remove an index reference in the list
<code>DeleteIndex</code>	

Twin node structure: The twin nodes introduced are ordinary CDC nodes with identical CDC object structure. To reflect the fact that their structure information would allow their collection in a single CDC node, we link them together by introducing *twin references* to the CDC node. These links form a chain connecting all twin nodes of the same structure. We will later see that single CDC objects contained in any of the twin nodes may be freely moved to another twin which may receive additional CDC objects (*i.e.* may become target). To check whether all these CDC objects do share their structure information, or whether a newly created object may be stored in the same node, the CDC node class contains a method which calls the appropriate test of the CDC (`EqualCDCStructure`).

Including the history reference lists and the twin node entries in the CDC class, we get the following extended definition.

*CDC object node**(continued)*

Extended definition to include references to parameter lists which were used to create the CDC objects in the node. Additional extensions to manage twin nodes.

Fields	
<code>fAlgParamReference</code> <code>fCDCParamReference</code>	Indirectly referenced lists of ancestors
<code>fLeftTwin</code> <code>fRightTwin</code>	Adjacent twin nodes, NIL if none

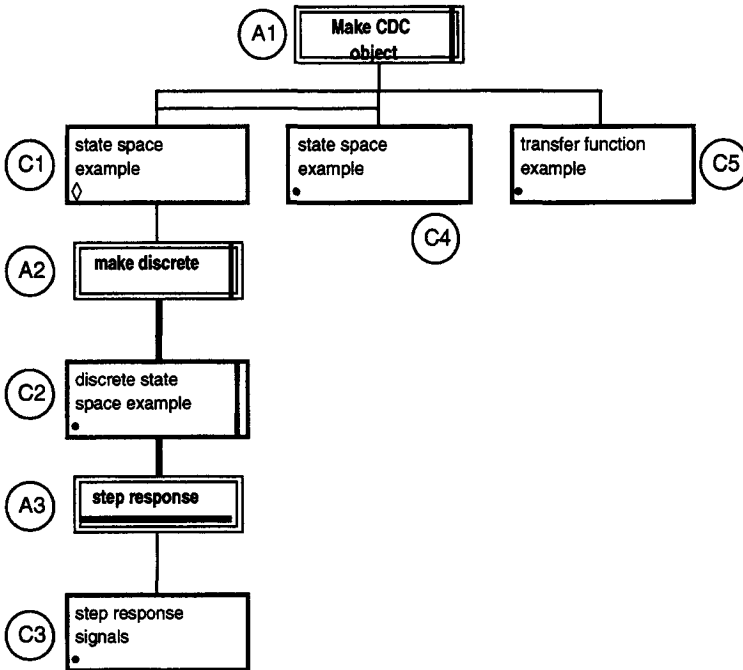


Figure 4.7: Multi-object operations examples

Display

In Fig. 4.7 all multi-object features are combined. The tree displayed was created in the following order:

1. Algorithm A1 is executed, which creates CDC node C1.
2. Algorithm A2 is executed once, this creates CDC node C2.
3. Algorithm A3 is executed on the CDC object in C2. This creates the CDC node C3.
4. Algorithm A2 is executed a second time. C2 receives a second CDC object and node A3 becomes inconsistent. This is indicated by the shown status bar in A3. The multi-parameter node A2 prevents CDC node C1 from becoming target (to avoid automatic parameter size explosion). This is shown by the '◊'-mark (red on a color display).
5. Algorithm A1 is executed again, the result has the same structure as the CDC object already contained in C1. Since C1 cannot automatically become target (*i.e.* not receive any further objects), a new twin node with equal structure is created (C4). This new node receives the result of the second execution of A1. Its close relation to C1 is shown by the twin branch which connects C1 and C4. Since there are not yet any multi-parameter algorithms applied to C4, it may still receive additional CDC objects. This is shown by the target mark (•, red on a color display).

6. Algorithm A1 is executed for the third time. We assume, that the result has a different structure this time than the objects in C1 and C4. A new sister CDC node (C5) receives the result. Any further results with the same structure may be stored in this new node; it therefore receives a target mark.

4.2. Managing Multi-object Nodes

4.2.1. Extract a Subset Node from a Multi-object Node

Task

Multi node operations presented earlier helped to accumulate data. Parameter variations are supported to evaluate the best result from a whole range of algorithm parameters. However, once we decide which CDC object to choose for further calculations, we have not yet the means to separate this chosen object from the multi-object node. We need a task to break a multi-object node into subsets.

In the previous section (Chapter 4.1) we have prepared the structures to support this operation: Separating CDC objects from a multi-object node may be achieved using the twin CDC node introduced there. A series of twin nodes all contained the same CDC information (name, class properties etc.), but the CDC object information (parameters, object properties) resulted from several executions of its parent algorithm. Instead of just storing different algorithm run results in various twin nodes as previously shown, we may also fill one multi-object node with all the results and dissect it later in a separate task.

Once again, working on a tree leaf does not impose any problems. Creating a twin node is merely a matter of moving items from one node to another. The task becomes more complicated if we choose a CDC node with children. If we just move items between the node and its new twin, all results derived from the moved CDC objects loose their ancestors.

Twin subtrees: To keep the tree consistent we have to transfer all descendants of the objects now contained in the twin node too. Descendants of the twin node are removed from the original subtree and added to the twin's subtree. All tree references are then up to date.

If we just create a twin node to remove a CDC object history from a subtree (if we are not interested in the work done on the moved CDC objects), we can decide whether or not to copy the subtree. If we do not, all selected items are moved into the twin node, their descendants are deleted.

Target selection: If we re-execute the algorithm which is parent to a series of twin nodes, the question arises which one of the twins will receive the new results. There is no automatic solution to this problem. We therefore let the user choose a target twin to contain further algorithm results. If no target is set, the algorithm node automatically

assumes we do not want any of its children to receive additional results and creates a new twin node, which automatically receives the target mark.

Objects

We have already introduced the twin node extension of the CDC node class which is used to break up multi-object nodes. We merely have to include methods to move CDC objects between twin nodes. These methods do not simply remove the CDC object from one node and put it into the twin node. The moved object may take its previously applied subtree with it, which imposes some problems. The descendants of the moved object have to be identified and transferred to the corresponding new node. The indirectly referenced lists are used to accomplish this, they have in turn to be re-mapped.

Target tag: The current target twin is marked by a simple boolean tag. The `SetTarget` method sets the mark if the node is allowed to become target and removes it from the previous target twin.

The CDC object node is now extended by the following fields and methods:

CDC object node

(continued)

Object definition extended to manipulate twin nodes.

Fields	
fTargetMark	Flag to mark the target twin node
Methods	
MoveToTwin	Moves the indicated CDC objects to a given twin node
IsTarget	Target management
SetTarget	
GetTargetTwin	
CanBecomeTarget	True, if this node is allowed to become target

The algorithm node is adapted to look for a target child to store its results upon completion. If none is found it continues as if it did not have any children, *i.e.*, it creates a new child.

Display

A pair of twin nodes has already been shown in Fig. 4.7. The case where twin nodes are created not because one of them may no longer become target, but by user selection, is not much different. Now each of the twins may receive the target marker. We therefore allow it to be moved by the user. Setting the mark (by a command-click or by pressing the corresponding button when the designated target is selected) removes it from the previous target to assure that the node which receives more results from its

parent is well defined. The target mark is removed from the twin group if the target is command-clicked.

An example has been shown in Fig. 4.7: Nodes C1 and C4 could have been created by first executing algorithm A1 twice (which creates C1 as a multi-object node), then performing all the actions described in Chapter 4.1.4 and finally moving the second result of A1 to a new twin node without copying its subtree.

4.2.2. Merge Subset Nodes

Task

To complete the operations on twin nodes, we need to add a task which merges two previously separated twin nodes.

Both CDC nodes need to be in the same twin group in order to be merged. This is easily done if both nodes are leaves, *i.e.*, do not have any algorithms applied to them. The resulting node contains all CDC objects previously stored in the merged twin nodes.

The task becomes more complicated if one node is parent to a subtree, *i.e.*, has algorithms applied to it. After the merge operation, the node contains a number of CDC objects to which its children algorithms are not yet applied (the node is inconsistent). The whole subtree has to be rendered consistent before any subsequent operations are applied to it.

Limitations: Merging nodes becomes almost impossible, if both nodes have algorithm children. Merging both subtrees results in a large number of inconsistent nodes, inconsistent in different respects. Some algorithms have been applied to one of the merged nodes only. Some others have been applied to both of them, but with partly equal parameter sets (Unfortunately, we cannot recognize two identical parameter sets). To circumvent these problems, we only keep the subtree of the target node which receives the CDC objects of the source node. The source node's subtree is discarded. If it does contain data which are to be kept, its algorithms need to be applied to the target node prior to the merge (see Chapter 4.1.2).

Objects

Merging to twin nodes does not require any additional fields or methods in any of our classes. The CDC node method which moved some CDC objects from one node to one of its twins serves also to move the items back. The complications we may run into when we merge two subtrees we avoid by discarding the subtree of the source CDC node.

Display

The display elements used when merging two twin nodes have all been introduced before. The target mark is used to indicate the node into which the selected CDC node is copied and the shown status bar of its children algorithms indicates their inconsistency.

4.3. Managing Tree Complexity

All the operations described up to now helped to accumulate data and to expand the action tree. If a given project (tree) may be clearly divided into subtasks, we might want to perform them in different tree documents. Copying results from one tree and pasting it into the other is a handy operation then. In some situations we would also like to be able to clean up parts of the tree. Some procedures applied may not lead to satisfactory results. The branches and subtrees containing these procedures may be removed from the tree. This leaves the successful (and also the interesting) parts of the tree in the final design tree of the project for further work or for documentation.

4.3.1. Copy and Paste Operations

Task

Copying objects and pasting them somewhere else is a standard operation in today's user interfaces. In Leporello, general copy/paste operations are only possible with limitations, depending on the copied objects and the location where they are pasted to.

Pasting to other applications: In addition to the Leporello related data the copy operation puts a picture of the selected part of the tree to the Macintosh clipboard. From there, it can be pasted into any other application which handles pictures (all tree pictures in this thesis were treated that way).

Pasting within Leporello: This task is not feasible for all possible situations. One limitation is imposed by the fact that the target tree must conform to all Leporello constraints after the paste operation, *i.e.*, it must be a proper tree as defined in Chapter 3.2.3. Hence, we only allow one connected subtree to be copied. The location where this subtree is copied to is always a single node. The root of the copied subtree will become a new child of the selected target. We therefore encounter different situations depending on the class of the copied root, the number of copied nodes and the class of the node which receives the pasted subtree.

Copy one algorithm node: Copying a single algorithm node is handled like as if a re-applied algorithm node were produced (*cf.* Chapter 4.1.1). The node's algorithm must be executable on the designated target, which must therefore be a CDC node. If this is possible, the copied node is treated like the marked algorithm node in the re-applying task described in Chapter 4.1.1. If the algorithm has previously been applied, this previous algorithm node receives the parameter source of the copy to be applied later.

Copy the subtree of an algorithm node: Since the root algorithm receives a new parent with the paste operation, all data available in the copied subtree's CDC nodes become invalid. The paste operation therefore copies empty, inconsistent CDC nodes to the target. These pasted nodes need to be rendered consistent afterwards.

Copy a CDC node: We recall that the ancestors of a CDC node contain its history. If we copy a CDC node to another parent outside its history path, this quality is lost. There

is no possible explanation of a copy operation of this sort, we don't consider it reasonable and therefore necessary to be implemented in general. However, it may be necessary to copy the results of a lengthy operation to an empty tree to continue working on it. Since each tree needs to be rooted in an algorithm node we introduce a paste algorithm. A node with this algorithm is automatically created to receive the results of the paste operation as its children.

Copy the subtree of a CDC node: Contrary to the algorithm node's subtree, the nodes of a CDC node's subtree keep their valid reference to their ancestor data up to the copied CDC root. After a successful paste operation, the whole connected subtree may be attached to the pasted root.

Objects

The copy/paste operations within Leporello require the following object adaptations:

Copy one algorithm node: No extension is required. The necessary structures have been introduced when we allowed to reapply an algorithm to another CDC node.

Copy the subtree of an algorithm node: We will later show how a series of allocated nodes can be executed in sequence (hierarchical algorithm, Chapter 4.4.2). The subtree of a pasted algorithm node is handled similarly.

Copy a CDC node: The paste algorithm created with the operation has quite a different behavior compared to other algorithms. It receives its source CDC objects not from its parent node but instead it takes them from the clipboard. The execute operation only passes these objects to the result node where they are stored in the run of the algorithm completion operations.

The simplicity of the operation and the special behavior of the algorithm led to the decision to support the paste operation not by introducing a special algorithm, but a special algorithm node: the `TPasteNode` subclass of the `TAlgoATNode` class.

"Paste CDC object"-algorithm node

```
TPasteNode = OBJECT (TAlgoATNode)
```

Special node which creates a CDC node which contains the CDC objects on the clipboard.

Methods	
AlgorithmReady	Override to return true if the correct node is on the clipboard.
NodeExecution	Overrides to organize the correct passing of the CDC node on the clipboard into the target CDC node.
DoParameterRequest	
ReceiveParameters	
Cleanup	

Display

The pasted nodes do not receive special attention in the display. A pasted algorithm can not be distinguished from an algorithm selected with any other method. The pasted CDC also shows no difference to ordinary CDC nodes. In order to reflect the fact that the CDC node history, which was lost by the paste operation, contained a sequence of algorithms, the paste CDC node is displayed as a hierarchical algorithm.

4.3.2. Delete Nodes

Task

Any node in the tree may be removed from the project. Its subtree loses the connection to the tree root and is therefore deleted too.

Objects

The Free methods of both the CDC node and the algorithm node classes sufficiently manage the task. Since we always delete the whole subtree connected to a node, the complicated update of indirect references mentioned in Chapter 4.2.1 is not necessary.

Display

A deleted node is removed from the display. Showing an empty tree display would certainly add some humorist touch to this thesis, however, we refrain from that in order to save space.

4.3.3. Mark Deleted Subtree

Task

Deleted node objects: In a complete design documentation it may also be interesting to see which tests and algorithms did not produce adequate results. If we remove these paths entirely, the information about the inadequacy of the results is lost. If the tree is still being worked on and someone tries to follow the same misleading path when trying to improve the design, it may be helpful to get a warning.

To store the information about the deleted subtree, we insert a specially marked node at its root. This deleted algorithm node contains the information about the root algorithm which was deleted. Instead of algorithm parameters, the node contains the reason, why this subtree was deleted. This information is asked from the user.

Reactivating deleted nodes: If a deleted algorithm is again chosen for execution, we get a warning indicating the reason for discarding this path. If we then still want to repeat this algorithm, the node is reactivated and may be used as normal. However, previous parameters and subtrees have been removed when the tree was marked deleted; they are no longer accessible.

Objects

The information we want to store about a removed algorithm node is the name of the algorithm and the reason, why this algorithm did not produce any satisfying results. To achieve this, we do not need a new node class, we simply replace the executed algorithm executor with a 'deleted algorithm' object. The deleted algorithm is a subclass of the algorithm executor class. It therefore contains the same fields as the algorithm which is replaced by the deleted algorithm. Two differences are introduced to implement the deleted algorithm: It can not be executed and a text parameter containing the reason for its deletion is made its only parameter.

In all other respects the algorithm node behaves the same as any other algorithm node. If the deleted algorithm is again chosen in the algorithm selection window, the deleted node is selected, and if the user tries to get information about the node, the parameter list of the algorithm describes the reason for the node deletion.

Deleted algorithm

`TDeletedAlg = OBJECT (TAlg)`

An algorithm removed from the tree.

Fields	
(deleted reason)	The reason for deletion does not require a field of its own, it is made a parameter and inserted into the parameter list.
Methods	
Execute	Override to disable algorithm execution. Instead, the user is asked whether to reactivate the algorithm. The deleted algorithm is then replaced by the algorithm which used to be contained in the node before its deletion.

Display

An algorithm node marked deleted keeps the algorithm name in its display. The node itself is crossed out by a bar line (red on color displays). An example of such a node is shown in Fig. 4.8.

4.4. Automatic Subtree Execution

In the preceding chapters we have seen many examples where previously applied algorithms are re-executed. Since we store all the information necessary, this may not only be done step by step initiated by the user, but also automatically. We will now show a few procedures which provide automatic execution of a series of algorithms.

