INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600

Application of Technology Insertion to Particle Accelerator Modernization and Operations Support

by

PETER CHRISTIAN LIND

B.Sc. (Hon.), M.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

McMaster University

PARTICLE ACCELERATOR MODERNIZATION AND OPERATIONS SUPPORT

DOCTOR OF PHILOSOPHY (1997)
(Electrical Engineering)

MCMASTER UNIVERSITY
Hamilton, Ontario

TITLE: Application of Technology Insertion to Particle Accelerator Modernization and Operations Support

AUTHOR: Peter Christian Lind, B.Sc. (Hon.) (Queen's University), M.Sc. (McMaster University)

SUPERVISOR: Dr. W.F.S. Poehlman, B.S., B.Sc., M.Sc., Ph.D., P.Eng.

NUMBER OF PAGES: xiv, 220

Thesis Abstract

This thesis discusses a design methodology that can be employed as a framework for the design of computer-centered performance enhancement systems for technology insertion into non-computerized human-machine systems. This methodology seeks to hybridize and exploit the benefits of several diverse areas of electrical and computer engineering, specifically knowledge-based reasoning, fuzzy logic-based control, real-time system design, human-machine interaction, computer-human interface, and software engineering.

The methodology for combining these disciplines is used in the realization of a specific application: The Particle Accelerator Control Expert System (PACES), an artificial intelligence-based performance enhancement and control system for a KN-3000 Van de Graaff particle accelerator. PACES combines knowledge-based reasoning and human factors concepts to furnish accelerator operators with a computer-centered operations support facility. Knowledge-based reasoning is coupled with real-time systems concepts to provide the capability for autonomous accelerator control. Technology insertion, human-machine interaction and human-computer interface concepts are used to design a simple, safe, flexible and low-cost operations support system that is accepted by both veteran and novice operators alike.

This thesis is dedicated

to my parents Veronica and Niels,
together in my heart,

to my wife, Laurel,
always by my side and in my thoughts,

to my daughter, Emily,
the joy of my life and my inspiration.

Acknowledgements

A great many people deserve credit for assisting me during the years when I worked on this thesis. They are listed here in no particular order:

Dr. Poehlman, for his long-lasting guidance, enthusiasm and patience.

Dr. McCrackin and Dr. Qiao, members of my thesis committee, for their helpful participation and suggestions over the years.

Dr. Wilkinson and Dr. Wesolowsky, members of my examination committee, for not being too hard on me during my oral defense.

My parents, my brothers and my sister, for their love and encouragement.

My wife and her parents, for their long-time support and generosity.

Mr. Bob McNaught, McMaster Accelerator Lab (retired), for his kind tutelage and assistance during development of the circuitry, and for his lasting friendship.

Mr. Jim Stark, for his enthusiasm in the PACES project, credit for the initial concept of PACES, and sagely advice in the black art of accelerator operations.

Mrs. Marg Belec and Mrs. Sandy Trottier, Dept. of Computer Science and Systems, for their years of courtesy and administrative assistance.

Ms. Cheryl Gies, Dept. of Electrical and Computer Engineering, for her administrative finesse and cheerfulness.

Mr. Frank Strain, M. Jean-Roch Brisson, Mr. Bernie Hoffarth and Mr. Trevor Jones, of DREO, for their aid in development and testing of the PACES prototype.

Mr. Dan Trottier and Mr. Chris Bryce, Dept. of Computer Science and Systems, for all the times they provided help and advice with computer-related problems.

Mr. Gary Mulligan, formerly of the McMaster Accelerator Lab, for the late Friday nights he spent with me trying to get the accelerator to work properly.

Dr. Brad Rodriguez, my cellmate released early on good behaviour, for his helpful suggestions and camaraderie in the trenches of accelerator automation.

Mr. Alan McIlwain, formerly of AECL Whiteshell Labs, for his efforts to help develop the field version of PACES.

Mr. Sam Hosein, Mr. Terry Reimer, and Mr. Ray Warenko, of AECL Whiteshell Labs, for their aid, knowledge and expertise as accelerator operators.

My many friends in Pinawa, Manitoba, for their hospitality and companionship during my field visits to AECL Whiteshell Labs.

Mr. Winston Williams and Mr. John Cave, for their help with the nuts and bolts.

Dr. Tom Cousins, for his role as the PACES project's scientific authority at DREO.

Dr. Harold Haugen, for his support during his term as director of the McMaster Accelerator Lab.

Dr. Rick McCauley-Newcombe for initial training as an accelerator operator, and help during knowledge engineering.

Mr. Sam DeMooy for his pioneering work in accelerator simulation software and early work in knowledge engineering for accelerator control.

Mr. Ernest Siddall for his insight and advice on systems reliability.

All of my teachers, past, present and future, for bestowing the great gift of knowledge.

It's a 'book' I've been working on

for the past four years,

entitled "Impossible Vacation"...

due to be published ... two years ago.

It's 1900 pages long,

and I think it's just about finished...

— Spalding Gray, "Monster in a Box", *1992*

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

List of Figures

List of Figures (continued)

List of Figures (continued)

List of Figures (continued)

List of Tables

Chapter 1

Introduction

"The modern computer is truly an extension of the human mind.
It has no intellect of its own but it is a tremendously energetic clerical slave
that works tirelessly and uncomplainingly for very low wages.
As a result, it is potentially useful in almost the whole gamut of human activities."
— Ernest Siddall, ([Sid94], § 2, p. 5)

Today, in this modern era of computerization, there remains an immense establishment of mature systems[1] and complex processes[2] that are not computerized to any appreciable degree. A serious quandary facing engineers is the problem of reconciling the wide-ranging power of modern computers with such systems and processes. This thesis presents a methodology for computer-based modernization of small-scale, human-tended complex processes.

Computer-centered modernization has great potential benefits, but it must be performed sufficiently well that the resulting system functions *properly* and *reliably*, and is *accepted* by the users. The thesis identifies and explores some important concerns and details of technology insertion. It suggests a hybridized methodology for computerization to accomplish performance enhancement and operations support. The methodology serves to produce reliable and user-friendly systems.

One novel aspect of this research is the systematic hybridization approach that has been developed and has been demonstrated to work in practice. The result is a framework that can be applied to design and implement computer-based systems to provide both autonomous capability and operator performance enhancement for human-tended complex

[1] The term *mature* is intended to imply a complex process or system which has been in operation for a long period of time, and has therefore reached a state of maturity and has been well 'debugged'.
[2] In the sequel, the terms *system*, *process*, *complex process* and *machine* will be used interchangeably.

process modernization. The autonomous capability is made possible through the incorporation of artificial intelligence (AI) techniques for real-time control.

The hybridization methodology introduced in this thesis is believed to be novel. It provides a balance between human operator and complex machine in an effort to ensure that the operator enjoys increased ability to operate the machine. At the same time the machine exhibits expanded capability to perform its job. It is the contention of this thesis that such modernization is best accomplished by integrating several relevant but distinct disciplines of computer engineering to yield a hybridized system that bridges the gap between a mature complex process and its human operator.

The thesis concludes that it is not only possible but readily accomplishable, via computer-centered technology insertion, to modernize and upgrade mature, human-tended machines even if they were never intended to be modernized in such a way.

The potential benefits of computerization are many-fold, including such things as improved process efficiency, operator performance enhancement, operations support, and reliable automation. Consequently, there is significant impetus for engineers to take advantage of computerization whenever possible, be it as an integral feature of the original design of a *new* system or as a new component *retrofitted* to an *existing system.*

In the world at large there is a panoply of mature, complex machines (such as particle accelerators, nuclear reactors, factory assembly lines, water treatment plants and power plants) which rely heavily (or entirely) on human control and supervision.

The operation of such a mature 'human-tended' machine can be enhanced through modernization and computerization, extending the machine's useful lifespan. In an era of budget-cutting, workforce reduction and limited funds for commissioning of new machines, any such modernization must be of low cost and high extensibility. Essential to such technology upgrading is that the modifications must be safe, reliable, flexible, low-cost and operator-friendly.

Whereas the design and development of new complex systems can (and usually does) incorporate computerization from the outset, there is difficulty in adding such

computerization *a posteriori* to existing, non-computerized systems. The latter is a form of *technology insertion*, in that it involves the insertion of modern technology into an existing, mature system which quite likely was never intended or designed to make use of computerization. Technology insertion, in this context, is a process of *modernization*, a challenging undertaking rich in complications and problems, and often multi-disciplinary in nature. Many of these problems are concerned with 'interfacing' the computer system with both the machine and the humans who operate it. Interfacing with the machine requires real-time systems and control systems concepts, while interfacing with the human operators[3] involves human factors concepts, primarily in the sub-domains of human-computer interaction (HCI) and the computer-human interface (CHI).