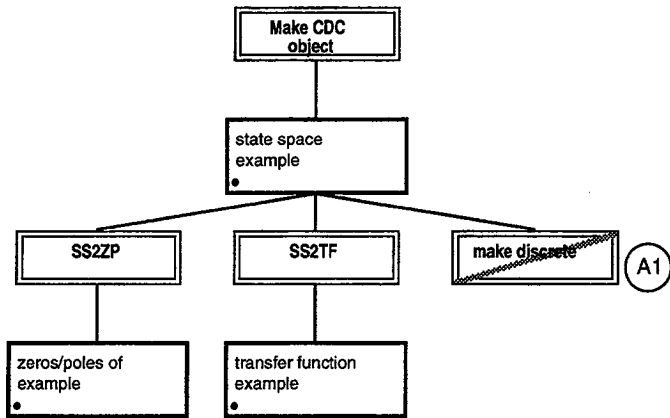


Figure 4.8: Algorithm node marked deleted

4.4.1. Make Subtree Consistent

Task

We encounter the first motivation for a series of algorithms to be automatically re-executed when an intermediate algorithm node becomes inconsistent, *i.e.*, when its parent CDC node receives additional CDC objects. This intermediate algorithm has to be repeated with its previous parameter lists on all newly added CDC objects. In turn, all its grandchildren become inconsistent. Maintaining consistency in a whole subtree can easily be automated.

If we choose to automatically keep a subtree consistent, each inconsistent grandchild of an algorithm is scheduled for re-execution as soon as the algorithm terminates.

Objects

Since the algorithm execution cycle contains several asynchronous calls (*cf.* Chapter 3.3), scheduling a series of algorithms is not as easy as calling one after another. The next algorithm to be applied may only start after its predecessor has finished. This event is not synchronously detectable. We therefore have to introduce a new object class: the algorithm scheduler. This object is activated on algorithm completion and can then start another algorithm.

Algorithm scheduler

```
TAlgScheduler = OBJECT (TObject)
```

Object which schedules algorithms and algorithm sequences

Fields	
none	Fields are added as required in the subclasses.

<i>Methods</i>	
ExecuteNextAlgorithm	This method is called after algorithm completion. It may initiate the next algorithm execution.

In order to be able to signal completion, an algorithm node includes an algorithm scheduler object whose ExecuteNextAlgorithm method is called after completion.

Algorithm node

(continued)

Additional definitions to include algorithm scheduling facilities.

<i>Fields</i>	
fNextAction	Algorithm scheduler which schedules the next algorithm.
<i>Methods</i>	
AlgorithmCompletion	If the fNextAction field is not nil, its ExecuteNext-Algorithm method is called.

Maintaining consistency in a subtree requires a subclass of the algorithm scheduler which tests all grandchildren of the algorithm node for inconsistency. A new consistency scheduler is allocated for each inconsistent node and they are all connected to form a dynamically linked list. Each consistency scheduler removes itself from the list and starts its successor.

Consistency scheduler

TConsistentScheduler = OBJECT (TAlgScheduler)

Scheduler which executes each inconsistent grandchild of its owner.

<i>Fields</i>	
fFirstAlgorithm	The owner of the scheduler, <i>i.e.</i> , the node which triggers the next algorithm upon completion.
fNextAlgorithm	The next node to be executed. This node contains the next consistency scheduler and therefore links the list.
<i>Methods</i>	
ExecuteNextAlgorithm	Override to detect inconsistent grandchildren and schedule them for re-execution.

Upon creation, the consistency scheduler enters itself in the fNextAction field of the algorithm node it is assigned to. If this field is already occupied by another scheduler, this scheduler is linked to the consistency scheduler.

Display

Automatic consistency update does not show in the display. Inconsistent nodes are marked by a visible status bar. One after the other changes its state to executing and finishes. The algorithms which are rendered inconsistent by this operation are marked in turn and executed later. Depending on the algorithm complexity (the time spent in the external package) this process may be monitored on screen (very short algorithms manage to terminate before the state change is visible).

4.4.2. Hierarchical Algorithm

Task

If during a design cycle one branch contains a series of algorithms which could be used repeatedly in the same order, it would be convenient to automate this execution sequence.

Since we rely on Matlab running in the background, one possible solution is to leave Leporello, write a Matlab file containing the algorithm sequence and add it to the set of available algorithms in Leporello. However, this task is far too complicated and imposes problems if Matlab is accessed on an external server or some algorithms involved in the sequence access other packages or internal algorithms.

Hierarchical algorithm: Once the sequence has been executed in a tree, the branch containing it is available on screen. We use this branch to define a new algorithm which applies the branch sequence to another CDC node. Once this hierarchical algorithm is defined, it is available in the algorithm list like any other algorithm. A branch which contains a hierarchical algorithm may therefore be used to define another hierarchical algorithm, hence allowing nested hierarchies in algorithm definitions.

Hierarchical algorithm classification: In order to fit into the algorithm ordering and selection mechanism, we need to provide the same information which is available for other algorithms, *i.e.*, what type of source CDC it needs and what CDC type it produces.

This information can be taken from the algorithms combined in the hierarchical algorithm. The new parameter list contains the parameter lists of each single algorithm. Once defined, the user does not see any difference between a hierarchical algorithm and an ordinary one.

Objects

When we discussed the automatic tree update in Chapter 4.4.1, we introduced all the mechanisms to schedule and execute a series of algorithms. This allows us to define a hierarchical algorithm (a series of algorithms) as a tree branch re-executed automatically. We therefore need two additional classes:

Algorithm information object: First we need to define a new algorithm information (algorithm head) object which contains enough information to classify the algorithm and create an algorithm executor. This is done in a subclass of the `TAlgHead` class.

The hierarchical algorithm head inherits the algorithm information fields (cf. Chapter 3.3). Their data are copied from the corresponding fields of the algorithm sequence combined in the hierarchical algorithm. The source CDC and properties are the same as in the first algorithm in the sequence, the produced CDC information is taken from the last algorithm in the sequence. The name of the new algorithm is supplied by the user.

The sequence is stored in a list of algorithm head references.

Hierarchical algorithm information

THierarchAlgHead = OBJECT (TAlgHead)

Store information about a hierarchical algorithm

<i>Fields</i>	
fAlgSequence	List of the TAlgHead references of the algorithm sequence.
<i>Methods</i>	
CreateParamList GenAlgorithm	Overrides to create a hierarchical algorithm executor object.

Hierarchical algorithm executor: The hierarchical algorithm executor which is created by a THierarchAlgHead object mainly consists of a tree branch which has the algorithm nodes filled with the algorithm sequence, and the CDC nodes allocated, but empty. The root of the branch is an empty CDC object. Setting the source CDC object of the executor (SetDemandsCDC method) copies it into this empty root node. The target CDC object of the last algorithm is made the target of the hierarchical algorithm, i.e., it is returned by the TakeAwayProducesCDC method.

When a series of algorithms is defined as a hierarchical algorithm, we assume that the user is interested in the final result of the series only. This allows us to clean up the intermediate CDC objects after each execution. Intermediate algorithm parameters are contained in the parameter list of the hierarchical algorithm executor. They are stored there for later reference and do not have to be stored with the intermediate nodes.

Hierarchical algorithm executor

THierarchAlgHead = OBJECT (TAlgHead)

Algorithm executor which controls the sequence of algorithms combined in a hierarchical algorithm.

<i>Fields</i>	
fFirstCDCNode	Root CDC node of the local algorithm branch.
fFirstAlgNode	First algorithm node of the sequence.
fLastAlgNode	Last algorithm node of the sequence.
fScheduler	Algorithm sequence scheduler.
<i>Methods</i>	
SetDemandsCDC	Overrides to correctly access the corresponding items in the algorithm sequence.
SetParameterList	
TakeAwayProducesCDC	
Execute	Override to schedule the whole sequence.

The sequence is scheduled by the hierarchical algorithm scheduler object contained in the fScheduler field. Contrary to the consistency scheduler presented earlier, there is only one scheduler for the whole algorithm sequence. It does not derive the next algorithm from consistency evaluation, but retrieves it from the algorithm list.

Hierarchical algorithm scheduler

THierarchAlgScheduler = OBJECT (TAlgScheduler)

Object to schedule hierarchical algorithm execution

<i>Fields</i>	
fHierarchAlg	Reference to the hierarchical algorithm executor.
fCurrentAlgorithm	Current algorithm node being processed.
fLastAlgorithm	Last algorithm node to be processed.
<i>Methods</i>	
ExecuteNextAlgorithm	Override to schedule the next algorithm node in the sequence.

To hold a hierarchical algorithm executor, we do not want to change the structure of the algorithm node. The parameter passing mechanism (parameter source structure) should be maintained as well as the user interface (get info window). This requires the hierarchical algorithm executor to contain only one parameter list. It includes the parameters of all algorithms contained in the algorithm sequence. Upon execution, these parameters need to be distributed to their corresponding algorithm nodes.

The object responsible for this feature is the *extracting parameter source*, a subclass of the parameter source. Each algorithm in the sequence has its own source assigned permanently, contrary to the other parameter sources. Once the hierarchical algorithm receives a new parameter list, it is assigned to each of the sub-algorithm nodes's extracting sources, which then supply the correct parameters to the algorithm executor contained in the node¹.

Parameter extracting source

TExtractParamSrc = OBJECT (TParameterSource)

Object to retrieve a parameter subset from a given parameter list.

Fields	
fSuperParameterList	Parameter list of the hierarchical algorithm executor.
fFirstParameterIndex	Index of the first and the last parameter of the sub-algorithm node's parameters in the super-parameter list.
fLastParameterIndex	
Methods	
SetSuperList	Set a new parameter list of the hierarchical algorithm executor.

The internal structure of a hierarchical algorithm executor is shown in Fig. 4.9. Please note that on display is a single algorithm executor, not an algorithm node. The corresponding structure in Fig. 4.6 is the box in the centre labelled 'Algorithm executor'. The other node items (parameter sources and adjacent nodes) are not contained in Fig. 4.9.

Display

A tree node containing a hierarchical algorithm is displayed in Fig. 4.10 (node A4). It contains the sequence of marked algorithms in the left part of the picture. Although the special display is not necessary for operational reasons, the larger node picture indicates that the execution of this node may require more processing time than ordinary nodes. It also represents the fact that the algorithm was declared locally, that it may not be available in other documents without copying it first.

Hierarchical algorithms are also handled separately in the algorithm selection window. They are stored with the tree document they were defined in and do not use the file access mechanism which is needed by globally available algorithms (built-in and external algorithms). The list of available algorithms therefore shows an item with the locally defined algorithms. This item changes when the user selects another document for on-line algorithm query (see [22] for details of the algorithm query window).

1. The indirectly referenced list presented earlier may not be used here. The list returns a parameter list, whereas the algorithm node requires a parameter source for execution. Object Pascal does not allow multiple inheritance, we therefore need to define a new class, which is adapted to suit its purpose better.

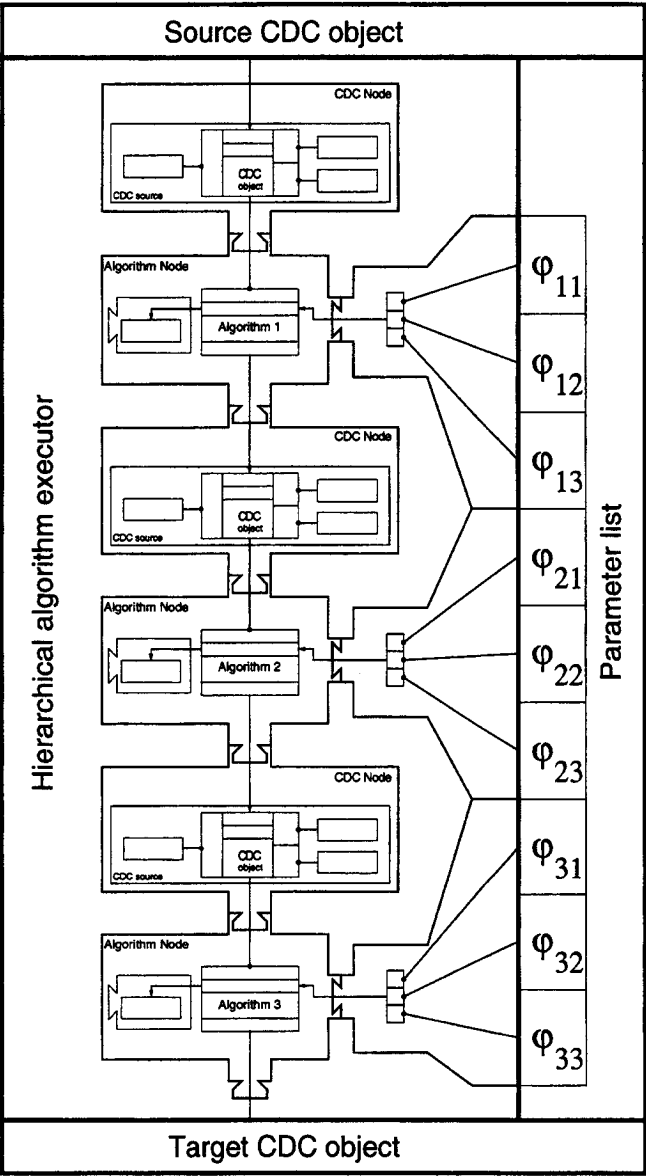


Figure 4.9: Internal structure of a hierarchical algorithm

The local nature of hierarchical algorithm definitions also requires an addition to the paste operation of algorithm nodes. If a branch containing a hierarchical algorithm node is pasted into another document, the hierarchical algorithm information has to be pasted into the target document's list of local algorithms. This allows us to re-use a successful design branch in other documents.

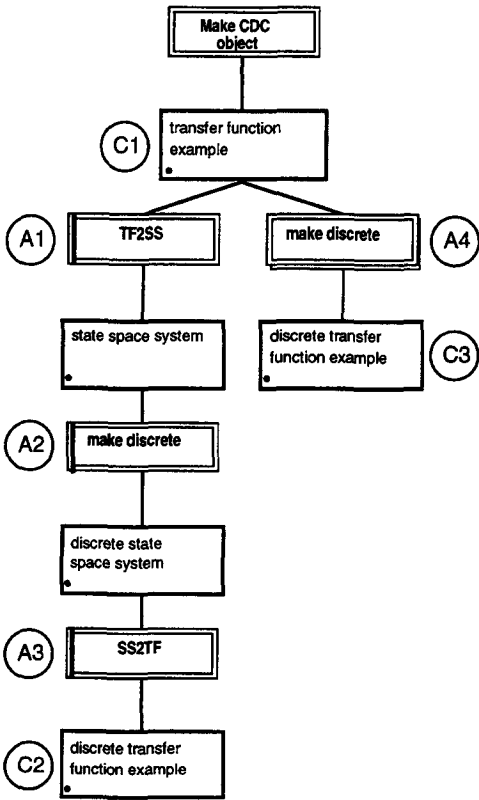


Figure 4.10: Hierarchical algorithm definition and node

4.5. Special Tasks

In the previous chapters we have presented the action tree structure and have shown some automated tasks which are available in the current version of Leporello. In the following we present a few extensions which indicate a possible direction in which Leporello’s functionality may be enhanced. All tasks described in this chapter are not yet implemented.

Key classes which are likely to be subclassed to provide extended features are the parameter source and the algorithm scheduler. We have already seen some implementations, two more are now presented.

4.5.1. Parameter Sweep

Task

The parameter variations presented earlier were always explicitly specified by the user. Each parameter set applied to an algorithm was entered from the keyboard parameter by parameter. However, there are situations where we encounter systematic parameter variations which follow a given pattern. If we want to execute an identification algorithm with model orders 3 to 7 we do not necessarily want to enter 3, 4, 5, 6, and 7 in sequence. Giving the bounds and an increment would be much easier.

Objects

A special parameter source which manages these variations automatically is very easily specified.

Parameter sweep source

```
TParameterSweep = OBJECT (TParameterSource)
```

A special parameter source to automatically vary parameters in a given range.

Fields	
fFirstSet	Range of the parameter sweep.
fLastSet	
fIncrementSet	Parameter increment.
Methods	
RequestNextSet	Override which calculates the next parameter set from the last set and the increment.

The user interface asks all three sets (first, last and increment) instead of one in the parameter input window.

4.5.2. Parameter Optimization

Task

A more complex task where a subclass of the parameter source is useful, is the parameter optimization problem. An optimization algorithm varies algorithm parameters and adjusts them according to some evaluations on the result. An optimization combines several of the tasks presented previously: parameters are applied to an algorithm on a given CDC object. After algorithm execution, a rating algorithm is applied to the result (performance index) and according to the value returned the new parameters are derived.

Objects

The optimization source subclass of the parameter source requires more complicated extensions. First of all, calculating the next parameter set cannot be as easily done as in the case of the parameter sweep source. Instead of just adding the increment to the last value applied we need to calculate a new parameter set from the results of the cost function.

The number of algorithms available to do this is quite large. Optimization algorithms exist for all kinds of situations and function properties. This variety leaves us with two implementation alternatives: We could decide on one special algorithm to be used in one subclass of the optimization source (other methods then require new subclasses), or we could leave the algorithm used to be determined at runtime as we do it with external algorithms.

The optimizer needs some information about the algorithm, its result (the target CDC) and the cost function. All these objects are already available within Leporello; the optimization source contains one of each.

Automatic parameter optimizer

`TParameterOptimizer = OBJECT (TParameterSource)`

<i>Fields</i>	
<code>fAlgorithm</code>	Algorithm to be optimized.
<code>fTarget</code>	Target CDC object to be used with the cost function.
<code>fRatingAlgorithm</code>	Cost function.
<i>Methods</i>	
<code>GetFirstParameterSet</code>	Initialization of the optimizer.
<code>RequestNextSet</code>	Evaluation of the cost function to determine the next parameter set.
<code>MoreSets</code>	True, if the desired accuracy is not yet reached.

Since we do not need to keep all intermediate results of an optimization process, the target CDC node only stores one CDC object per source CDC object and optimization. The information whether each parameter set results in a CDC object to be stored or not is given by the parameter source. The classes currently available contain this feature, they all require storage of each result.

Display

The user interface of the optimizer requires a dialog which allows the selection of a cost function and some other optimization parameters or even optimization routines. During runtime, parameter modifications could be displayed on screen and the user could interact to increase optimization speed, all depending on the chosen algorithm.

4.6. Hierarchical Systems

One task which has consequently not been mentioned until now is *systems interconnection*. Building a system by connecting sub-systems is a very frequent task in control engineering. As we have seen in Chapter 2.3, working with block diagrams to achieve this is one of the visual methods already quite well supported by commercial tools. A CACSD package which does not provide any systems interconnection and simulation facilities can not support all tasks necessary for a full controller design.

In its current state of development, Leporello does not. This is not a problem of incomplete concepts, this is mainly a problem of missing manpower and time. We will in this chapter describe how the concepts of hierarchical interconnected systems fit into the Leporello environment. The ideas expressed here may be regarded as an indication of further research in this direction.

4.6.1. Hierarchical Systems Class Definition

By hierarchical system we denote a system which is composed of one or more sub-systems which are interconnected. In general, the resulting system is nonlinear. Basically, hierarchical systems are systems like the others presented earlier. They are therefore defined as a subclass of the system class. The original CDC specification on page 65 already contained the necessary structures: the `fComponentList` was introduced to hold a list of constituting sub-objects. This list is not yet used in the implementation of the current system classes. To access it in a hierarchical system, access management methods need to be defined. The following discussion of concepts will show the necessity of some special access methods. The hierarchical system's parameter list needs to be mapped to the individual parameter lists of the sub-systems, a reference which can be implemented similar to the parameter extracting source introduced in the definition of hierarchical algorithms. We will not go into further details describing hierarchical system classes; the definition of control data objects is not part of this thesis.

Our interest lies primarily in the handling of hierarchical systems in the action tree.

4.6.2. Block Diagram Editor and Action Tree Operations

The preferred tool to create and modify hierarchical systems is a block diagram editor. We have previously shown two implementations of block diagram editors (Figures 2.12 and 2.13). If we assume a Leporello editor to be available, the blocks which are interconnected (and are therefore the components of a hierarchical system) can basically be divided into two parts:

- System blocks: These sub-systems are subject to some design work. Examples are a system model which has been estimated from measurements, or a controller who's parameters need to be optimized to satisfy some design requirements on a given plant. The design of these system blocks is reflected in their respective action trees, modifications are stored therein.

- Library blocks. These elements are too simple to be contained in an action tree of their own. They either do not contain any parameters which could be modified during a design, or their modification is only reasonable in the context of the block diagram. Examples of the first kind are sum, multiplication, etc. Example of the second kind are single gain values, noise amplitudes, etc.

A very simple and also a widely used block diagram structure consists of a controller, a plant, and unit feedback, as the one shown in the lower part of Fig. 4.11 (labelled 'Block diagram'). It contains two system blocks (controller and plant) and a library block (sum). Both system blocks were designed in the same action tree (Tree A in Fig. 4.11). This tree has its roots in measurements of the real plant. It contains the identification routines, the identified plant parameters, the controller design algorithms and the controller.

In order to fit this closed loop system into the Leporello concepts, we need an action tree node where it can be placed. The first idea is to insert it in the tree of the plant and the controller. We could decide to make it a descendant of the node which contains the identified plant, with a 'connection' algorithm node to produce it and the designed controller as one of the algorithm's parameters. Vice versa the hierarchical system could be made descendant of the controller node, with the plant as an algorithm parameter. Another option is to make the block diagram a descendant of both nodes. All these solutions share the same disadvantage: they destroy the tree structure of the action tree, making it a trellis. In addition to contradicting the tree concepts, this causes inconsistencies in most methods previously presented which rely on the tree structure. Another solution would be beneficial.

Let us take a closer look at the 'connection' algorithm mentioned above: It connects the controller, the plant model and some library blocks to form a new, the closed loop system. It is therefore justified to regard it as a CDC creation algorithm which is used in a new tree root. The hierarchical system is hence manipulated in its own action tree (Tree B in Fig. 4.11). However, its close links to CDC objects contained in other action trees require special access and modification techniques.

Numerical algorithms applied to block diagrams: Numerical algorithms applied to block diagrams (i.e. nonlinear systems) are rare in the engineering practice; today's research in this field does not show a mainstream, a generalized theory is not available at the time of writing. The most common algorithms applied to nonlinear block diagrams are simulation, linearization, steady state analysis, etc. These algorithms can be implemented like the algorithms applied to other CDC objects. Their handling and execution is done as described earlier (Chapter 4.1.1).

Topology changes: Changing the block diagram topology (connections or blocks contained) interactively is regarded as a modification algorithm. The algorithm node does not have any parameters, the resulting CDC node contains the modified block diagram. Since the block diagram passes a number of inconsistent states during interactive modifications, the user is responsible to declare the modifications to be

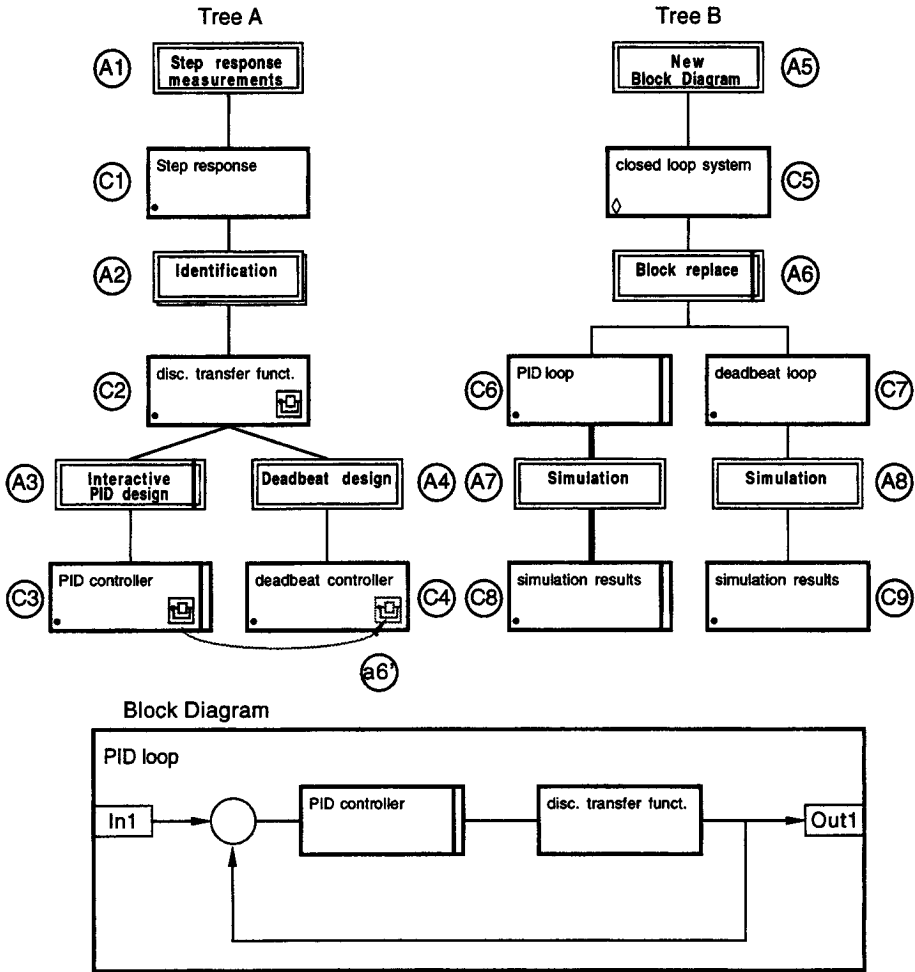


Figure 4.11: Leporello block diagram and associated trees

completed. This can be done by explicitly pressing an 'ok'- button to store the current block diagram in the tree, or by applying another algorithm (simulation, linearization) to the diagram.

Sub-system variations: One of the advantages of the action tree is its possibility to compare different designs and parameters. In our simple control loop example there were several controllers calculated. The PID controller was designed interactively with varying parameters (nodes A3, C3). As an alternative, a deadbeat controller was calculated (nodes A4, C4). The plant, which in the diagram is a linear system in transfer function representation, stands for a laboratory plant which was used to get some measurements (nodes A1, C1). With these, a linear model was derived, its parameters were identified by a series of algorithms combined in the hierarchical algorithm node

labelled 'Identification' (node A2). This linear model (node C2) was then used to design the controllers mentioned.

The different controllers should now be judged by simulating them with the model. Similar to the algorithm execution on a multi object node which was discussed in the action tree chapters, we can here simulate the closed loop system with the whole series of PID controller parameters available in the multi object PID controller node (node C3). The CDC node C6 ('PID loop') in tree B of Fig. 4.11 shows a possible interpretation of this situation. The node is marked being a multi object node. The series of CDC objects contained in this special multi object node is created by replacing the PID controller parameters by the whole series of parameters contained in the PID controller node (node C5) in sequence. By an appropriate design of the parameter mapping mechanism which assigns sub-system parameters to parameters of the hierarchical system, this is achieved with only minor adaptations of the tree storage mechanisms. The simulation algorithm (node A7) applied to the block diagram node in Fig. 4.11 (node C6) therefore is executed for all PID controllers contained in the 'PID controller' node (node C3) in tree A. The results of this simulation are all contained in the multi object node C8. If several block diagram elements originate from multi object nodes, the variation of all parameter lists easily leads to data explosion. The selection of the parameter sets used in parameter variations is left to the user. With the current tree manipulation mechanisms this is done by extracting the CDC objects as described in Chapter 4.2.1. Other user interface features to control this selection more easily are probably required to convince the user of the quality of this feature.

Sub-system exchange: If we not only want to compare the influence of sub-system parameter variations but also different sub-systems, the automatic multi object node creation mentioned above is not appropriate. If we exchange the PID controller with the deadbeat controller of node C4 of tree A, the structure of the block diagram changes. Its parameter list does not contain the PID parameters anymore. This place in the list is now occupied by the deadbeat controller parameters. The number and structure of the hierarchical system's parameter list changes, which clearly is a structural change. It is also not a priori known which nodes of tree A are to be used in a comparison of simulation results. They need to be chosen by the user. Both the structure change and the replacement selection require the sub-system exchange operation to be regarded as an algorithm. In our control loop example, the controller block was exchanged twice: first to contain the PID controller, and second to hold the deadbeat controller. The structural difference of the two resulting block diagrams is indicated by the second child (node C7) of the block replace algorithm (node A6).

A possible user interface feature is indicated by the block diagram markers of nodes C2 and C3 (block diagram icons on the lower right of the nodes): The block replace algorithm does not necessarily need to be called like other algorithms (algorithm selection and later execution), it could be initiated by dragging the marker to another node in the tree (arrow labelled a6'). The contents of this new node will then replace the original bearer of the block diagram mark in the block diagram.

Application

After the presentation of the concepts and features of Leporello, we will in this chapter describe the use of the Leporello prototype implementation to solve a selected control problem. The plant chosen is a Mettler electronic balance [5], one of the control experiments used in the laboratory courses for students at the automatic control lab of the ETH. The reader who is familiar with current research in automatic control will soon realize that the algorithms applied in this example are at most state of the art. In order to concentrate on the presentation of the Leporello working mechanisms and features we did not put much emphasis on the incorporation of methods currently under development in research.

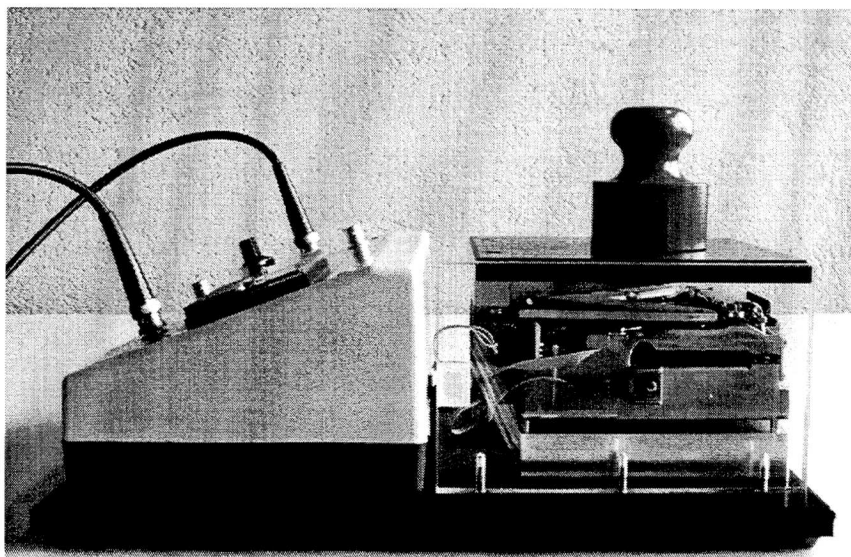


Figure 5.1: *Photograph of the laboratory balance. Weighting hardware is on the right, the box on the left contains additional electronics.*

5.1. Plant and Hardware Description

5.1.1. Electronic Laboratory Balance

The balance used with the Leporello prototype application is utilized in the student labs to teach typical control mechanisms and effects. In particular, there are three experiments available which show various controller implementations and require different information about the plant [41].

- Experiment A19: Continuous PI controller design based on step response experiments (Ziegler-Nichols, etc.).
- Experiment A20: Digital controller design using analytical design methods based on a transfer function estimate of the plant.
- Experiment A23: Lead-lag controller design using a measured Bode diagram for the design in the frequency domain.

Balances of the type discussed are used where the weight measurement needs to be precise and fast. A schematic display of the balance is shown in Fig. 5.2. The bar position is measured opto-electronically. It is controlled by the current in the inductivity. The weight is regarded as a disturbance which needs to be rejected. For small variations of the bar position, the relation between anchor current of the inductivity and the weight of the probe can be assumed to be linear.

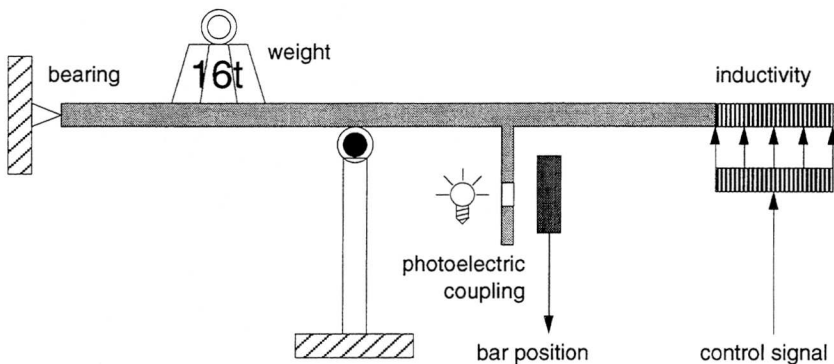


Figure 5.2: Schematic display of the laboratory balance

The controller to be designed is used to speed up the weighting process. It should therefore comply to the following requirements:

- After a change of weight the bar should be brought to its initial position as fast as possible with a minimal stationary error.
- Overshoot should be low. If there are overweight alarms connected to the balance, they could be triggered by a high overshoot signal.

Although a simple controller structure (PI-controller) does give satisfactory results, we will make use of Leporello's advantages to quickly compare different approaches to the problem.

5.1.2. Computer Hardware

The Leporello prototype application used in the following examples was programmed on an Apple Macintosh using MPW and the MacApp object library. The Leporello application and the Matlab server were installed on a Macintosh Quadra 700 computer, the real-time experiments were conducted on a Macintosh fx computer equipped with a LabNB analog I/O board from National Instruments which has been extended with a voltage converter to transform the ± 5 V of the board to the ± 10 V required by the plant. The software used for the real-time experiments was programmed in LabVIEW. The use of a dedicated computer for real-time experiments was possible due to the Leporello networking facilities. The whole experiment setup is shown in Fig. 5.3.



Figure 5.3: Experiment setup. The balance is connected via an analog I/O board to the Macintosh fx computer.