In the case of machine interfacing, the computer system must minimally control the machine *properly* and *reliably* (that is, respond with timeliness and without error). This implies that the machine interface (MI) relies on electrical and computer engineering principles such as control systems concepts, real-time systems concepts and instrumentation circuitry design. In the case of human interfacing, it is crucial that the computer system is both accepted and used by the human operators. *If either of these interface requirements fails, the technology insertion exercise, as a whole, fails.* Therefore, both of these interface constraints must be properly addressed for the technology insertion operation to succeed.

As illustrated in Figure 1-1, a mature human-machine interaction is both highly complex and highly effective due to the many work-hours that human operators have interacted with the machine; the human-machine system can be considered 'well debugged' in that the system possesses a high degree of evolution with regard to proper operation, robustness and reliability. Consequently, any attempt to separate the human-machine relationship via insertion of a *computer-based intermediary* would require great care to prevent loss of overall system effectiveness (Figure 1-2).

---

[3] In the sequel, the terms *human*, *user* and *operator* will be used interchangeably.

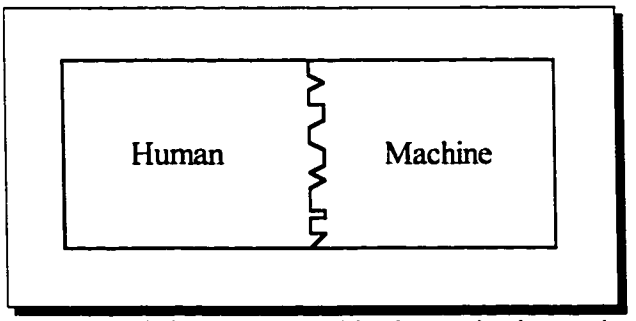


Figure 1-1. Existing human-machine interaction is complex.

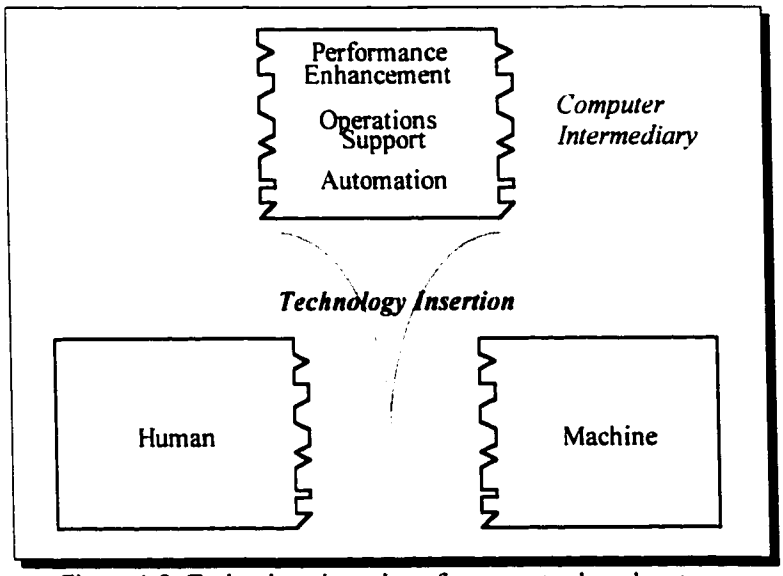


Figure 1-2. Technology insertion of a computer-based system.

The technology insertion disrupts the existing system by creating a *technology insertion gap* between the user and the machine; the intermediary *must* bridge this gap to such a degree that neither user nor machine suffers from the insertion. In practice, such ideal technology insertion is impossible, but it serves as an asymptote to which any technology insertion operation should approach within the confines of practical limits, such as time-frame and budget.

Once the computer intermediary is in place, the new composite system enters a state of *post-insertion development*, and begins operation as a 'disparate relation' between human, computer system and machine (Figure 1-3a). This stage can be described as a 'state of shock' in which all three components (but primarily the human) experience disarray due to the disruption of the original human-machine relation. A crucial first step occurs when (if) the new computer system is *accepted* by the human (Figure 1-3b). This occurs when the

human has sufficiently adapted to the new computer system that machine operation can resume. Hollnagel ([Hol91]) calls this state the 'embodiment (amplificatory) relation', describing it this way:

> "[The computer intermediary] transforms the experience [of the machine] and mediates it to the user. More than that, it also amplifies the experience, e.g., by highlighting those aspects of it that are germane to the task while simultaneously reducing or excluding others."

That is, the user perceives the computer as a *tool* for using (accessing, operating) the machine; the computer is perceived as an entity *separate from* the machine through which the user and machine interact.

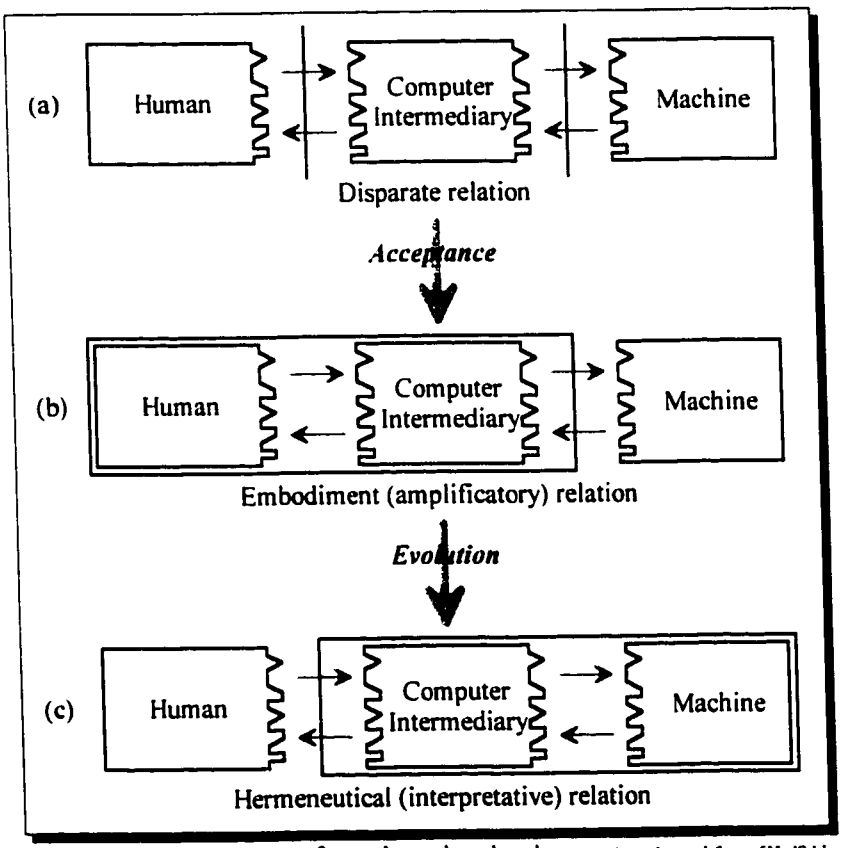


Figure 1-3. Three stages of post-insertion development. *Adapted from [Hol91]*

Gradually, as the user becomes better attuned to the computer intermediary, there is an evolutionary shift towards a different relation (Figure 1-3c), the hermeneutical[4] (interpretative) relation. Ihde, cited by Hollnagel (*Ibid*), describes this relation as one in

[4] *Hermeneutic* is an obscure word meaning *interpretive*.

which the user's "experiential terminus is with the [computer]". Hollnagel (*Ibid*) writes further:

> "The user has moved from an experience *through* the [computer] to an experience *of* the [computer]. It is thus the state of the [underlying machine] as represented by the [computer] which in itself becomes important. In the extreme case there actually is no experience of the [underlying machine] except that provided by the [computer]."

In this case, the user begins to perceive the computer as *part of* the machine, the frontispiece through which the user interacts with the machine. During this evolutionary phase, users begin to realize the potential of the computer intermediary, and often offer suggestions as to how the computer system can be expanded or extended in its role. This state can be viewed as the final (and most valuable) stage of 'acceptance' because the users are sufficiently comfortable with the computer system that they are suggesting how it can be improved to suit their perceived needs.