5.2. Working with Leporello

In this chapter we will try to give some ideas about how we think a user will work with Leporello. Most pictures shown are screen dumps or printouts of one of the packages used. However, most two dimensional signal plots were generated by saving the data from Leporello to a file. This file was then read into Matlab, where the graphs were created 'by hand'. This was necessary because the Leporello signal display facilities are not available in the current version, and because the LabVIEW procedures used instead (see Fig. 2.2 on page 19 for example) were designed to provide interactive graph handling features and not to print out nicely.

The numerical algorithms used in the balance example are all implemented in Matlab. They access one of the toolbox routines, mostly with some interface file which handles parameter conversion between the Leporello object parameters and the Matlab parameters. These Matlab algorithms are not mentioned in the text to come. We only describe the experiments and methods used. However, the reader who is interested in algorithm details can find the Leporello algorithms used in the action tree pictures. The algorithm nodes are labelled with the Leporello name of the algorithm. A list of Leporello algorithms, the package they access, and the package command file they execute can be found in Appendix A.

5.2.1. Leporello User Interface

Before we concentrate on the balance problem, we will give a short overview of the Leporello user interface features which are not related to the action tree display (and which therefore have not yet been described in the previous chapters).

Communication setup: The limited number of algorithms built into Leporello requires communication facilities to other packages as mentioned in Chapter 3.1. For each of the packages accessed (Matlab and LabVIEW) we provide a communication interface based on the Program-to-Program Communication Toolbox built into Macintosh System 7. The connections are controlled in the respective window in each packages. External links can be established on the local computer or on a remote server accessible on an AppleTalk network. All external communication handlers are set up to automatically restore the communication after its failure. The connection is then re-initiated from Leporello, there is no need for manipulations on a remote computer.

Action tree manipulation: The action tree access and modification tasks described in Chapter 4 are all controlled from the toolbar available in the action tree document window (Fig. 5.4). Tool-buttons are disabled if an action is not possible (mostly, if no appropriate node is selected). Disabled buttons can still be pressed. A message window is then shown, which explains the reason why the desired action is temporarily not available.

The check-boxes below the tool-buttons allow to set some task preferences. If one of them is not checked, a dialog box is displayed which allows the user to select the

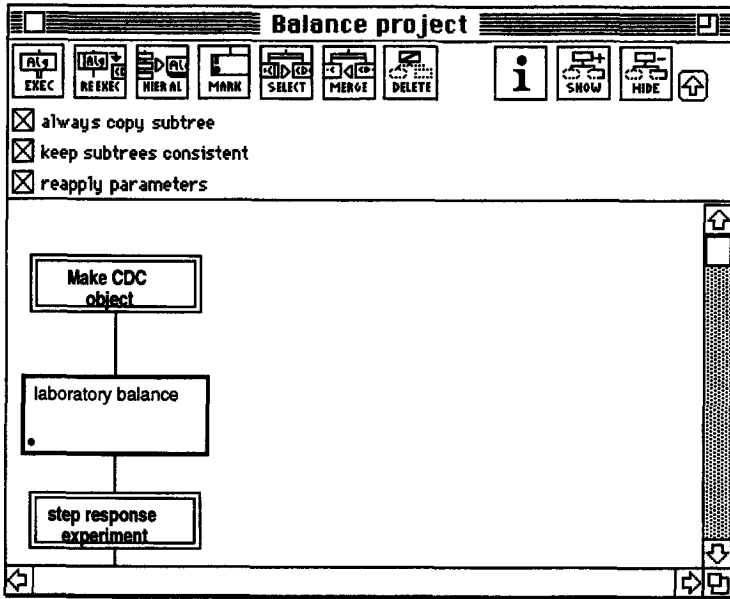


Figure 5.4: Leporello document window with toolbar, preferences and tree display

desired task behavior. The arrow button to the right of the toolbar shows or hides the preferences area.

The action tree display itself allows a number of interactions. Pressing the command and arrow keys exchanges the position of the selected node with its neighbor, pressing command and clicking on a node toggles its mark. If the mouse button is pressed during mouse movements starting at a node (dragging), nodes which are directly related to the first node are highlighted. Dragging between an algorithm node and one of its twin children sets the target to this child, dragging between two twin nodes selects the first twin and sets the target to the second. This allows an immediate node merge in the dragging direction.

Node information: Full information about a node's contents will be displayed by the 'Get Info'-button or by clicking the same node twice (double-click). The window, which shows this information for one of the nodes containing identified models in the example to come, is shown in Fig. 5.5. Name and purpose fields of the CDC object can be modified, all other information boxes can be shown or hidden. Algorithm node information is displayed similarly.

Another Leporello window (algorithm selection window) has already been shown on page 77 (Fig. 4.1). Parameter windows are extensively discussed by Kolb in [22].

name and usedAs:			
name of control data object disc_balance.sl.sp.20ms			
object is used as: model of laboratory balance			

input/outputs:			
system inputs		system outputs	
name:	SI-unit:	name:	SI-unit:
u1: inductivity current		y1: balance position	

fixed Properties:	
parametric systems	state space system
Sampling	discrete
Inputs	single input
Outputs	single output

tested properties:		CDC 1:		CDC 2:																															
controllable		isTrue		controllable																															
asympt. stable		isTrue		asympt. stable																															
isTrue				isTrue																															
tested ratings:		system order: 2		system order: 3																															
parameters:		A: <table border="1"> <tr><td>0.1576</td><td>-5.3490</td></tr> <tr><td>0.0098</td><td>0.9313</td></tr> </table> B: <table border="1"> <tr><td>0.0096</td></tr> <tr><td>0.0001</td></tr> </table> C: <table border="1"> <tr><td>2.7246</td><td>2581.4387</td></tr> </table> D: <table border="1"> <tr><td>0</td></tr> </table> Xo: <table border="1"> <tr><td>0</td></tr> <tr><td>0</td></tr> </table> TSamp: 0.0200 [s]		0.1576	-5.3490	0.0098	0.9313	0.0096	0.0001	2.7246	2581.4387	0	0	0	A: <table border="1"> <tr><td>0.0348</td><td>18.7600</td><td>-97.0358</td></tr> <tr><td>0.0003</td><td>0.1473</td><td>-5.3023</td></tr> <tr><td>1.522e-5</td><td>0.0102</td><td>0.9374</td></tr> </table> B: <table border="1"> <tr><td>0.0003</td></tr> <tr><td>1.522e-5</td></tr> <tr><td>1.798e-7</td></tr> </table> C: <table border="1"> <tr><td>0</td><td>0</td><td>1.611e+6</td></tr> </table> D: <table border="1"> <tr><td>0</td></tr> </table> Xo: <table border="1"> <tr><td>0</td></tr> <tr><td>0</td></tr> <tr><td>0</td></tr> </table> TSamp: 0.0200 [s]		0.0348	18.7600	-97.0358	0.0003	0.1473	-5.3023	1.522e-5	0.0102	0.9374	0.0003	1.522e-5	1.798e-7	0	0	1.611e+6	0	0	0	0
0.1576	-5.3490																																		
0.0098	0.9313																																		
0.0096																																			
0.0001																																			
2.7246	2581.4387																																		
0																																			
0																																			
0																																			
0.0348	18.7600	-97.0358																																	
0.0003	0.1473	-5.3023																																	
1.522e-5	0.0102	0.9374																																	
0.0003																																			
1.522e-5																																			
1.798e-7																																			
0	0	1.611e+6																																	
0																																			
0																																			
0																																			
0																																			

Figure 5.5: Information window of node C1 of Fig. 5.11 which contains the identified model of the balance, calculated from the Bode diagram

5.2.2. Balance Measurements and Interactive Controller Design

Step response: The first steps on our way to a successful controller design are selected to become accustomed with the plant behavior in characteristic situations. The first experiment applied to the balance is the measurement of its step response. The Matlab plot of these measurements is displayed in Fig. 5.6. At first sight, one dominant pole on the negative real axis can be assumed, some faster dynamics can be guessed from the behavior in the first sampling intervals.

Bode diagram measurements: More insight into plant dynamics can be retrieved from the Bode diagram. For its measurement we designed a LabVIEW interactive tool which

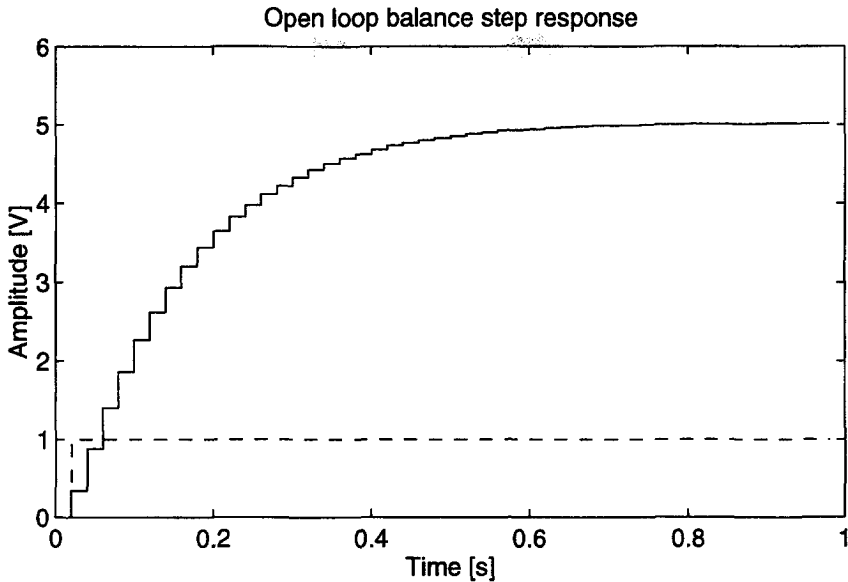


Figure 5.6: Balance step response. Measurement of the bar position after a change in the anchor current. Sampling time: 20 ms

allows to record the frequency response of a plant by sequentially feeding sine waves in the desired frequency range to the plant and measuring the plant responses. Since we assume that this response is approximately sinusoidal, we calculate amplitude and phase of each point at the peak amplitude of the Fourier transform of the input and the output signals with removed trends. The VI front panel is shown in Fig. 5.7. The parameters are asked from the user in Leporello and are sent to the LabVIEW VI. Each measured sine wave is shown, 'abnormal' results (too much noise, no signal, etc.) can be detected immediately and the experiment can be repeated with improved settings. After the responses over all selected frequencies are recorded, selected measurements can be repeated to remove erroneous samples or to achieve increased precision.

Interactive lead-lag controller design: Based on the experimentally obtained Bode diagram we can use the interactive controller design tool shown in Fig. 2.4 on page 22.

The required Bode diagram margins are taken from the description of Laboratory Experiment A23:

static gain:	20dB
phase margin:	$\phi \geq 100 \text{ deg}$
critical frequency:	$10 < \omega_D < 20 \text{ s}^{-1}$

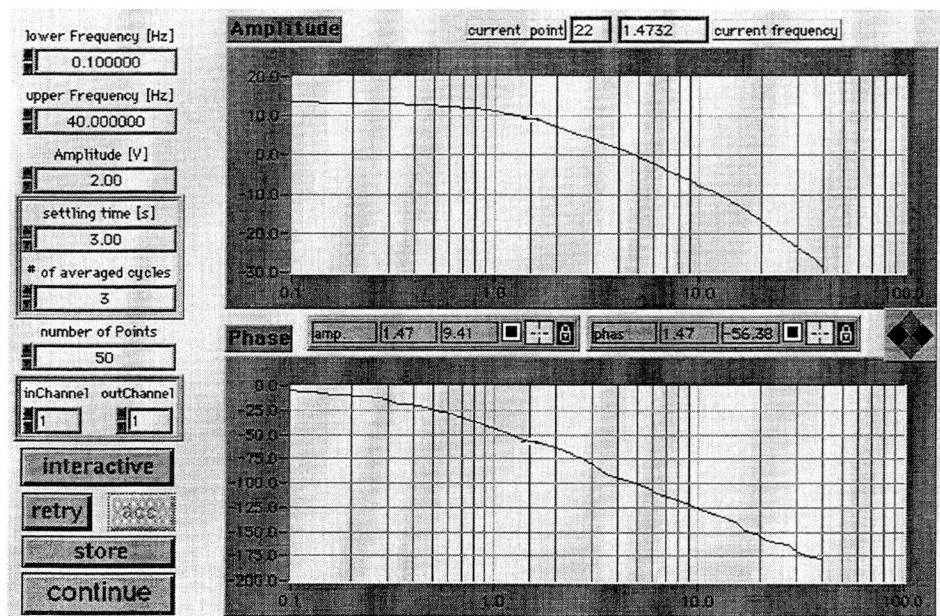


Figure 5.7: LabVIEW VI front panel of the swept sine tool applied to the balance

The lead-lag parameters are now tuned interactively by the user until the above requirements are met. An exemplary controller is expressed in the following equation:

$$G(s) = k \cdot \frac{(1 + T_1 s)(1 + T_2 s)}{(1 + \alpha_1 T_1 s)(1 + \alpha_2 T_2 s)}$$

with $T_1 = 0.1$, $\alpha_1 = 0.75$, $T_2 = 0.16$, $\alpha_2 = 9.68$ and $k = 3.3$.

This transfer function is transformed into discrete state space representation assuming a sampling interval of 20 ms. A faster implementation (10 ms) is possible on the configuration used (Macintosh fx, LabVIEW 3), but the faster rate does not allow any interactive user operations.

Looking at the controller poles we see, that the sampling time and the fastest pole are conflicting. The continuously designed controller should be sampled at approximately ten times its fastest time constant or faster. We therefore try the following:

- complete lead-lag implementation at 20 ms
- complete implementation at 10 ms without user interaction
- lead-lag controller at 20 ms after model reduction
- pure lag controller at 20 ms

Controller implementation test: All four controllers were tested on the real plant. The measured step responses of the closed loop systems are displayed in Fig. 5.8. We see

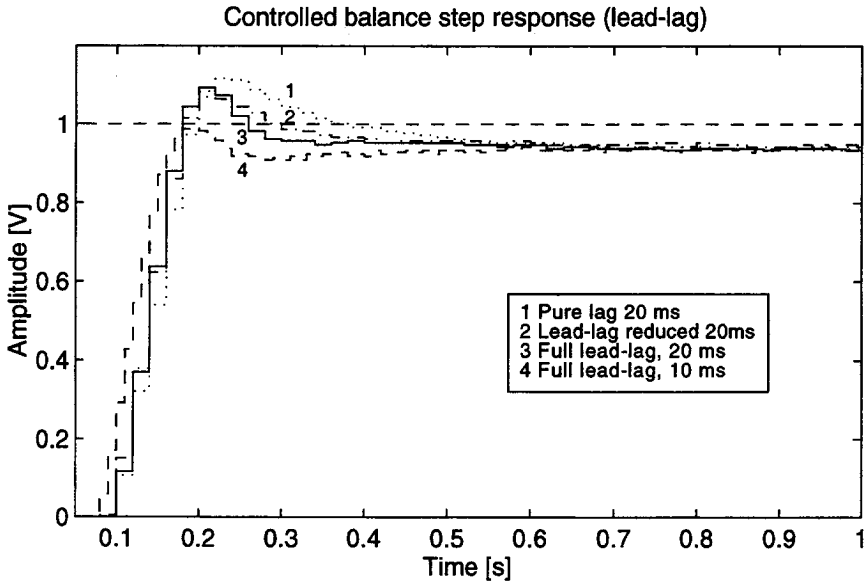


Figure 5.8: Step responses of the plant controlled by lead-lag controllers.

that the idea to just remove the faster pole resulted in a clearly slower controller. The reduced controller is still slower than the full implementation, its overshoot is also less than with the pure lag. Although we sample too slow, the full controller containing the fast pole still gives the fastest responsiveness.

Doubling the sampling frequency does reduce raise time and overshoot, the steady state position is reached much later.

All four controllers tested did comply to the initial specifications. However, the question may be raised whether stronger specifications could be reached by applying some of the more complex design algorithms. We will therefore not only continue to investigate the methods used in the laboratory experiments, but also try out other algorithms accessible in the Leporello environment.

Leporello features: The tree which was used for the lead-lag design is shown in Fig. 5.9. The initial node contains the balance analog I/O channels used for measurements and control. It serves as root for the step response and the Bode diagram measurements. We include all analysis algorithms (plot, etc.) as an example of a tree 'under construction'. To reduce tree complexity, these nodes are removed in most other diagrams shown on the next few pages.

One very useful Leporello feature is not visible in the tree picture: once an algorithm is applied to a data node (*i.e.* all the step response nodes), it is easily reapplied to all other nodes which are to be compared. This guarantees equal parameter settings (equal step height and length) and eases algorithm selection without switching windows and repeatedly searching algorithm lists.

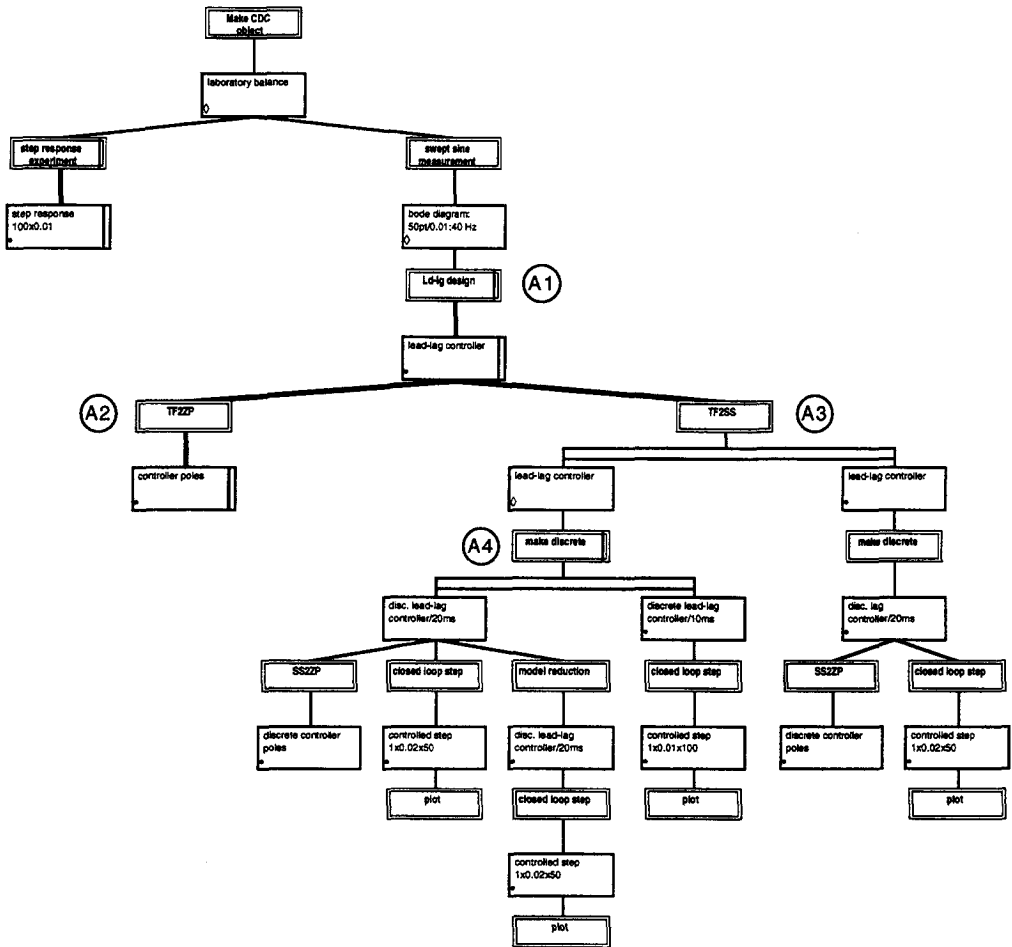


Figure 5.9: Lead-lag design tree including Bode diagram measurement.

Another feature shown is the use of a separate target node for different algorithm results. The branch containing the full controller with the 10 ms sampling time (subtree to algorithm node A4) was created by removing the target mark from the node containing the 20 ms sampled controller. The discretization algorithm was then executed, which created the new branch.

The same technique was used to create the pure lag controller (right subtree to algorithm node A3): The lead-lag design algorithm (node A2) was executed the second time after the faster and the reduced implementations were done. The pole/zero calculations (node A2) were updated automatically, the transformation to state space created the second twin branch since we removed the target mark from the first CDC node prior to the execution. The whole subtree of node A4 was therefore not repeated automatically, the algorithms were re-applied individually.

5.2.3. Identification of a Continuous-time Model

Most controller design algorithms require the presence of a plant model of some sort, found through the use of appropriate identification algorithms.

Least squares transfer function estimation: Since we are already in possession of Bode diagram measurements, we can first apply an identification algorithm which acts on the frequency response of the system. This algorithm estimates the transfer function of assumed order using the least squares method. From the Bode diagram and the open loop step response we expect to receive model orders of 2 or 3, with no or one zero. All four combinations are evaluated, by visual comparison of the Bode diagrams calculated from the estimations we keep the model with order 2 and one zero, and the model with order 3 and no zero. Both models are kept for further evaluation during controller design. They are first transformed into discrete state space description using a sampling time of 20 ms.

The measured Bode diagram with the diagrams calculated from the transfer function estimations is shown in Fig. 5.10.

Leporello features: The tree as it is displayed in Fig. 5.11 was created mainly in one single tree containing all identified models. After the selection of the models men-

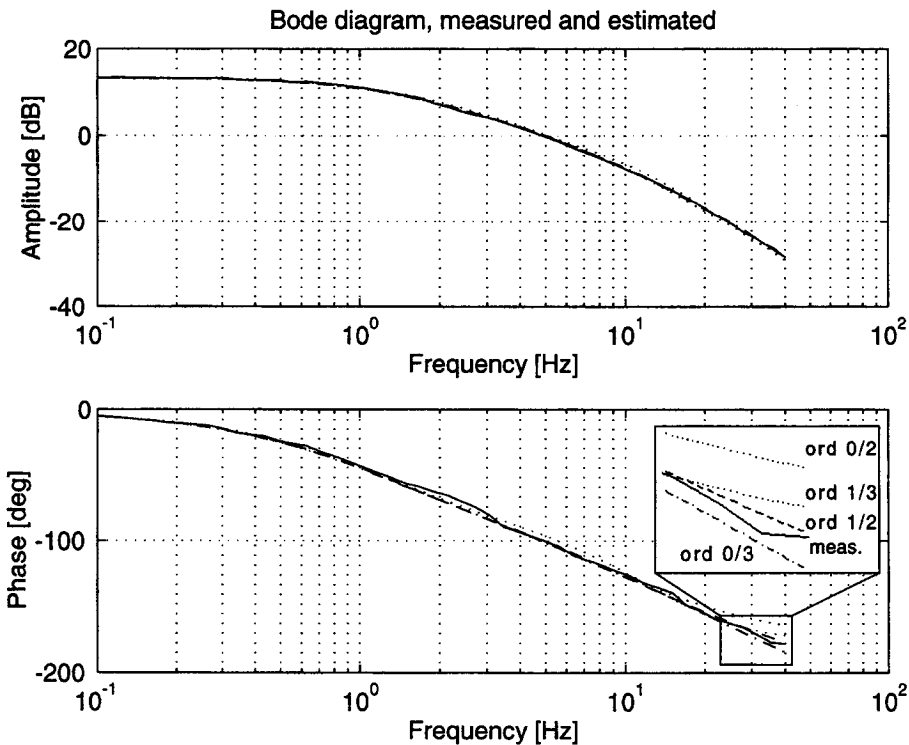


Figure 5.10: Measured Bode diagram and diagrams calculated from estimations

tioned above, the tree was divided into two subtrees, one containing the discarded models and the related results. If we continued to work on this tree, the unused branch would be clipped drastically.

In the far right branch in the tree picture, the selected models are transformed to discrete state space. This node (node C1) is later copied into a new document to keep the tree pictures within the size of a page.

5.2.4. State Feedback Controller Design

Deadbeat controller design using pole placement: We start experimenting with the second order model derived from the Bode diagram. Since the state feedback controllers all vary in size with the model order, we can not automatically reuse algorithm parameters for models of different orders.

The first controller we would like to design is a deadbeat controller. This type of controller imposes deadbeat behavior on the controlled plant; the number of steps required to bring the error to zero is equal to the system order. In our case, we should therefore be able to eliminate an error in two steps, *i.e.*, within 40 ms.

A discrete time system with deadbeat behavior has all its poles at the origin. The desired poles of the control system were accordingly all placed at zero (The state feedback controller can only force equal poles in the number of system control inputs. One of the poles is hence slightly displaced). The simulated step response in Fig. 5.12 shows perfect results, the setpoint is reached in two steps as desired.

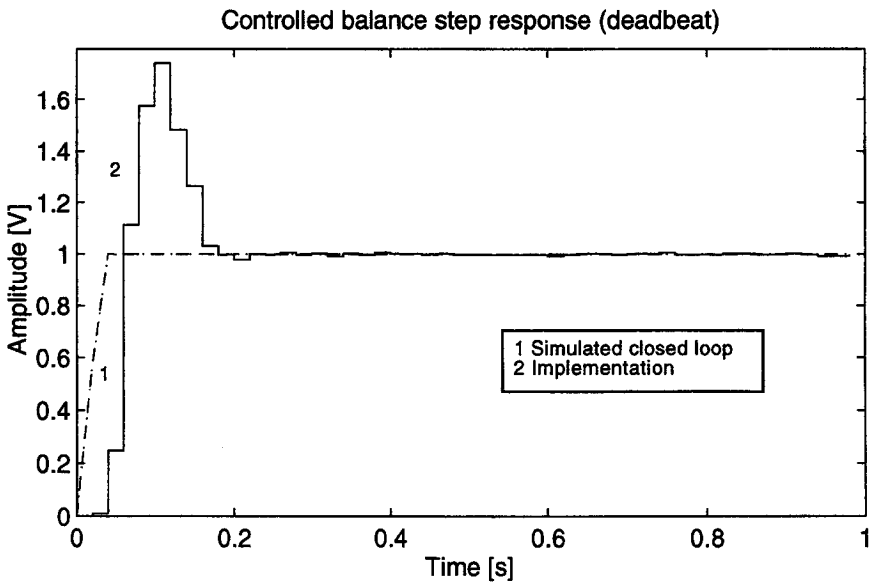


Figure 5.12: Simulated and measured step responses of the closed loop deadbeat system

Since we can not measure a second state at our balance plant, we need to design an observer. The observer poles are also placed in the origin with small displacements, the step response simulation shows the same perfect results we expect (the curve is not shown in any of the graphs, we kindly ask you to believe this).

Controller test and implementation: The most important test is the controller implementation. The sampling time of the controller is 20 ms. We see in the measured step response in Fig. 5.12 that the raise time of this controller is faster than any of the lead-lag controllers's. The stationary error is smaller, but the increased performance is payed by a much higher overshoot. We will try to handle this effect by a LQR-design.

LQR-design: The popular LQR-design is also applied to the second order model of the balance derived from the Bode diagram. This algorithm is well suited to be used in the Leporello environment: its design parameters, matrix Q for state weighting and R for control signal weighting are subject to engineering experience and may therefore cause several retries of the design ('fine-tuning').

The initial settings (randomly chosen $Q = I$, $R = 1$) do not return appropriate results. The step response of the controlled system derived from these parameters is not faster than the uncontrolled plant, yet we do get the desired stationary gain. Different parameter settings do not show the desired improvement (Fig. 5.13, curves A).

The reason is the difference in magnitude of the state signals and the control signal. Since the chosen state space representation does not have any physical meaning, we may as well balance it prior to the controller design. The same algorithms applied to the balanced balance representation show immediate improvement even with the initial settings of both design parameters Q and R (Fig. 5.13, curves B: $Q = I$, $R = 1$ and $Q = 0.2 \cdot I$, $R = 1$). The step response of the resulting closed loop control system is a little slower than the one with the deadbeat controller, we can expect less overshoot in the implementation.

Controller and observer implementation: As with the pole placement algorithm, the state feedback designed by the LQR algorithm requires an observer to be implemented with the balance SISO system. The observer is also designed to have its poles at the origin, i.e., to be as fast as possible. The test simulation shows the same behavior as the original state feedback system (not shown in any graph either).

The implementation of this controller with the same sampling time of 20 ms does show much less overshoot than the deadbeat controller. Little modifications in the Q weighting matrix eliminates overshoot, the measured step response of the controlled system (Fig. 5.13, signal C) is now very similar to the simulation.

The most successful experiments are afterwards repeated with the third order model with comparable results. These approaches are contained in the action tree in Fig. 5.14.

Leporello features: To reduce the size of the example tree, the models derived from the measured Bode diagram are copied to a new document which is then used for the controller design and verification (Root in Fig. 5.14).

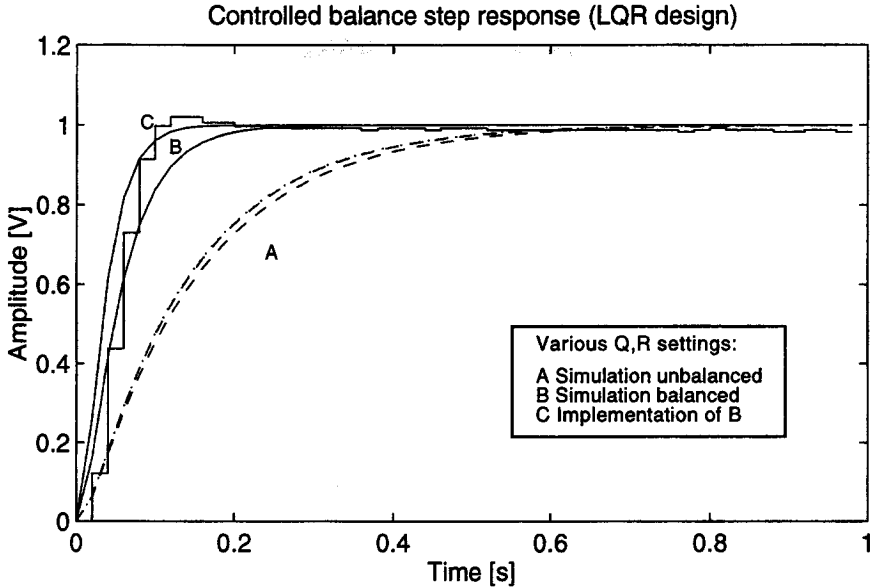


Figure 5.13: Step responses of the state feedback controllers designed using the LQR methods

The controller design subtree structures (one for the deadbeat and one for the LQR) show the increasing complexity of the controller from left to right. For each of the design objectives, the pure state feedback simulation, the simulation with the observer, and the implementation with the physical plant are contained in separate branches of the tree.

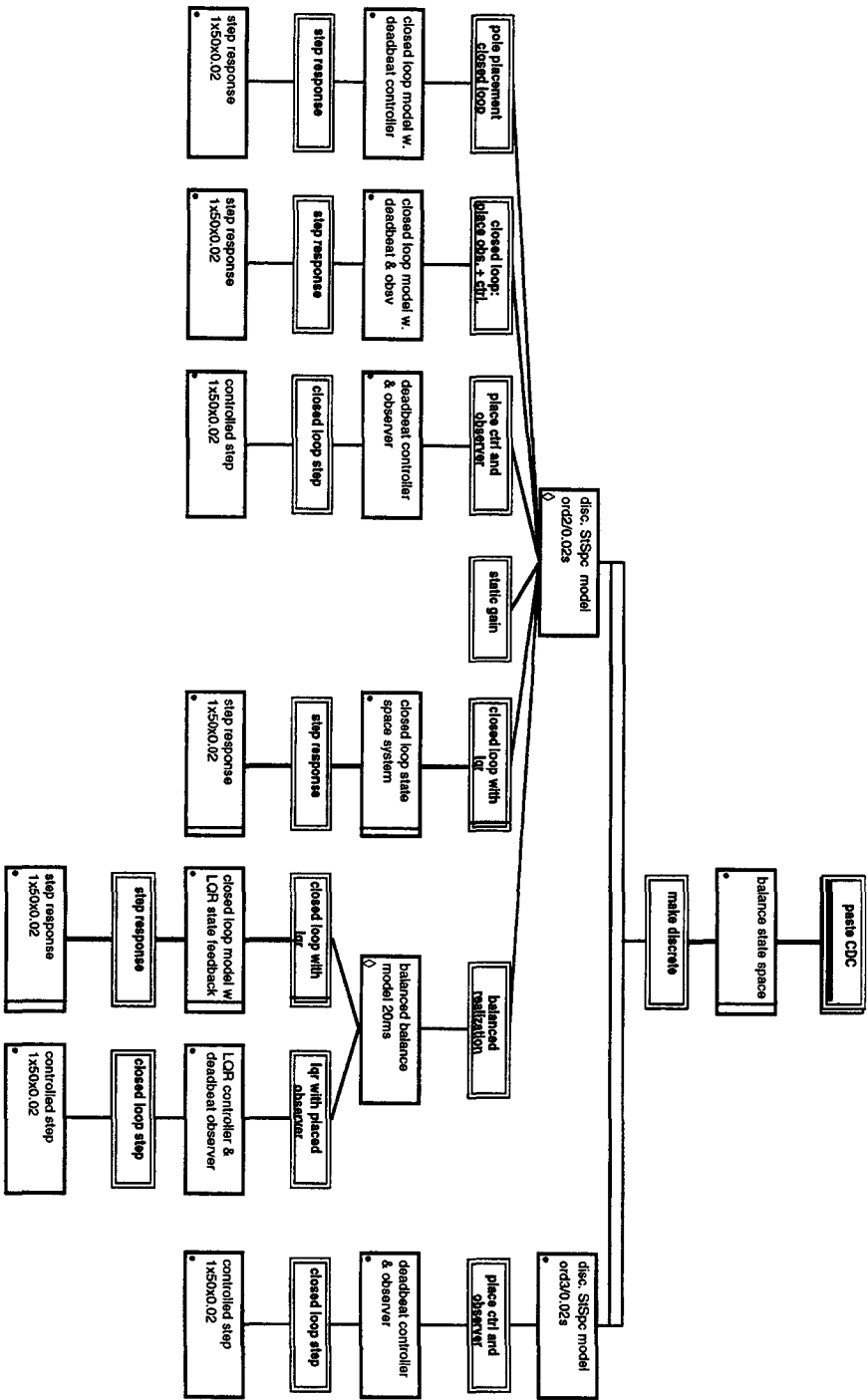
This structure is not optimal. If the Leporello block diagram editor were available at the time of writing, the three branches would be contained in the tree of the closed loop block diagram.