In summary, technology insertion as a means of modernizing and computerizing an existing human-machine system must embrace two forms of interfacing: both computer-machine and computer-human interfacing must be addressed. There are, in a sense, two 'faces' to the interface problem, and the resulting system will only succeed if both types of interfacing are accomplished successfully.

1.1 Thesis Outline

It is clear that upgrading a non-computerized, human-tended system cannot be solved simply by installing a computerized controller. In addition, it is necessary to 'upgrade' the human operators as well, by taking care to develop a human-computer interface that is both accepted and usable by the operators. Consequently, a hydridized approach to complex machine modernization is required. It is essential that researchers and engineers seeking to perform such modernization are able to draw upon several pertinent fields, including real-time control systems, artificial intelligence, human-machine interaction and computer-human interface.

Since such hybridization introduces a high level of complexity and complication, it is imperative that system designers are aware of potential conflicts between these fields, such

as, for example, the problem of reconciling the slowness of artificial intelligence reasoning with the necessity for real-time control responses, or the problem of striking a balance between ease of learning (for novice users) and ease of use (for experienced users).

The technology insertion problem can be approached by using concepts taken from the areas of real-time systems, artificial intelligence, and human factors, as shown in Figure 1-4. These three fields are united within the technology insertion exercise through judicious use of software engineering principles and techniques.

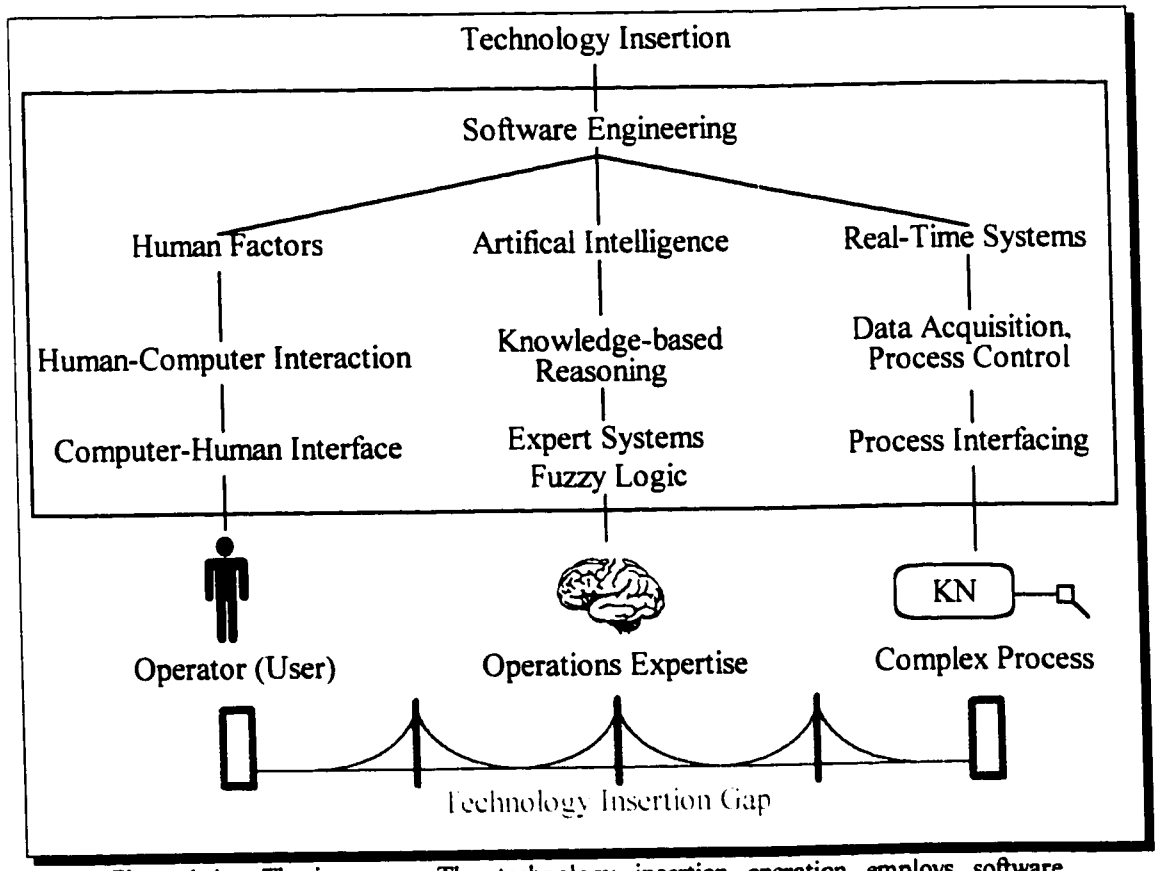


Figure 1-4. Thesis scope: The technology insertion operation employs software engineering principles to combine elements of real-time systems, artificial intelligence and human factors to form a multi-disciplinary, hybridized computer system for modernization of complex process operation and automation.

This thesis presents a framework for a general methodology underlying the design of computer-based performance enhancement and operations support systems for modernization of small-scale, human-tended complex processes. In this context, the term *small-scale* is used to describe systems that are small in terms of complexity (compared to, for example, a nuclear reactor), small in terms of number of control variables, and

small in size of operator pool. Technology insertion for such a small-scale system need only involve a few workers (e.g. control engineer, electrical engineer, process engineer, software engineer, human factors engineer, computer scientist, domain experts, and machine operators), and can be accomplished under a relatively small budget and in a relatively short timeframe. In contrast, technology insertion for larger-scale systems would be expected to require a considerably greater workforce, larger monetary expenditure and longer development time.

The methodology for small-scale technology insertion is illustrated through a case-study application concerning the design and development of the Particle Accelerator Control Expert System (PACES). This research took part under the auspices of the Department of National Defence and the Defence Research Establishment Ottawa (DREO). PACES was consigned to provide operators with a computer-centered means of modernizing a particle accelerator site to improve operational expertise and performance. Three versions of PACES were developed during the project: First, a proof-of-concept prototype was implemented for the KN-3000 at the McMaster Accelerator Lab, McMaster University. Next, a field version was customized for the KN-3000 at DREO. Finally, PACES was upgraded and customized as a second field version[5] for the KN-4000 at the Whiteshell Labs of Atomic Energy of Canada Ltd (AECL). Consequently, a large portion of PACES development involved implementing a generic control system which was tailorable to individual accelerator sites and specific operating regimens.

One novel aspect of this research is the hybridization approach that has been followed, and the net result is a framework methodology that can be applied to the design and implementation of computer-based systems to provide both autonomous capability and operator performance enhancement for human-tended complex process modernization. The autonomous capability is made possible through the incorporation of artificial intelligence (AI) techniques for real-time control, specifically knowledge-based systems and fuzzy logic concepts. The operator performance enhancement stems from the utilization of human-computer interaction and computer-human interface concepts in system design. These diverse areas are united into a cohesive system using principles of

[5] This field version was named the "Particle Accelerator Control and Operations Support System" (PACOSS). For clarity, the acronym PACES will be used throughout this thesis to represent both the prototype and field versions.

software engineering, such as modularity, information hiding and the software development life-cycle.

The hybridization approach explored in this thesis seeks to strike a balance between human operator and complex machine, ensuring that: ① The operator enjoys increased ability to operate the machine (e.g. the computer system helps streamline the operator's job by providing automated decision support and control assistance); and, ② The machine exhibits expanded capability to perform its job (e.g. the computer system performs automatic optimization, fault detection, diagnosis and recovery). It is the contention of this thesis that such modernization is best accomplished through the marrying of several different but relevant facets of computer engineering to yield a hybridized system that bridges the gap between human operators and mature complex process.

* * *

This chapter has introduced the notion of employing computer-centered technology insertion as a means of upgrading and modernizing non-computerized complex human-machine systems in order to prolong their useful lifespan and to enhance and extend their utility in ways not previously possible.

Chapter 2 introduces the PACES case study by describing the three specific particle accelerator research facilities and how the accelerators are operated. This background information serves to set the stage for the following chapters that describe the design, development and performance of the control system.

Chapter 3 presents pertinent background information on the four fields that are applied to the technology insertion problem: real-time systems, artificial intelligence, human factors and software engineering.

Chapter 4 elaborates on the design and development of PACES, illustrating how this specific technology insertion exercise uses a hybridization of concepts and techniques borrowed from artificial intelligence, human-computer interaction, computer-human interface, real-time systems and software engineering. This hybrized approach is used to facilitate the dual task of performance enhancement and operations support, including:

① Employment of real-time systems and process interface concepts for connecting PACES with the particle accelerator for data acquisition and control; ② Use of artificial intelligence techniques, such as knowledge-based inferencing and fuzzy logic-based reasoning, for autonomous accelerator operation; and, ③ Application of human-computer interaction and computer-human interface principles to the design, implementation and evolution of the PACES graphical user interface.