5.2.5. Discrete-time ARX-model Identification

It is well known that state feedback design requires a very adequate model of the plant. If we consider the fact that the sub-optimal results presented up to now may be caused by an error in the model, we may get better results by an improvement of the identification. We will in this approach try a discrete ARX model identification determined using the LS method and pseudo random binary sequence system responses.

Plant excitation experiments: An experiment very popular in system identification is the measurement of the system response if fed with random binary signals (signals which jump between two voltage levels at randomly chosen time instances). These signals are known to excite the plant in a broad spectrum (persistent excitation) and are therefore well suited as a starting point for identification methods. Three signals with different bandwidths are generated and used to feed the plant.

Figure 5. 14: State feedback and observer design and implementation tree



The experiments applied to the balance include all three signals at a sampling rate of 10 ms and the second signal only at a sampling rate of 20 ms.

ARX-model identification: Based on the system responses to the random binary signals, a discrete transfer function is estimated. The algorithm used automatically tests a range of numerator and denominator polynomial orders given by the user. Each possible pair of orders results in an optimized transfer function. They are all simulated with the same input signals, and the difference between the simulation and the measurements results in a performance index. The best function is returned.

This algorithm is first tested on the '20 ms'-signal. To obtain adequate results, the data must have zero mean value. We therefore first remove any offset from the measurements. We then apply the automatic ARX identification with both polynomials evaluated at orders one to five. The result is a fourth order system in the identification toolbox Theta format. This format is then transformed into a discrete state space description.

Inspecting the zeros and poles of this system we see two near pole/zero cancellations. It is therefore apparent that similar modeling could be achieved by a second order system. The ARX identification is applied again, this time only with the desired orders two of both polynomials. The resulting distribution of poles and zeros shows the cancellation of two of the poles. The other ones are left almost unchanged.

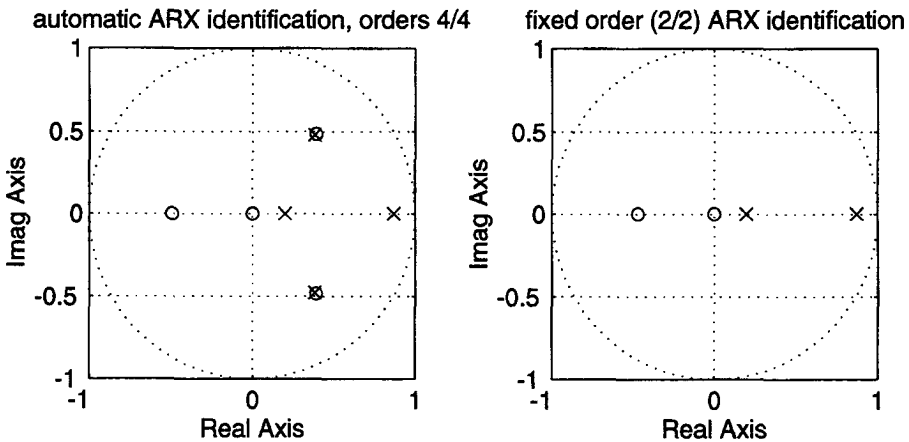


Figure 5.15: Poles and zeros of the automatically identified system and the one with fixed orders

The same experiments are then applied to the faster measurements. Again we receive higher order systems with pole/zero cancellations, the final systems are all second order.

Controller design: The design of the same deadbeat and LQR controllers which were previously (Chapter 5.2.3) derived from the continuous identification based on the Bode diagram did not produce any improved results.

Leporello features: The tree which contains the ARX experiments shows some additional features of Leporello not previously used in the example.

The tree shown in Fig. 5.17 is based on the one shown in Fig. 5.11: the resulting document is to contain all identification results. The previous results are hidden, the root of the subtrees fully shown in Fig. 5.11 are here left with a fat bottom line which indicates the hidden subtree. This not only simplifies the tree on the screen, it also significantly speeds up screen redrawing.

The identification experiments were all first applied to the one signal sampled at 20 ms. The algorithm sequences used there were then applied identically to the faster signals. To repeat the whole branches more easily, we created two hierarchical algorithms from the branches rooted in node C1. They were then applied to the rest of the signals contained in node C2.

Finally, all identified systems were simulated with the last of the test signals (Proper identification would require a different signal for model validation. Choosing one of the input signals we only violated this rule for one of the systems). The 'compare' algorithms shown in the tree diagram call a LabVIEW instrument which allows interactive comparison of not only the signals themselves, but also of their differences, cross correlations and other operations. Interactive signal comparison is not yet possi-

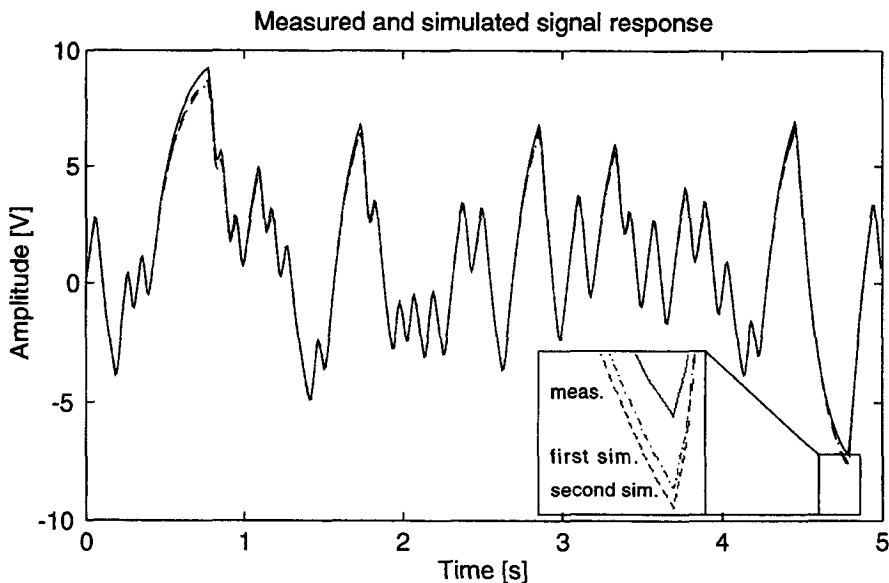


Figure 5.16: System response to random binary signals, measurements and simulations



Figure 5.17: Action tree of the ARX identification

ble in this printed version, we can therefore only display them in a static graph. Fig. 5.16 shows the measurements and two of the simulations.

5.2.6. Remarks

The examples shown on the last few pages do only represent a small amount of the work done with Leporello during these experiments. The application did indeed motivate many different parameter settings, algorithms and variations which would not have been used by the author if only confronted with Matlab and a pile of manuals.

The action trees displayed only reflect the static result of the work. Their real advantage is during work, when they are used as a dynamic tool. Different results are easily compared, and algorithms are quickly applied to other versions of measurements or results.

Working with Leporello clearly proved the advantages of the concepts proposed, I really enjoyed it (if you allow me this very personal remark).

Conclusions and Further Work

The contents of this thesis can be summarized as follows:

- We have briefly shown how scientific visualization is used today and what further development and advantages may be expected in the future for its application to automatic control.
- We have presented visual programming languages in general and specific implementations to be used in algorithm development and controller implementation in the field of automatic control.
- We have presented the Leporello tool which provides a new graphical representation of the control systems design process as a whole.

The conclusions to be drawn from the different aspects handled up to now are summarized in this chapter. We will then give our ideas about further work and research possible in the directions indicated in this thesis.

6.1. Conclusions

The graphical methods and examples presented in the different parts of this thesis all served the same ultimate goal: to reveal more information about the problem being solved with the computer. The graph which shows data trends, the visual language which presents data flow dependencies and the Leporello action tree which displays a complete controller design all help to present information which is otherwise hidden from the user. The contribution of this thesis is summarized in the following key points:

6.1.1. Scientific Visualization

- The interactive creation of higher dimensional data display turns out to be an important tool in the analysis of signals and systems.

- Many algorithms benefit from an interactive graphical user interface. The display of important internal state variables or the development of iterative or recursive results does offer additional insight into algorithm behavior. Interactive modifications of variables controlling algorithm behavior allow the quick investigation of alternatives. Engineering parameters otherwise chosen intuitively or by experience can be found or tuned faster and in a more intriguing manner.

6.1.2. Visual Languages

- In selected areas of control engineering, visual languages do ease software development. The most obvious case is the use of a block diagram editor and simulation language which allows real-time code generation and therefore facilitate the prototype implementation of control systems.
- Apart from their special advantages in control engineering, visual languages speed up software development in general. If done on the computer using one of the visual languages presented in this thesis, design sketches can be turned into executable software with only little more effort. Changes in these sketches are instantly reflected in the program itself. Drawings and implementation are always in a consistent state.

6.1.3. Visual Control Systems Design in Leporello

- The Leporello package presented in this thesis provides a fully graphical user interface for a complete control system design cycle. This user interface is based on the graphical display of the action tree, a tree description of the design history which contains all data and information which have been used in the design. Besides being a data base of all intermediate results of the design process, the action tree is also displayed on screen and as a part of the user interface allows interactive modification of any part of the design.
- The complete information stored for each algorithm allows an automatic repetition of experiments and algorithm sequences on modified data. The influence of alternate algorithm parameters or plant parameter modifications can thus be investigated easily.
- The incorporation of commercial software packages for algorithm implementation and their access from within Leporello provides an unified user interface for all external packages used. The Leporello user does not need to handle different data formats and command syntaxes. By the integration of Matlab and LabVIEW, Leporello benefits from their advantages: the large number of available algorithms and the ease of extension in Matlab and the graphical user interface and the visual real-time language of LabVIEW.
- Strict data and algorithm classification ease the search for the proper method which is to be used in a specific situation. The advantages of this approach are described and summarized by Kolb in [22].

- The Leporello prototype implementation has successfully been applied to the controller design of an electronic balance (described in this thesis) and of a servo system (described in [22]).

6.2. Further Work

Throughout this thesis we have often mentioned possible extensions and further research and development. These points are summarized in the following.

The research in scientific visualization and visual languages is in process, both subjects are far from being investigated to their full extent. Results from both fields are worth being applied to automatic control problems.

6.2.1. Leporello Concept Extensions

Apart from the technical application improvement, which is necessary anyway at this stage of development, several concept extensions have already been proposed in the run of this thesis. The most important and significant are:

- *Parameter administration.* Two additional parameter handling facilities have already been proposed in chapters 4.5.1 and 4.5.2: the parameter sweep and the branch optimization. Other methods to generate algorithm parameter sets can be thought of, which may require an improved parameter administration concept.
- *Hierarchical systems concepts.* This is where most research work still needs to be done since engineering jobs contain some part where things are put together. In control engineering, this is done using a block diagram editor or a similar tool. Although we did present the concepts to represent the 'system assembly' in an action tree, this is only the first step. There must be better solutions to this common engineering problem. Exchanging different parts of a design and watching the effects on the overall behavior of the system is a basic experimenting technique frequently used. An improved action tree could maybe handle problems of this sort. Investigations in this direction may be very fruitful.
- *Package incorporation.* Many of the tasks described could be achieved more efficiently if Leporello had closer access to the internal structures of the external packages used. The current implementation can only execute algorithms in one step, interactive modifications on an algorithm user interface can not be traced. We think that the incorporation of the Leporello user interface and concepts in one of the mathematical packages currently in use in the control systems community would be beneficial to both.
- *Design data base.* In the beginning of the project, the incorporation of a data base into the Leporello environment was planned. Successful designs and typical problems could be stored there and could be retrieved systematically. The incorporation into Leporello would then allow the re-execution of solutions found in the data

base similar to the hierarchical algorithms presented. Project descriptions could also contain many more comments and textually added information as well as other related documents (plant pictures or even project submissions and invoices).

The direction pointed to by the Leporello tool does still show some great improvements of user interfaces not only in CACSD, but also in other fields of engineering.

6.2.2. Future Leporello Development

Whether the Leporello project will be continued is not fully our decision. However, we do consider it interesting and bearing challenging aspects. Since the Leporello prototype was designed mainly to prove the usefulness of the action tree and data abstraction concepts, further development on the chosen path will probably prove not to be the easiest solution. Elaborate tools which are available today should be used to create a portable, more efficient implementation of Leporello. To produce a commercial tool or to continue research in this area, at least partly redesign will be necessary.

Still, we hope that the ideas expressed throughout this thesis will find their way to the control community one day.

Appendix A: List of available Leporello algorithms

The following tables contain a list of currently available algorithms. These lists are thought to provide the necessary information about the algorithms appearing in action tree displays in Chapter 5.2.

The columns contain algorithm names as they are used in Leporello, the external package which executes the algorithm, the interface file which is called by the Leporello algorithm and the file which contains the algorithm code.

Table 2 contains a column which indicates whether the algorithm provides an interactive user interface, whether real-time and I/O facilities (an I/O card with on-board timer) is required and whether the user interface allows local storage of algorithm results for later comparison (these properties are currently available only in LabVIEW algorithms). The last column of Table 3 indicates, whether the indicated algorithm is a property (CDC object does or does not have the tested property) or a rating (numerical information about the tested CDC object) algorithm.

Leporello name	Pk ^a	interface called	algorithm file	pr. ^b
ARX identification	Mt1	LARX	arx	
automatic ARX identification	Mt1	autoARX	arxstruc, selstruc, arx	
balanced realization	Mt1		balreal	
bode diagram	Mt1	lbode	bode	
bode diagram	Mt1	ldbode	bode	
close with Simulink model	Mt1	xclose	linmod	r
closed loop step	LV	RTControlStep	RTctlStep.vi	
closed loop with (-1)	Mt1		cloop	
closed loop with lqr	Mt1	dlqrAndClose	dlqr	
closed loop with lqr	Mt1	lqrAndClose	lqr	
closed loop: obs. + state ctrl.	Mt1	dlqrObsAndClose	dlqr	i, m
closed loop: place obs. + ctrl.	Mt1	dPlaceObsAndClose	place	
compare	LV	SignalGroupCompare	2D signal compare	
control with plant model	Mt1	dlqrObsAndModel	lqr	
controllability form	Mt1		ctrbf	
create signals for identific.	Mt1		makeIOSignal	
extract contin. linear system	Mt1		linmod	
freq.resp. to TF	Mt1	Larxfreqs	arxfreqs	
impulse response	Mt1	Ldimpulse	dimpulse	
impulse response	Mt1	Limpulse	impulse	
impulse response experiment	LV	ImpExperiment	WaveGen	i, r
InOutModel->SS	Mt1		th2SS	
InOutModel->TF	Mt1	Lth2tf	th2tf	
interactive PI-design	LV	PIDesign	PIDesign.vi	
Ld-lg design	LV	LLDesign	LL design/bode.vi	
Ld-lg design	LV	LLDesignBode	LL design/bode.vi	i
lqr designed controller	Mt1	ldlqr	dlqr	
lqr designed controller	Mt1	llqr	lqr	
lqr with observer	Mt1	llqrObs	lqr	

Table 2: Leporello Algorithms

Leporello name	PK ^a	interface called	algorithm file	pr. ^b
lqr with placed observer	Mt1	dlqrObsPlace	lqr, place	
make continuous	Mt1		d2c	
make continuous	Mt1	d2cTF	d2cm	
make discrete	Mt1		c2d	
make discrete	Mt1	c2dTF	c2dm	
make lin/ discrete system	Mt1		dlinmod	
Make new CDC object	L11o		<i>internal</i>	
minimal realization	Mt1		minreal	
model reduction	Mt1	dSSreduct	dbalreal, dmodred	
model reduction	Mt1	SSreduct	balreal, modred	
Modify CDC object	L11o		<i>internal</i>	
nyquist diagram	Mt1		dnyquist	
nyquist diagram	Mt1		nyquist	
observability form	Mt1		obsvf	
P-controlled closed loop	Mt1		PandClose	
P-controller	Mt1		PController	
PI-controller	Mt1		PIController	
place ctrl and observer	Mt1	dPlaceObs	place	
plot	LV	FreqMagPhasPlot	BodePlot	i, m
plot	LV	SignalGroupPlot	2D signal graph	i, m
plot	LV	SignalPlot	2D signal graph	i, m
pole placement closed loop	Mt1	placeAndClose	place	
pole placement controller	Mt1	lplace	place	
poles/zeros plot	Mt1	Lzpplot	zpplot	
pseudoInverse	Mt1		pinv	
random cont. SS	Mt1		rmodel	
random cont. TF	Mt1		rmodel	
random discrete SS	Mt1		drmodel	
random discrete TF	Mt1		drmodel	
real time control	LV	RTControl	RTController.vi	i, r
remove constant trends	Mt1	Ldtrend	dtrend	
remove linear trends	Mt1	LdtrendLin	dtrend	
robust controller	Mt1		robustCtrler	
signal response	LV	FileExperiment	WaveGen	r
simulate Simulink diagram	Mt1		rk45	
SS2TF	Mt1		SS2TF	
SS2ZP	Mt1		SS2ZP	
state space identification	Mt1	SSIdentPEM	pem	
step response	Mt1	Ldstep	dstep	
step response	Mt1	Lstep	step	
step response experiment	LV	StepExperiment	WaveGen	r
swept sine measurement	LV	SweptBode	Swept Sine	i, r
TF2SS	Mt1		TF2SS	
TF2ZP	Mt1		TF2ZP	
y-feedback compensator	Mt1		WaageCompensator	
Z/P plot	LV		ZPPlot	i
ZP2SS	Mt1		ZP2SS	
ZP2TF	Mt1		ZP2TF	

Table 2: Leporello Algorithms

a. Algorithm package:

LV: LabVIEW, Mt1: Matlab

b. Algorithm properties: i: interactive user interface, r: real-time and I/O required, m: multiple results compared locally

Leporello name	Pk	interface called	algorithm file	tp. ^a
# samples	Mt1		sigsize	rt
asympt. stable	Mt1		cZPstable	pr
asympt. stable	Mt1		dZPstable	pr
asympt. stable	Mt1	cSSstable	eig	pr
asympt. stable	Mt1	cTFstable	roots	pr
asympt. stable	Mt1	dSSstable	eig	pr
asympt. stable	Mt1	dTFstable	roots	pr
bode margins	Mt1	bodemargins	margin	rt
controllable	Mt1	controllable	ctrb	pr
eigenvalues	Mt1	eigval	eig	rt
eigenvectors	Mt1	eigvect	eig	rt
min/max y	Mt1		sigminmax	rt
observable	Mt1	observable	obsv	pr
regularMatrix	Mt1		regular	pr
signal length	Mt1		sigLength	rt
static gain	Mt1		dcgain	rt
static gain	Mt1		dcgain	rt
system order	Mt1		orderTF	rt
system order	Mt1		orderZP	rt
system order	Mt1		rank	rt

Table 3: Property and Rating Algorithms

a. Algorithm type:

rt: Rating, pr: Property

Leer - Vide - Empty

Appendix B: Action tree node states and pictures.

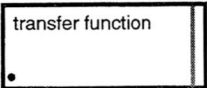
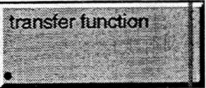
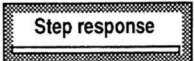
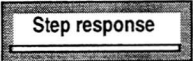
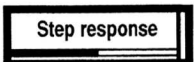
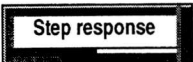
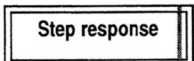
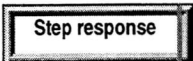

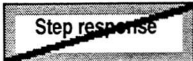
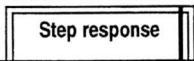
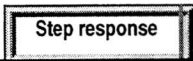
Node State	black and white	color
<i>CDC nodes</i>		
The picture contains a multi-object marker and the target mark. The text line contains the name of the CDC object contained in this node.		
<i>Algorithm nodes</i>		
<i>Allocated node:</i> This node's algorithm has not yet been applied. A visible status bar indicates a parameter source ready to be executed. The text line contains the name of the algorithm assigned to this node		
<i>Executing node:</i> The algorithm assigned to this node is currently being executed. The status bar fills gradually.		
<i>Completed node:</i> This node contains algorithm parameter lists of previous executions. The multi-object marker on the right of the node indicates multiple parameter sets. In case the status bar is shown in this situation, either the parent CDC node or the parameter source contain objects which have not yet been applied (inconsistent node).		
<i>Deleted algorithm node:</i> This algorithm has previously been applied but was deleted later. A reason for deletion can be given or retrieved in the information window of this node.		
<i>Hidden subtree indicator:</i> The wide bar at the bottom of this node indicates a hidden subtree. This mark can also appear on a CDC node.		

Table 4: CDC and algorithm node pictures

Leer - Vide - Empty

Appendix C: Object syntax and naming conventions

In Object Pascal, an object is declared as follows:

```

TYPE
TMyObject = OBJECT(TObject)

    fFirstField: INTEGER;
    fSecondField: INTEGER;

    PROCEDURE TMyObject.Initialize; OVERRIDE;
    { initializes the object }
    PROCEDURE TMyObject.Free; OVERRIDE;
    { frees the object from memory }
    PROCEDURE TMyObject.IMyObject(firstItem, secondItem:
    INTEGER);
    { supplies necessary initial values }

    FUNCTION TMyObject.GetFirstItem: INTEGER;

END;
```

The class defined by this statement is a subclass of `TObject`. In addition to the fields of `TObject` it contains the two integer fields `fFirstField` and `fSecondField`. It inherits all the methods of the `TObject` class and adds the two methods `IMyObject` and `GetFirstItem`. Of the inherited methods, it overrides `Initialize` and `Free`.

A new object of the `TMyObject` class is created in the following short code sample:

```

VAR
    aNewObject: TMyObject;
    value: INTEGER;

BEGIN
    NEW(aNewObject); { allocates a new object and calls
                     its Initialize method }
    aNewObject.IMyObject(3,5); { initializes the fields of
                               the object }
    value := aNewObject.GetFirstItem; { call of an object
                                       method }
    value := aNewObject.fFirstField; { access to an object
                                       field }

END;
```


Naming conventions:

To have the kind of an item reflect its name, we follow the naming conventions proposed by MacApp:

Library units:	<i>ULibraryUnit</i>
Classes:	<i>TClassName</i>
Fields:	<i>fFieldName</i>
Methods:	no convention
initialize method:	<i>IClassName</i>
Constants:	<i>kConstantName</i>
Command numbers:	<i>cCommandNumber</i>
global variables:	<i>gVariableName</i>

Classes presented throughout the thesis are listed in tables of the following form:

Class name

```
Object_name = OBJECT(TSuper_Class)
```

The code excerpt above shows the declaration header of the class. It contains the name of the class used in the code and the name of its superclass.

This paragraph contains a short description of the object functionality.

<i>Fields</i>	
<i>fField1</i>	Short description of the fields
<i>Methods</i>	
<i>Method1</i>	Short description of selected methods

Appendix D: Managing Dynamical Data Structures.

D.1. Variable Size Data Structures

Any scientific software package has to deal with the problem of variable size data structures. In control applications, the main structures are matrices of arbitrary size. The matrices of a state space representation vary with the system order and input and output numbers, measured signals vary in the number of points stored.

Today's programming languages all support n-dimensional arrays, but the array sizes have to be known at compile time. Most languages do support variable size array parameters in procedures, but all of them limit variations to one dimension.

Dynamic memory allocation on the other hand is well supported in the common programming languages. Memory blocks of arbitrary size can be allocated and decollated without any problems. The missing part is only the calculation of the address of a given element from its array indices.

The Macintosh memory management even takes the idea of dynamic memory allocation one step further. To gain increased flexibility in the organization of the available memory, access to dynamic data structures is done through handles instead of pointers (handles are pointers to so called master pointers). This allows a given memory block to be moved. The pointer is adjusted to point to the new location. The handle however still points to the same location since the pointer did not move.

The objects created with Apple Object Pascal make use of this indirectly addressed memory access: all objects are dynamically allocated and referenced to by handles. This is done fully transparent, the programmer never needs to dereference handles in order to access objects or object fields. By hiding this implementation, the object and the handle data structures can differ and therefore hide some data necessary for object management. The first two or four bytes of the dereferenced handle (depending on the compiler switches used) contain an internal object identifier, which can not be

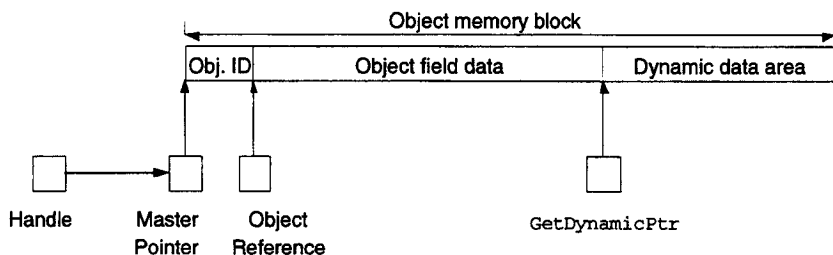


Figure 6.1: Pascal object memory structure

accessed using the object description of the same data structure. This object identifier is used to correctly access the object's methods. Fig. 6.1 shows this structure.

Since the object size does not correspond to the handle size because of the extra word added at the beginning of the object anyway, we can make use of this technique to get additional memory space at the end of the object. By resizing the handle which points to a larger area than the object data fields require, we get additional dynamic memory space where we can store data which vary in size.

The TObject class of MacApp provides the methods to access the necessary structure:

<i>Methods</i>	
GetDynamicPtr	Returns a pointer to the dynamic memory area at the end of the object.
SetDynamicSize	Sets size of dynamic memory area (this sets the additional size, not the size of the handle).

Management of this additional memory area is done in the MacApp TDynamicArray class. It contains methods which provide direct access to data elements located in the dynamical area. Classes which are derived from this class are:

Object list

`TList = OBJECT(TSortedList)`

MacApp class which implements a list of objects. The object handles are stored in sequence. Objects can be inserted or removed from the list.

<i>Methods</i>	
Insert At DeleteAt, etc.	List access methods.

Tree node

`TTreeNode = OBJECT(TLList)`

Leporello class which implements a tree node. In order to save space and inherit list management functions, the tree node does not have a list of children, its children are stored directly in its dynamical memory area. The management of this area is done by the inherited list functions.

Abstract numerical array class

`TVector = OBJECT(TDynamicArray)`

Due to a design limitation, the MacApp dynamical array only allows element sizes which are a power of two. In order to store numerical data (which can be of size 10 or 12 in case of the extended format), the limiting

methods of the dynamic array are overridden to allow other element sizes. This is done by a small speed sacrifice).
 TVector is an abstract class, objects of that class should not be created.

Numerical two-dimensional complex matrix

TMatrix = OBJECT(TVector)

While lists and tree nodes all contain object references in their dynamic memory area, the matrix contains numbers. The TMatrix class allows storage of two-dimensional matrices. Its subclasses are used to store parameters and other Leporello numerical data. It can either contain real or complex elements. If it is complex, the imaginary parts of its contents are located in a consecutive memory block after the real values. Its methods contain element access as well as matrix manipulation methods (sub-matrix extraction, matrix merge, column and row-access).

Methods	
element access methods are provided in subclasses where the element type is fixed.	
InsertRowsBefore	Sub-matrix access
DeleteRows	
InsertColsBefore	
DeleteColumns	
GetSubMatrix	
SetSubMatrix	

Sub-classes of TMatrix are TIntegerMatrix and TDoubleMatrix. Their elements are accessed by the methods GetElement and SetElement.

Note: During the implementation of the Matlab communication and file exchange facilities of Leporello we realized that in the internal data structure of Matlab, matrices are still stored transposed, i.e. in Fortran notation (column-wise). Although the worldwide non-Fortran software community uses row-wise matrix storage format, we decided to adapt the Matlab notation in order to ease communication and file storage.

D.2. Garbage collection

It is generally known, that garbage collection (removal of objects which are not used anymore) is a problem in object oriented environments. Very often an object can not be assigned to a fixed owner. It is therefore not a priori known who is responsible for freeing it from memory. The most crucial situation occurs if the critical object is pointed to from several other objects. They all start using the object, the order in which they stop using it is not defined. If some of the owners wants to free it, the other owners should

be either asked for permission or at least be notified. It is mostly not possible to detect an invalid reference to the freed object.

One solution to that is the implementation of a use-counter. This technique has been used with the lists used in Leporello¹.

We introduced the TLList class, a subclass of MacApp's TList class.

Leporello list class

```
TLList = OBJECT(TList)
```

TLList overrides several of the TList methods to suit our needs (*i.e.* read and write not only move the list's data fields but also all elements in the list).

In addition to that, it includes a garbage collection mechanism.

<i>Fields</i>	
fUsedCount	Number of objects which use this list
<i>Methods</i>	
StartUse StopUse	Functions which update the use count of the object

If the list is to be used by another object, it is not assigned to a variable by using the Pascal ':=' operator, this is done by a call to StartUse:

```
BEGIN
  fStoredList := someList.StartUse;
END;
```

The StartUse method returns SELF, the fUsedCount field is incremented. If the object is no longer needed, the owner calls the StopUse method. fUsedCount is decremented, and if this causes it to reach zero, the list is freed. StopUse returns NIL.

This technique requires all objects accessing a shared list object to use this mechanism. All use-calls must be symmetrical, *i.e.*, each object which called StartUse must call the StopUse method.

1. The best place to implement this functionality would be the TObject class. However, we did not want to change MacApp code. The most general class of all the classes which needed this garbage collection was the list class.

Glossary

Abstract class: (*OOP*)¹ A class which is not used to create object instances, but to provide a set of fields and methods which are overridden in specialized subclasses.

Action tree: (*L*) A tree structure which contains the history of a design process.

Action tree root: (*L*) An algorithm node whose algorithm produces CDC objects (*i.e.* reads them from file, asks them from the user or performs measurements)

Algorithm: (*general*) A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps.

(*Software*) A finite set of well-defined rules which gives a sequence of operations for performing a specific task. [15]

Algorithm node: (*L*) An action tree node which contains all information about the execution of an algorithm: a reference to the algorithm information and the parameter list.

Block diagram: A diagram of a system, a computer, or a device in which the principal parts are represented by suitably annotated geometrical figures to show both the basic functions of the parts and their functional relationships. [15]

CDC object node (CDC node): (*L*) An action tree node which contains the CDC object which resulted from an algorithm execution. A CDC object node is always child to an algorithm node and vice versa.

Consistent algorithm node: (*L*) An algorithm node whose parameter sets have been applied to all the CDC objects contained in its possibly multi-object parent node.

Data flow diagram: A graphic representation of a system, showing data sources, data sinks, storage, and processes performed on data as nodes, and logical flow of data as links between the nodes. [11]

Grandchildren: (*L*) Children of one of this node's children.

- of a CDC node: the results of the algorithms applied to this node
- of an algorithm node: the algorithms applied to its results

Grandparent: (*L*) Parent node of this node's parent.

1. Entries marked (*OOP*) are terms used in object oriented programming. Entries marked with an (*L*) contain terms which were used and defined for the Leporello package.

- of a CDC node: the CDC node its parent algorithm node was applied to
- of an algorithm node: the algorithm node which produced its parent CDC node

Leporello: (*L*) The servant of Don Giovanni in Mozart's opera with the same name [31]. In the first act of the opera, he is to tell all the adventures of Don Giovanni to Donna Elvira. He reads from his 'catalogo', where he is carefully keeping track of all these adventures. He knows their exact numbers ('*Ma in Spagna, son gia mille tre*') and other details. Our CACSD package Leporello is supposed to do exactly that: keep track of what we did during a control design cycle.

Multi-parameter node: (*L*) An algorithm node whose algorithm has been applied repeatedly to the same CDC object. It contains a list of all parameter sets used

Multi-object node: (*L*) A CDC node which contains the results of an algorithm node executed repeatedly, either with several parameter sets on the same CDC object or with one parameter set on another multi-object node. All CDC objects collected in a multi-object node are the results of the same algorithm node and share the same structure.

Multi-object subtree: (*L*) Part of the action tree which was created by applying algorithms to a multi-object node. Each node of the multi-object subtree is a multi-object node.

Override: (*OOP*) A method inherited from a superclass which is changed by a subclass is called overridden.

Parameter: (*mathematical*) A variable that is given a constant value for a specific purpose or process.

(*physical*) One of the constants entering into a functional equation and corresponding to some characteristic property, dimension, or degree of freedom.