Chapter 5 extends the information presented in the preceding chapter by analyzing the performance of the system's artificial intelligence-based controllers, illustrating how PACES is able to start up the accelerator and maintain particle beam stability.

Chapter 6 presents a general discussion, suggesting how the approach followed during PACES design and development can be extrapolated to technology insertion operations in general for the modernization of small-scale non-computerized human-machine systems.

Finally, Chapter 7 summarizes the main tenets of this thesis and proposes some avenues of study for future work.

Chapter 2

The KN-3000 Particle Accelerator

The KN-3000 particle accelerator of the Nuclear Effects Division, Defence Research Establishment Ottawa (DREO), is an elderly workhorse in the field of particle physics. In active service since the 1950s, this machine has provided many years of good research, and promises to continue to do so for several more years to come. Two key problems have arisen in recent years, relating to availability of operator expertise and the gradual ageing of the accelerator. First, the long-time operators have reached mandatory retirement age, resulting in an acute decrease in on-site operator expertise; simultaneously, due to budgetary constraints, there is little prospect of hiring new operators with sufficient expertise to replace that lost due to retirement. Second, as the accelerator ages, it suffers increasingly from failure due to wear and tear of its components; simultaneously, the same budgetary issues limit the amount of refurbishment or upgrading that can be performed, and in most cases the machine can only be restored to operation, without significant financing for modernization or improvement of components. In essence, these two problems mean that the accelerator facility is operating in a 'preservative mode' or 'holding pattern': there is only sufficient funding to ensure the facility continues to operate at its present level of machine utilization and operations expertise. Yet, there is a desire to extend the usability and capability of the machine within the limits set by the site's operating budget.

To this end, it was decided that computer-based enhancement of the accelerator promised a high degree of benefit for a reasonable cost. A computer-centered, artificial intelligence-based control system was commissioned to address the key problems of expertise loss and cost-effective machine management. The first problem could be solved

by capturing operations expertise in the form of a knowledge-based artificial intelligence system for accelerator automation. The second problem could be remedied by using the computer system as an aid for both operators and site managers in order to monitor and organize machine maintenance. Additionally, the computer system could be designed to 'piggy-back' the existing accelerator control system, allowing for simple, flexible and cost-effective modernization. Of pivotal concern was that the computer system should not replace the operators, but rather aid them in their job and improve their performance and productivity.

The Particle Accelerator Control Expert System (PACES) was conceived to meet the demands of the accelerator site at DREO. It was designed not only to provide assistance for new operators, but also to reduce the operator time required for accelerator start-up, shut-down, and for particle beam maintenance during accelerator runs, allowing operators to perform other, more valuable tasks concurrently with automated accelerator operation ([DeM91], [Lin91], [Lin92a], [Lin92b], [Lin93a], [Lin93b], [Lin94]).

Figure 2-1. Transparent mock-up of a KN-3000. *High Voltage Engineering Corp.*

2.1 An Overview of Particle Accelerator Operation

The KN-3000 is a Van de Graaff particle accelerator that uses the generation of static electric charge to provide particle energies up to 3MeV (mega-electron volts).[7] Figure 2-1 shows a transparent mock-up of the accelerator, and Figure 2-2 illustrates the configuration of the accelerator and its beam line at DREO. As is the case for similar accelerator facilities in general, the DREO accelerator facility consists of a Van de Graaff *accelerator* for producing accelerated charged particles, a *beam line* for conveying these particles to a target area, and various ancillary devices (such as analyzing magnets, Faraday cups, and quadrupoles) used for altering characteristics of the particle beam, such as steering, shaping, filtering and focusing.

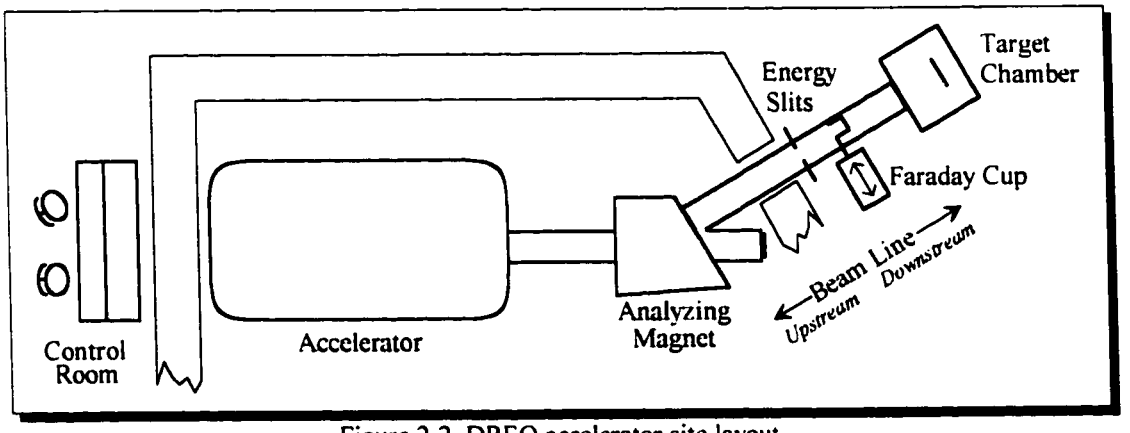


Figure 2-2. DREO accelerator site layout.

The accelerator produces a significant level of ionizing radiation, principally in the form of x-rays. This radiation is termed 'non-persistent' because it quickly vanishes when the machine is shut-down. While the machine is running, however, precaution must be taken to minimize or prevent operator exposure. For this reason, the accelerator tank is isolated from its control panel in a separate room with thick, shielded walls. The beam line runs from the accelerator tank into another shielded room, the target area, where experimental apparatus is located. Typical operating procedure requires that an operator is present at the control panel at all times, while the experimenter moves between target area and control room as needed.

---

[7] One *electron volt* (1 eV) is the work (energy) required to move an electron across a voltage gradient of one volt (1 V).

As shown in Figure 2-3, the Van de Graaff generator (VDG, ❶) employs a moving belt to transport electrical charge from a power supply to the *terminal* (❷) where charge accumulates to build up a high voltage. This high voltage is applied to a series of resistors in the *accelerating column* (❸) to form a voltage gradient which determines the kinetic energy of accelerated particles.[8] Gas flowing into the *ionizing chamber* (❹) is ionized to form a plasma stream of charged particles which is drawn into the accelerating column. The initial generation of this plasma is called a *strike*.[9] The particle beam is accelerated along the voltage gradient and conveyed downstream into the *beam line* (❺), where it is focused, shaped and steered to the target area where experiments occur.

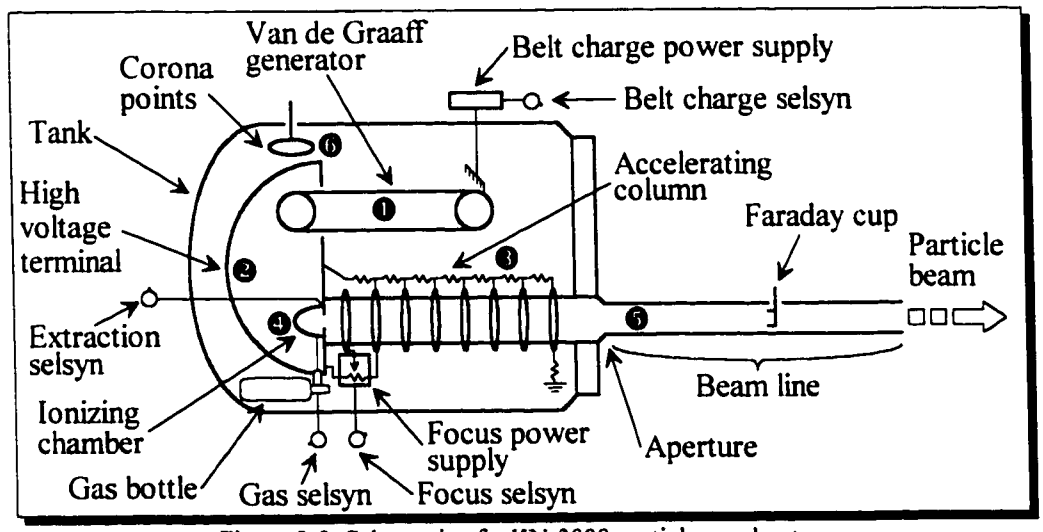


Figure 2-3. Schematic of a KN-3000 particle accelerator.

The *corona points* (❻) are used for terminal voltage stabilization. Their position can be varied to increase or decrease their proximity to the high-voltage terminal. Coronal discharge from the terminal causes some charge to leak away as current through the corona points. This current can be monitored and controlled for terminal voltage stabilization.

Usual operation of the accelerator (a *run*) begins with the operator carrying out a prescribed start-up procedure which may call for re-establishment of a previous run's operating conditions. When the accelerator is to be operated at voltages near its upper limit, start-up may also require *voltage conditioning* to be performed, during which the

[8] For example, a terminal voltage of 1 MV accelerates singly-charged particles to an energy of 1 MeV.
[9] The term *strike* is used because the plasma stream typically first announces its presence by *striking* the Faraday cup and causing a flow of beam current.

terminal voltage is increased in gradual steps towards the target level in order to improve the accelerator's voltage stability. Start-up culminates in the generation of a stable particle beam. Thereafter, *beam maintenance* operations are performed as needed to maintain or alter the characteristics of the particle beam. The accelerator is run in this beam maintenance mode for some length of time, typically several hours. At completion of the accelerator run, the machine is deactivated using a prescribed shut-down procedure. Aside from daily variations in operating conditions and particle beam requirements, start-up is essentially the same process each time. The shut-down procedure seldom varies. Beam maintenance operation, however, depends largely on the nature of the run's experiment(s) as well as on the dynamic behaviour of the machine itself, such as changes in behaviour due to heating, age and condition of components. It may be necessary during beam maintenance operation to perform small parameter changes, such as altering particle energy, beam current or ion species.

In the accelerator at DREO, hydrogen gas (H_2) is ionized to produce protons (H^+) which are accelerated onto a target. A nuclear reaction occurs at the target, producing neutrons that are used for calibration of radiation detectors. The energy of the neutrons is determined by the energy of the incident protons. Hence, to produce neutrons of a specific energy, the operator must set and maintain the accelerator's terminal voltage to a specific value.

Two major problems are associated with this method of generating precise particle energies. First, electromechanical and thermal noise in the accelerator causes fluctuation in particle energy, so built-in control circuitry is used to stabilize the energy level; the control system software must be aware of and be able to direct this closed-loop subsystem. Second, the accelerated particles (Figure 2-4) will not have equal energy (will not be *monochromatic*); most will be of approximately the correct energy, but some will have excessive or insufficient energy (will be *polychromatic*), and it is necessary to screen out these undesired particle energies.

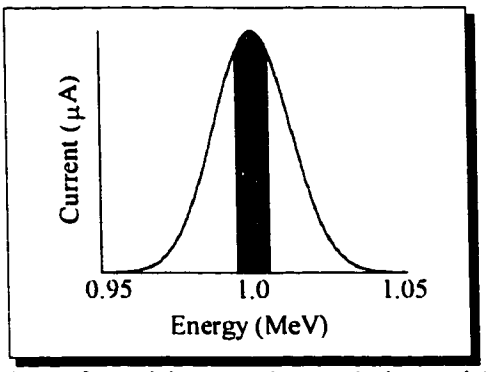


Figure 2-4. Distribution of particle energies (polychromatic). The shaded area contains particles of desired energy to within some margin of error (monochromatic); unshaded areas contain particles of undesired energies.

One useful property of a magnetic field is that it can be used to alter the trajectory of a charged particle. If the strength of the magnetic field is known, one can determine the resulting deflection for a particle of arbitrary momentum.[10] The accelerator therefore uses a *magnetic spectrograph* (an electromagnet, also referred to as an *analyzing magnet*) to focus charged particles of identical momentum and disperses particles of different momenta, ([Eng67]).

During an accelerator run, the operator adjusts the analyzing magnet's power supply output current to isolate particles of a desired energy and to maximize beam current (number of particles reaching the target per unit time). Since magnetic field strength cannot be measured directly, a nuclear magnetic resonance (NMR) technique is employed: The hydrogen atoms of water molecules in the NMR's sensor probe will resonate at a specific frequency determined by the magnet field strength, and by measuring this frequency, the magnetic field strength can be calculated precisely.

The accelerator's beam line, which is usually straight, makes a precise bend at the analyzing magnet (Figure 2-5). When the magnet is set to select a specific particle energy, only those particles of correct energy will be deflected properly to negotiate the turn and reach the target (to be measured as *target current*). Particles with excess energy will not be sufficiently deflected, and will eventually collide with the outer wall of the beam line. Likewise, low-energy particles will be deflected too much, and eventually collide with the inner beam line wall.

---

[10] Momentum is the product of rest mass and kinetic energy. In the sequel, the terms *momentum* and *energy* will be used interchangeably, since only one particle type is used at DREO (i.e. rest mass is constant).

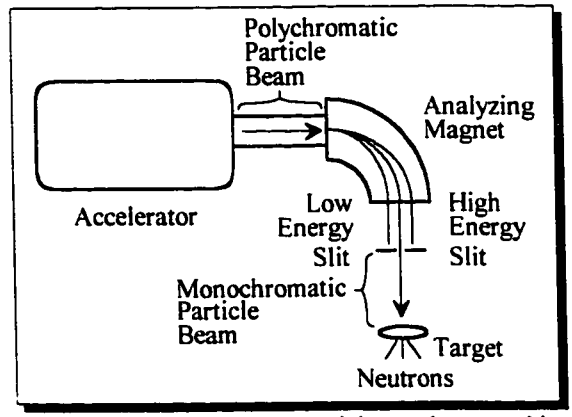


Figure 2-5. Configuration of the DREO particle accelerator with analyzing magnet.

These collisions are used for stabilization and target current optimization by employing the *energy slits* (see Figure 2-5). Particles of proper energy (within some margin of error) will pass between the slits and continue unimpeded down the beam line. Particles of improper energy will strike either the high energy slit or low energy slit, causing current flow. This current flow is measured, and the difference between high and low slit currents indicates the amount of energy imbalance. If the low energy slit current is greater (Figure 2-6a), terminal voltage should be increased to increase average particle energy (and increase target current). Slit balance (Figure 2-6b) implies the majority of particles posseses the proper energy. Yet, if the high energy slit current is greater (Figure 2-6c), terminal voltage should be decreased to decrease beam energy (and increase target current).[11]

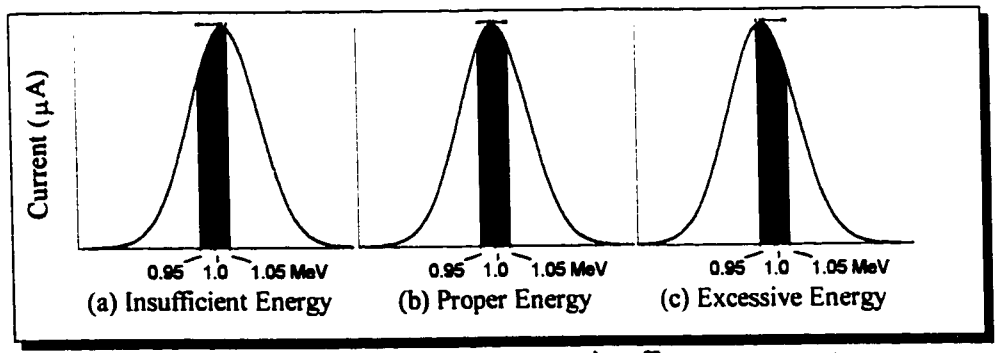


Figure 2-6. Errors in beam energy set-point affect target current.

Figure 2-7 shows how signals from the accelerator and beam line are conveyed to the control panel at DREO. Protons (H^+) emerging from the accelerator's aperture into the beam line possess a wide range of energies (as illustrated in Figure 2-4). The *analyzing magnet* (❶) bends the particle beam through a fixed angle, dispersing particles of different

[11] It is also possible to adjust the magnetic field strength, but this would alter the average energy of particles reaching the target.

energies. The strength of this magnet's field determines the energy of the 'analyzed beam': increasing the field bends higher particle energies onto target; decreasing the field bends lower particle energies onto target. The field strength is linearly proportional to the amount of electrical current flowing through the magnet's coils.[12] The energy slits (❷) screen and measure particles of improper energy. The *voltage stabilizer* (❸) is a self-contained hardware module which controls terminal voltage to maintain proper analyzed beam energy. The voltage stabilizer can monitor either the energy slits or the accelerator's generating voltmeter (GVM, ❹) to determine beam energy (accelerating potential, terminal voltage). Error between the analyzed beam energy and the beam energy set-point is used for corona current adjustment (❺) to fine tune the terminal voltage (and thereby fine tune the analyzed beam energy); increasing corona discharge decreases terminal voltage, and decreasing corona current increases terminal voltage.