(*control systems*) A quantity of property treated as a constant but which may sometimes vary or be adjusted. [15]

Sister Node: (*L*) Tree nodes with the same parent node.

- of an algorithm node: an algorithm applied to the same CDC node
- of a CDC node: a CDC node which contains results of the same algorithm node, but which have a different structure. -> twin node

Subclass: (*OOP*) A class which inherits all the fields and methods of its superclass. Additional fields and methods can be added, or inherited methods can be overridden

System: (*automatic control*) A collection of interconnected physical units or mathematical equations or operations. [15]

Twin nodes: (*L*) Multi-object node which was split up into several nodes. All twin nodes therefore contain results of the same algorithm node and share the same structure. They may have different children.

Target CDC node: (*L*) The twin node to receive the results of the next execution of its parent algorithm. If this result does not match the structure of CDC objects contained in the twin, a new CDC node is created.

Leer - Vide - Empty

Bibliography

- [1] A.L. Ambler and M.M. Burnett. *Influence of Visual Technology on the Evolution of Language Environments*, IEEE Computer, pp 9-22, October 1989
- [2] Apple Computer. *Programmer's Guide to MacApp*, Developer Technical Publications, 1992
- [3] H.A. Barker. *Open Environments and Object-Oriented Methods: The Way Forward in Computer-Aided Control System Design*, in Proc. IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, Tucson, 1994
- [4] K. Bastiaens and J.M. Van Campenhout. *A Visual Real-Time Programming Language*, Control Eng. Practice, Vol. 1, No. 1, pp. 59-63, 1993
- [5] L. Biétry and M. Kochsiek. *Mettler Wägelexikon* (Mettler weighting dictionary), Mettler Greifensee, 1982
- [6] M.H. Brown and R. Sedgewick. *A System for Algorithm Animation*, ACM Computer Graphics, Vol. 18, No. 3, pp. 177-186, July 1984
- [7] S.-K.Chang. *Visual Languages: A Tutorial and Survey*, IEEE Software, pp. 29-39, January 1987
- [8] M. Cianchi. *Die Maschinen Leonardo da Vincis* (Leonardo da Vincis machines), Florence 1984
- [9] J. Daehler. *Ein Werkzeug für den Entwurf verteilter Systeme auf der Basis erweiterter Petri-Netze*, Diss. ETH No. 8770, 1989
- [10] C. Ganz, P. Kolb and M. Rickli. *A Graphical Environment for Systems Engineering*, In Proc. Singapore International Conference on Intelligent Control and Instrumentation: SICICI'92, IEEE, pp. 1146-1151, Singapore 1992.
- [11] C. Ganz, P. Kolb and M. Rickli. *A Data Management Tool for Computer Aided Control Engineering*, in Proc. American Control Conference, pp. 3076-3080, San Francisco, 1993
- [12] E.P. Glinert and S.L. Tanimoto. *Pict: An Interactive Graphical Programming Environment*, IEEE Computer, pp. 7-25, November 1984
- [13] G. Grübel and H.-D. Joos. *The Control Systems Engineering Numerical Subroutine Library RASP*, TR R14-90, DLR Institut für Dynamik der Flugsysteme, 1990.
- [14] G. Grübel, H.-D. Joos, M. Otter and R. Finsterwalder. *The ANDECS design environment for control engineering*. In Proc. of the 12th IFAC World Congress, pp. 6-447 - 6-454, Sydney 1993
- [15] F. Jay (ed.). *IEEE Standard Dictionary of Electrical and Electronics Terms*, The Institute of Electrical and Electronics Engineers, Inc. New York, 1984
- [16] R.J.K Jacob. *A State Transition Diagram Language for Visual Programming*, Computer, pp. 51-59, August 1985
- [17] H.-D. Joos. *Informationstechnische Behandlung des mehrzieligen regelungstechnischen Entwurfs*, VDI Forschungsberichte, Reihe 20, Nr. 90, 1993

- [18] A. Kierulf. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies*. Diss. ETH No. 9135, 1990
- [19] J. Kodosky, J. MacCracken and G. Ryman. *Visual Programming Using Structured Data Flow*, in Proc. 1991 IEEE Workshop on Visual Languages, Kobe 1991
- [20] P. Kolb, M. Rickli and W. Schaufelberger. *Discrete Simulation and Experiments with FPU and BlockSim on IBM PC's*, In Proc. Conference on Advances in Control Education, pp 68-73, Boston 1991.
- [21] P. Kolb, C. Ganz and M. Rickli. *An Object Oriented Kernel for Control Systems Engineering*. In Proc. of the 12th IFAC World Congress, Vol. 6, pp 443-446, Sydney 1993
- [22] P. Kolb. *Nutzung der objektorientierten Methodologie für den computerunterstützten Entwurf von Regelsystemen (The Use of the Object-oriented Methodology for CACSD)*, Diss. ETH No. 10962, 1994
- [23] L. Ljung. *System Identification: Theory for the User*, Prentice Hall 1989
- [24] J. M. Maciejowski. *A Core Data Model for Computer-Aided Control Engineering*. Cambridge University Engineering Department Technical Report, CUED/F-CAMS/TR257. 1985
- [25] J. M. Maciejowski. *Data Structures and Software Tools for the Computer Aided Design of Control Systems: a Survey*. Proc. of the Congress Computer Aided Design in Control Systems, pp 27-38, Beijing 1988.
- [26] A. Marttinen and T. Telkkä. *A Hierarchical Process Modelling Environment*. Proc. of the 11th IFAC World Congress, pp 73-78, Tallinn, 1990.
- [27] S. E. Mattsson and M. Andersson. *A Kernel for System Representation*. Proc. of the 11th IFAC World Congress, pp 91-96, Tallinn, 1990.
- [28] J. Milek. *Stabilized Adaptive Forgetting in the Recursive Parameter Estimation*, Diss. ETH No. 10893, 1994
- [29] G.A. Miller. *The magical number seven, plus or minus two: Some limits on our capacity for processing information*, Psychological Review, Vol. 63, Nr. 2, pp. 81-96, 1956
- [30] F.S. Montalvo. *Diagram Understanding, the Symbolic Descriptions Behind the Scenes*, in Visual Languages and Applications, T. Ichikawa, E. Jungkert and R.R. Korfhage (Eds.), Plenum Press, New York 1990
- [31] W.A. Mozart and L. da Ponte. *Il Dissoluto punito ossia il Don Giovanni*, Leipzig 1801
- [32] I. Nassi and B. Shneiderman. *Flowchart Techniques for Structured Programming*, SIGPLAN Notices of the ACM, Vol. 8, No. 8, Aug. 1973
- [33] D.C. Smith. *Principles of Iconic Programming*, in E.P. Glinert (ed.) "Visual Programming Environments: Paradigms and Systems", pp. 216-251, IEEE Computer Society Press, Los Alamitos, 1990
- [34] G. Peretti, J. Milek and L. Guzzella. *CTRL-Lab - a Tool for DSP-based Implementation and Testing of Control Algorithms*. in Proceedings of SICICI'92, pp. 1130-1135, Singapore 1992
- [35] G. Raeder. *A Survey of Current Graphical Programming Techniques*, IEEE Computer, pp. 11-25, August 1985

- [36] O. Ravn and M. Szymkat. *The Evolution of CACSD Tools - A Software Engineering Perspective*, in Proc. of the 1992 IEEE Symposium on Computer Aided Control System Design, pp. 225-231, Napa, CA 1992
- [37] C.M. Rinvall. *Man-Machine Interfaces and Implementational Issues in Computer-Aided Control System Design*, Diss. ETH No. 8200, 1986
- [38] R. Samer and A. Spreiter. *Graphische Programmierung Adaptiver Regler*, Semester project IfA 8714, 1992
- [39] C. Schmid. *Techniques and Tools of CADCS*. Proceedings of the Conference Computer Aided Design in Control Systems, pp 91-99, Beijing 1988
- [40] U. Schult, *A Graphics-Only Programming Tool*, in Proc. of the EUROMICRO 89 Conference, Cologne, 1989
- [41] W. Siegl et. al. *Anleitungen zum Fachpraktikum in Automatik* (Instructions for the Automatic Control Laboratory Experiments), Institut für Automatik, 1994
- [42] M.N.S. Swamy, K. Thulasiraman. *Graphs, Networks and Algorithms*, John Wiley & Sons Inc, 1981
- [43] M. Szymkat. *CACSD Tools- Implementational Aspects*. Computer Aided Control System Design - Proceedings of the TEMPUS Summer school '91, Warsaw. 1991
- [44] S.L. Tanimoto and E.P. Glinert. *Designing Iconic Programming Systems: Representation and Learnability*, IEEE Proc. Workshop on Visual Languages, pp. 54-60, 1986
- [45] J. H. Taylor, D. K. Frederick and M. Rinvall. *Computer Aided Control Engineering Environments: Architecture, User Interface, Data-Base Management, and Expert Aiding*. Proceedings of the 11th IFAC World Congress, Tallinn, pp 55-66. 1990
- [46] L. L. Tripp. *A survey of Graphical Notations for Program Design - An Update*, ACM SIGSOFT Software Engineering, Vol. 13, No. 4, pp. 39-44, 1988

Software references

- [47] ACSL/Graphic modeller, Mitchell and Gauthier Associates Inc. 200 Baker Avenue, Concord, MA 01742-2100
- [48] BlockSim, Institut für Automatik, ETH-Zentrum, CH-8092 Zürich, no longer supported
- [49] GenT, ABB corporate research, not publicly available
- [50] LabVIEW, National Instruments, 6504 Bridge Point Parkway, Austin, TX 78730-5039
- [51] Matlab, The Mathworks, Inc, 24 Prime Park Way, Natick, MA 01760
- [52] Macintosh Programmer's Workshop, Apple Computer Inc, 20525 Mariani Avenue, Cupertino, California 95014-6299
- [53] Simulink, The Mathworks, Inc, 24 Prime Park Way, Natick, MA 01760
- [54] SystemSpecs, The IvyTeam, Alpenstr. 9, CH-6300 Zug
- [55] WorkBench, Strawberry Tree Inc., 160 South Wolfe Road, Sunnyvale, CA 94086

Trademarks:

Matlab and Simulink are registered trademarks of The MathWorks Inc.

Macintosh is a trademark of Apple Computer Inc.

Microsoft is a registered trademark of Microsoft Corporation

UNIX is a trademark of UNIX Systems Laboratories

LabVIEW® is a trademark of National Instruments Corporation

SystemSpecs is a trademark of IvyTeam

Product and company names listed are trademarks or trade names of their respective companies.

Index

A

ACSL 32, 34, 40
 action tree 59
 node information 115
 root 76
 adaptive controller 49–50
 algorithm
 CDC algorithm 67
 execution 69, 85
 external access 70
 parameter variations 81
 rating algorithm 67, 74
 selection window 77
 structures 67
 algorithm execution object 68
 hierarchical algorithm 101
 algorithm head 68
 algorithm information object 68
 hierarchical algorithm 100
 algorithm node 59, 74
 deleted node 95
 execution 72, 76, 85
 execution scheduler 97
 multi-parameter node 89
 re-execution 78
 algorithm node shape
 allocated empty node 78, 139
 completed 78, 139
 deleted node marker 96, 139
 executing 78, 139
 hidden subtree marker 139
 hierarchical algorithm 104
 marked node 80
 multi-object marker 82, 139
 status bar 78, 80, 89, 139
 algorithm scheduler 97
 automatic parameter optimizer 106
 consistency scheduler 98
 hierarchical algorithm 101
 animated pictures 21
 arborescence 61
 ARX-model identification 125, 127

B

block diagram 32
 action tree 107
 continuous 33
 control flow 34
 discrete 33

 simulation 34, 38, 40
 block diagram tree 107
 numerical algorithms 108
 sub-system exchange 110
 sub-system variations 109
 topology changes 108
 BlockSim 40–42

C

CDC 65
 CDC algorithm 67
 CDC node 72, 75, 87, 91
 creation 72
 history reference 85
 multi-object node 81, 89
 subset node extraction 90
 subset node merge 92
 target node 84
 target selection 90
 twin node 84, 87
 CDC node shape 139
 hidden subtree marker 139
 multi object marker 82, 139
 target marker 89, 91, 139
 twin branch 89
 CDC object source 82
 comic strip 5
 control data class 65
 control flow descriptions 26–30
 control systems design cycle 58
 copy and paste operations 93
 copy algorithm node 93
 copy CDC node 93
 copy pictures 93
 paste CDC node algorithm 94

D

data flow descriptions 30–34
 data nodes 58
 data visualization 36, 38, 46
 deadbeat controller 123
 delete node 95
 design cycle 58
 discrete event dynamic systems 28
 display dimensions
 one 17
 two 17
 three 19
 four 20
 dynamical pictures 21

E

Easy5 40
extended Petri net 31

F

flowchart 27
four dimensional displays 20
front panel 45, 46

G

G visual language 45
garage door example 28, 29, 30, 31, 37
garbage collection 145
generator controller example 33, 40
GenT 28, 42–45
graphical programming 23

H

hierarchical algorithm 99–103
 algorithm executor 101
 algorithm information 100
 algorithm scheduler 101
 node shape 104
 parameter extracting source 102
 structure 103
hierarchical systems 107
high level Petri net 31
human senses
 channel capacity 17
 information theory 17

I

IMPACT 63
indirectly referenced list 87
interactive algorithms 21, 22, 118

L

laboratory balance example 111–130
LabVIEW 17, 19, 22, 45–48, 114, 116, 118
LabVIEW algorithms 135
lead-lag controller design 22, 117
least squares transfer function estimation 121
Leporello
 available algorithms 135
 block diagram editor 107
 communication 25, 114
 document window 115
 interactive algorithm 22
 main features 56
 plot tool 19
 prototype implementation 113
 user interface 114
LQR controller design 124

M

map 6, 21
Maple 20, 51
Mathematica 20, 51
Matlab 18, 20, 38, 51, 56, 114
 scripting 18

Matlab algorithms 135
matrix data structure 145
MatrixX 40
mesh plot 20
movie 5, 21
multi-object subtrees 83

N

Nassi-Shneiderman diagrams 26

O

object oriented programming 25, 63
 fields 63
 inheritance 63
 methods 63
Object Pascal 141
 object declaration 141
 object instance 141
 object memory structure 143
object syntax 142
one-dimensional displays 17

P

parallel processes 25
parameter class 66
parameter lists 66
parameter optimization 105–106
parameter source 75, 79
 parameter extracting source 102
 parameter input 76
 parameter sweep 105
parameter sweep 105
paste algorithm 94
paste CDC node 94
Petri net 29
 high level Petri net 31
 marking 29
 places 29
 simulation 35
 tokens 29
 transitions 29
plot manipulations
 axis scaling 18
 hide curves 18
 interactive 19
 line styles 18
 read value 18
 show grid 18
 zoom 18
pretty-printer 24
program animation 36, 47

R

random binary signals 125
rating algorithm 67, 74
real-time programming 25, 39, 44
 exception handling 34

S

scientific visualization 15, 16–23

- sequential program 25
- Simulink 38–40
- software aspects
 - control flow 26
 - data flow 26, 30–??
 - data structure 26
 - topology 26
- software visualization 15
- state diagram 28
- state feedback controller 123
- structured program layout 24
- syntactical formatting 24
- system class 66
- SystemSpecs 31, 35–38

T

- TAlg 69
- TAlgATNode 74
- TAlgHead 68
- TAlgScheduler 97
- TCDC 65
- TCDCATNode 75
- TCDCSource 82
- TConsistentScheduler 98
- TDbIndexParamSource 87
- TDeletedAlg 96
- TExtAlgHead 69
- THierarchAlgHead 100, 101
- THierarchAlgScheduler 101
- three dimensional diagrams 19
- TLList 73, 146
- TLListSource 75
- TMatrix 145
- TNoCDC 76
- TParameterInput 76
- TParameterOptimizer 106
- TParameterSource 75
- TParameterSweep 105
- TPasteNode 94
- traffic signs 7
- tree node 73
- TSys 66
- TTreeNode 73, 144
- two dimensional displays 17

V

- virtual instrument 45
- virtual reality 17
- visual languages 23, 35–48
- visual programming 15

W

- Workbench 47



Richard Stra

Curriculum Vitae

I was born on April 25th 1963 in Jersey City, NJ. I spent my primary school years in Meggen (Switzerland) and attended the Gymnasium of Mathematics and Natural Sciences in Lucerne afterwards. From there I obtained the Matura Certificate in 1982. The same year I started to study Electrical Engineering at the Swiss Federal Institute of Technology in Zurich, where I graduated 1987.

Since then I have been working as a teaching and research assistant at the Automatic Control Lab of the ETH. During this time I attended postgraduate courses in automatic control. I then participated in a research project which concentrated on the use of visual languages in automatic control. The results of this project led to the work presented in this thesis.