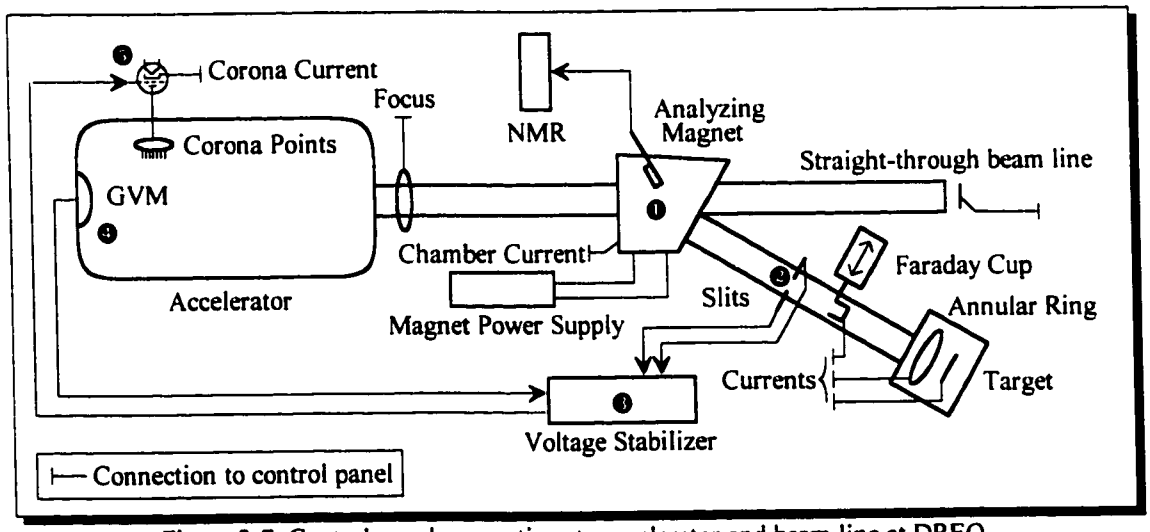


Figure 2-7. Control panel connections to accelerator and beam line at DREO.

◆ ◆ ◆

This section has outlined the basic theory and operation of the particle accelerator and its subsystems. As will be discussed in Chapter 4, the expert system software is responsible for presiding over this collection of subsystems, monitoring and controlling them during automated accelerator operation.

The next section describes how the accelerator's control panel is used during accelerator operation.

---

[12] For example, a specific model of analyzing magnet might produce 2 kG when supplied with 3 A.

2.2 The Control Panel

The accelerator control panel (Figure 2-8) is the arena of human-machine interaction, a window through which operator and accelerator interact. It comprises analog meters and indicator lights for monitoring the state of the accelerator, and selsyns and switches for control.

There are five main switches on the control panel, three used during start-up and shut-down, and two used during the duration of an accelerator run for positioning the corona points. The *control power* switch is a keylock that must be turned on to activate the control panel's power. The *drive motor* switch is used to start or stop the van de Graff belt's drive motor. The *belt charge* switch is used to turn on or off the belt charge power supply. The latter two switches consist of two pushbuttons connected to high-capacity switching relays; one pushbutton energizes and latches the relays, the other de-energizes the relays. The final two switches are for changing the position of the corona points relative to the high voltage terminal: The *corona extend* pushbutton moves the corona points closer to the terminal, and the *corona retract* pushbutton retracts the points.

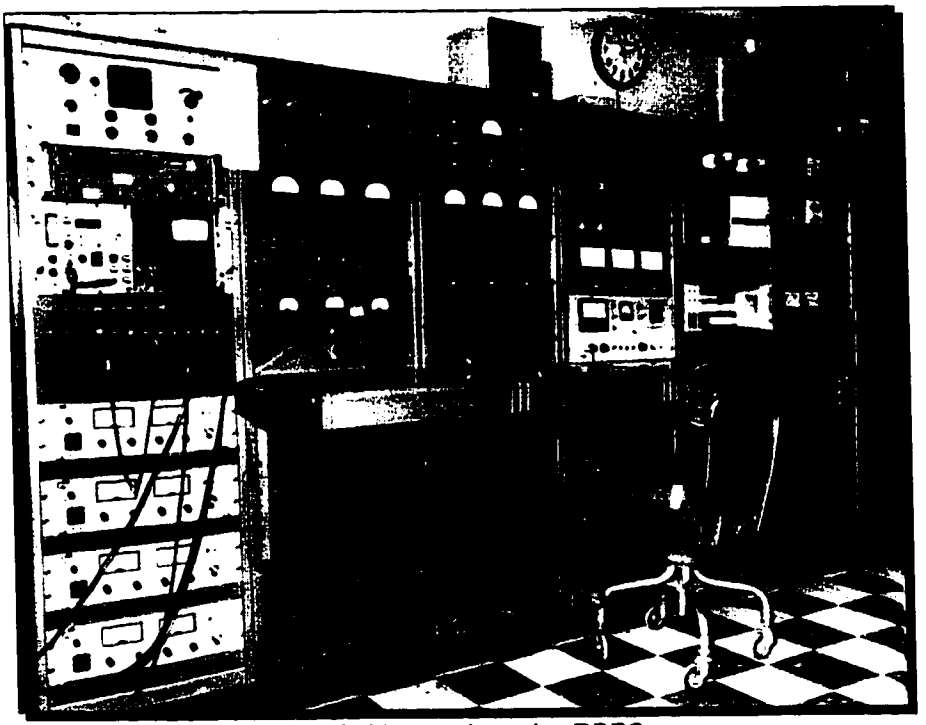


Figure 2-8. KN-3000 control panel at DREO. *Courtesy DREO.*

The control panel's meters are used for monitoring the state of the accelerator. The *vacuum* meter measures the beam line vacuum level. The *terminal voltage* meter indicates the accelerating potential being applied from the high voltage terminal to the accelerating column. The *belt charge* meter displays the amount of charge being deposited onto the VDG's belt by the belt charge power supply. The *column current* meter shows the amount of current flowing through the accelerating column from the high voltage terminal to ground. The current flowing off the high voltage terminal through the corona points is reported by the *corona load* meter, and the proximity of the corona points to the high voltage terminal is indicated on the *corona position* meter. Finally, the particle beam current is displayed on the *beam current* meter, whose signal can be taken from several Faraday cups in the beam line, or from the target chamber.

The selsyns are pairs of electrically-coupled dynamos, one mounted on the control panel and the other located inside the accelerator tank. The latter is usually connected to a rheostat so that turning the control-panel selsyn causes the tank-mounted 'partner' selsyn to turn the rheostat. The accelerator has four selsyns: The *belt charge* selsyn controls the amount of electrical charge that is deposited onto the VDG's belt and carried to the high voltage terminal. The *gas* selsyn adjusts the flow of source gas into the ionizing chamber. The *extraction* selsyn varies the voltage of the ionizing chamber's anode, determining flow rate of ions from the chamber. The *focus* selsyn is used to focus the particle beam as it emerges into the beam line.

The analyzing magnet is a sub-system whose control is straightfoward. Given a required beam energy, it is simple to determine the necessary magnetic field strength using a standard formula.[13] Magnet field strength is directly related to the electric current supplied to the analyzing magnet, and the operator can adjust the magnet's field strength by varying the current output from the magnet's power supply. As mentioned previously, a nuclear magnetic resonance (NMR) magnetometer is employed to measure and display the magnet's field strength, enabling the operator to set the magnet precisely as required to select the desired beam energy. Typically, once the magnet is properly configured, it does not need to be readjusted unless a different beam energy is desired; it is possible,

[13] The magnet field strength b required to bend a beam of protons of energy E, is determined by the formula: $b = (E/e_p)^{1/\Gamma_p}$, where e_p and Γ_p are constants specific for protons.

however, for the magnet's field strength to drift slightly during the course of an accelerator run, but this can usually be rectified by the operator without significant disruption of the beam energy.

As explained in Section 2.1, the voltage stabilizer is responsible for maintaining the terminal voltage (and therefore the beam energy). The voltage stabilizer is a commercial product whose control panel consists of two analog meters (*stabilizer balance* and *corona current*), and several indicators and controls. The principal control is the *stabilizer mode control*, which enables or disables the voltage stabilizer. It has five modes of operation:

- Off: No voltage control is performed.
- Standby: Voltage set-point is registered, but no control is performed.
- GVM: The stabilizer attempts to maintain terminal voltage set-point using feedback from the accelerator's generating voltmeter.
- Slit: The stabilizer attempts to maintain terminal voltage set-point using feedback from the beam line's energy slits.
- Automatic: The stabilizer runs primarily in Slit mode, but switches temporarily to GVM mode when large fluctuations in voltage make slit-based control untenable.

Typically, the operator performs the following steps to enable the voltage stabilizer:

1. The accelerator is started, and a certain terminal voltage is established while the stabilizer is Off. This step may involve belt charge selsyn adjustments, positioning of the corona points, and possibly conditioning.
2. When the voltage appears stable, the stabilizer is set to Standby so that the unit can acquire the voltage set-point.
3. The stabilizer is next switched to GVM, at which time it assumes active control of the terminal voltage. If the accelerator is to be run without using the analyzing magnet, no further action is required.
4. If the accelerator is to be run in conjunction with the analyzing magnet, the analyzing magnet is now configured as required to 'bend' the beam onto target. This step involves determining the analyzing magnet field strength required to select particles of the desired energy.
5. When the analyzing magnet is configured, the voltage stabilizer is switched to Slit, and assumes energy slit-based terminal voltage control.

When the voltage stabilizer is enabled, it attempts to maintain the terminal voltage set-point by adjusting corona discharge current in response to fluctuations in the terminal voltage (as indicated by the balance signal read as the difference between the high and low energy slit currents). Such adjustment is usually sufficient to maintain energy stability, and

therefore maintain the beam on target with the proper energy. Occasionally, however, a large terminal voltage fluctuation occurs (typically in the form of a spark) which is too severe for the voltage stabilizer to compensate. When this happens, the beam gets 'lost' from the target.[14] Usually, the voltage fluctuation is transient, and the voltage quickly (within a few seconds) returns to its set-point without the need for any control adjustments; but, other times, the spark precipitates a period of extreme terminal voltage instability which may require conditioning.[15]

When the beam gets lost, the voltage stabilizer is frequently able to recover the beam after the terminal voltage has returned to its set-point. Sometimes, however, the voltage stabilizer is unable to restore the beam without operator intervention: the operator is obliged to switch the voltage stabilizer to Off, recover the beam manually, and then re-enable the voltage stabilizer. As will be presented in Chapter 4, this method of operating the voltage stabilizer is ideal for automation using knowledge-based reasoning.

During normal accelerator operation, the operator watches the control panel's meters to observe accelerator behaviour and adjusts some or all of the selsyns to alter this behaviour to a more suitable state. Several noteworthy details of manual control are discussed below. Some of these details are elaborated upon in Section 4.4.1 to illustrate how computerization has affected (both improved and hindered) regular manual operation.

Meters: The control panel's analog meters provide good *qualitative* indication of accelerator parameters, but only limited *quantitative* indication. Exact meter readings must, in most cases, be interpolated from a meter's scale. Generally, operators are more concerned with the qualitative indication because it allows them to assess quickly the state of the accelerator based on their expertise; seldom are they concerned with exact numbers for meter readings.

A common problem with the meters is that they sometimes stick (that is, the needles become frozen in place), and operators have learned to tap a meter's faceplate if its needle is suspected to be sticking. This implies that meter readings are not always reliable, and it

[14] That is, the terminal voltage is no longer controlled at set-point, so the beam either is bent too much by the analyzing magnet, or not bent enough, and fails to strike the target. If the voltage has deviated to a large degree, the beam may not even strike the energy slits, making further slit-based control impossible without remediation.

[15] Sometimes the spark can even cause physical damage to the accelerator which may require shut-down and repair.

is possible that novice operators might be unaware of the sticking problem or how to correct it.

Finally, the meters have built-in integration, implying that high frequency signals are lost. Prior to the advent of digital signal acquisition, operators had to rely solely on the integrated meters, and therefore were not exposed to many subtle, short-duration transients exhibited during accelerator operation. Operators were able to form hypotheses or hunches about what was really happening inside the accelerator but were unable to observe such behaviour at the control panel.

Selsyns: The control panel's four selsyns are cross-coupled non-linear control points, so altering the set-point of one can affect the set-point of another (Figure 2-9). For example, when the accelerator is started, the operator turns the belt charge selsyn to establish a specific terminal voltage. Next, the extraction selsyn is turned to increase beam current, but this causes a drop in terminal voltage, so the belt charge selsyn must be re-adjusted.[16] Similarly, as the gas selsyn is varied, beam current can change, perhaps requiring a re-adjustment of the extraction selsyn. Finally, the amount of focus required depends on the accelerating potential, so varying the terminal voltage can necessitate focus selsyn compensation.

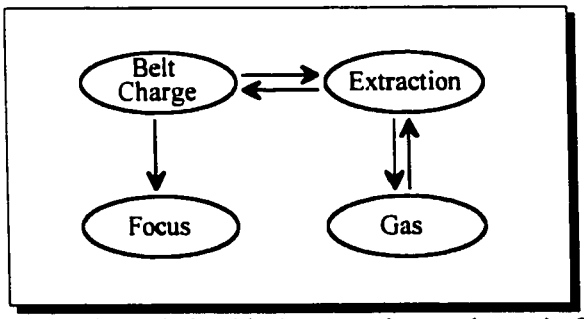


Figure 2-9. Cross-coupling relationship between the accelerator's four main control points (selsyns).

Since the selsyns possess such cross-coupling, operators frequently need to make adjustments to two or more selsyns simultaneously. Without the presence of automatic control, a single operator is never able to adjust more than two selsyns concurrently.

[16] The operator may also initially set the terminal voltage slightly higher than required so that when establishment of the particle beam causes the terminal voltage to drop, the terminal voltage reaches the desired level, and no belt charge selsyn compensation is needed.

There are two important characteristics of the selsyns. First, they provide tactile feedback to the operator, in the form of counter-torque, enabling the operator to sense when the associated rheostat has reached its upper or lower limit. Second, a control-panel selsyn can 'slip' when its tank-mounted counterpart fails to track it properly; this can lead to occasions when the setting indicated on the control panel selsyn is ambiguous.

2.3 Other Accelerator Sites

As mentioned in Chapter 1, two other accelerator sites were used during development of PACES: The KN-3000 at McMaster Accelerator Lab, McMaster University was used as a testbed for the implementation of the proof-of-concept prototype. The KN-3000 at DREO was then used for the first field version of PACES. Finally, the KN-4000 at the Whiteshell Labs of Atomic Energy of Canada Ltd. (AECL) was used as the platform for a second, updated field version of PACES. Table 2-1 summarizes the differences between these three accelerators. The following sections describe how the two ancillary accelerator facilities differ from the DREO accelerator site.

	DREO	**McMaster**	**Whiteshell**
Model	KN-3000	KN-3000	KN-4000
Maximum voltage (MV)	3	3	4.7
Source gas	H_2 / D_2	H_2 / ^3He / ^3He+^4He	H_2
Tank gas	CO_2+N_2 / SF_6	SF_6	SF_6
Analyzing magnets	1	2	1
Beam Lines	1	3	2
Faraday cups	None	3	3
Quadrupoles	None	6	2
Steerers	None	4	1

Table 2-1. Comparison of the accelerators at DREO, McMaster University and AECL Whiteshell Labs.

2.3.1 The KN-3000 at McMaster Accelerator Lab

A twin sister of the DREO KN-3000, the McMaster KN-3000 is of approximately the same age, and although the two machines are functionally equivalent, their beam lines differ significantly, and their operating regimens differ drastically. Whereas the DREO accelerator is used almost exclusively as a neutron generator whose operating parameters hardly differ from run to run, the McMaster accelerator exists in an academic institution

and has experienced a long and diversified career. During its lifetime, the McMaster accelerator has seen countless researchers, students and technicians using it for a wide range of experiments, including such things as Rutherford backscattering, gamma ray spectroscopy, neutron irradiation studies and various medical science experiments.

Since the McMaster accelerator was used as a testbed for PACES early in its development, much of the hardware that distinguishes the McMaster accelerator from its DREO counterpart was irrelevant; the main control points (the selsyns and switches) are identical on the two machines, enabling much of the early development to be performed at McMaster.

2.3.2 The KN-4000 at Whiteshell Labs

Whiteshell Labs is a research facility of Atomic Energy Canada Ltd. (AECL), situated near the town of Pinawa in eastern Manitoba. A KN-4000 particle accelerator is operated there by the Reactor Materials Research section to generate proton beams at up to 4 MeV which are used for "various types of radiation damage studies to measure the effects of prolonged irradiation on [nuclear] reactor materials", ([McI96]).

The KN-4000 accelerator is a newer, enhanced version of the KN-3000 which is able to generate higher accelerating voltages (up to 4.75MeV as opposed to 3MeV). The principles of operation remain the same, as do the basic control mechanisms and operating procedures. In the case of the Whiteshell Labs KN-4000, however, voltage conditioning is regularly employed to stabilize the accelerating potential at 4 MeV and higher; without conditioning, the accelerating potential is unstable and prone to sparking. Conditioning is performed after the machine's tank has been opened for maintenance, whenever the accelerator exhibits periods of voltage instability, and always after the machine has been idle for more than a few days (such as after holidays).

A large number of the experiments currently carried out involve irradiation of zirconium alloy samples to simulate conditions inside CANDU nuclear reactors. The samples are typically heated to approximately 300C by supplying them with dc power. A standalone control unit is used to maintain the temperature set-point. When the samples are irradiated by the accelerator's particle beam, less dc power is required to maintain the

sample temperature (due to heating caused by the particle beam). It is a simple matter to monitor the dc power output of the control unit (the *sample power*) to determine whether the beam is on target. Without the availability of this passive (indirect) method of detecting presence of the beam on target, active 'beam sampling' must be performed periodically to determine whether the beam has been lost from the target. Beam sampling requires insertion of a Faraday cup (or other monitoring device) into the particle beam, which diverts the beam from target. During some experiments, it is desirable to minimize such disruption of the beam, or eliminate it altogether. Monitoring the sample power avoids the need for such beam sampling: a sudden, pronounced rise in sample power indicates the beam has been lost from the target. This capability is of great use to the beam maintenance knowledge base described in Chapter 4.

Another feature of the Whiteshell Labs KN-4000 is the *tank ripple* signal, which indicates the first-order time derivative of terminal voltage. This signal is available as an oscilloscope trace on the control panel, and is used by the operators to determine terminal voltage stability visually. It is likewise used by the expert system to infer instability (see Chapter 4).

Of minor importance is the fact that various controls and indicators on the control panel have names different from those used at DREO. For example, the extraction selsyn is referred to as the "beam bias selsyn", and the belt charge selsyn is called simply the "charge selsyn". In the sequel, the DREO-site terms *extraction selsyn* and *belt charge selsyn* will be used in preference to their Whiteshell-site alternatives.

The Whiteshell Labs accelerator facility differs further from the DREO and McMaster sites in that the KN-4000 is generally run continuously, around the clock, five days a week. Consequently, as reported in [McI96], "[a]n automated, or partially automated [accelerator control] system would allow experimenters more freedom to do other work in the area during the day, and would eventually allow periods of unattended operation, if necessary. This could reduce operating costs in the future." To this end, AECL became interested in acquiring a customized version of PACES which would pave the way to automation of the Whiteshell Labs' accelerator facility.

2.4 Related Research

The area of computer-based particle accelerator control is currently blossoming. A large number of research projects have recently been reported concerning various aspects of computer-based control for particle accelerators and other related machines (such as linear accelerators, cyclotrons and storage rings). Some documentation of general accelerator control can be found in [Bar95], [Dic95], [Epa95], [Lin88], [Riv95], [Sil88], [Sta95], [Wan94], [Wu95], [Yos95] and [Zha95]. Research on applying several forms of artificial intelligence to accelerator control is published in [Jen94], [Jen96], [Lew95], [Mej95], [Rod97], [Ryb95], [Sak95], [Tan95] and [Wes95]. Some projects investigating the development of object-oriented class libraries and toolkits are reported in [Bir95], [Che95], [DiM95] and [Kuz95].

* * *

This chapter has introduced the KN-3000 and KN-4000 particle accelerators and described how they function and are operated. As discussed in Chapter 1, the intention is to perform a computer-centered technology insertion exercise as a means of modernizing the operation of this type of particle accelerator. In the next chapter, four important areas of electrical and computer engineering are surveyed in order to set the stage for a presentation in Chapters 4 and 5 of how PACES was designed and implemented to accomplish this modernization task.

Chapter 3

The Tools for Technology Insertion

As mentioned in Chapter 1, the computer-centered technology insertion exercise described in this thesis is based upon the hybridization of four distinct areas of computer and electrical engineering. This exercise can be thought of as a jigsaw puzzle (Figure 3-1) which relies on these four diverse areas to accomplish a seamless interface between computer and machine, and between computer and human. At the 'low end', principles of real-time systems and instrumentation interfacing are applied to connect the computer system to the machine to effect data acquisition and control. At the 'high end', principles of human factors, primarily in the sub-domains of human-computer interaction and computer-human interface, are used to 'connect' the computer system with the human operator. Between these two ends of the interface, artificial intelligence is used for problem solving and decision making in order to: ① Automate machine operation by mimicking the skills of expert human operators; and ② Assist in machine operation by augmenting the skills of novice human operators. Within this jigsaw puzzle analogy, software engineering can be thought of as the 'glue' that holds the pieces of the puzzle together; without such principles as modularity, information hiding and object-oriented structuring, it would be an extremely difficult task to meld the different pieces into a working system.

In the following sections, each of these areas is introduced and delimited within the scope of this thesis. Literature reviews serve as background material that will be cast appropriately to form a framework for the following chapters.

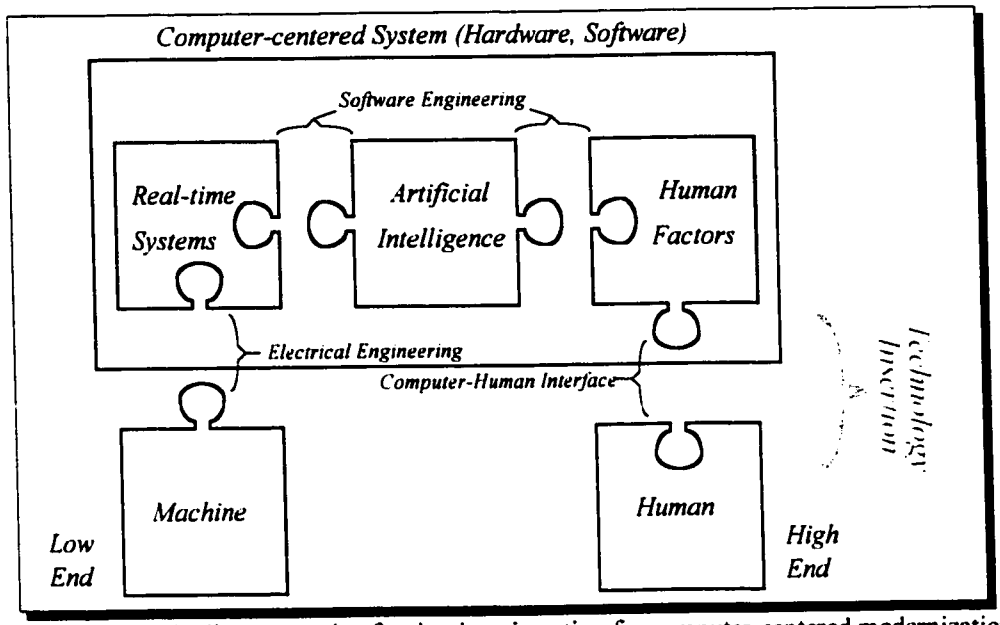


Figure 3-1. The jigsaw puzzle of technology insertion for computer-centered modernization.

3.1 Real-Time Systems

During the industrial revolution that swept Europe in the 19th century, specialized machines were successfully used for controlling and automating a variety of manufacturing plants and factories. For example, a machine developed in France could control the pattern of textiles woven on a loom: the machine would read a series of punched cards and interpret the arrangement of holes to alter the pattern being woven. Many similar schemes were conceived for controlling a wide range of tasks, such as wood lathing, cannon boring and sawmilling. Such automation was even used for entertainment, in, for example, the player piano that played music recorded as patterns of holes on a roll of paper. More recently, during the early years of the computer revolution, it was realized that electronic computers were well suited for controlling and automating many kinds of processes and systems; they offered many of the benefits of older, mechanical automation systems, but also promised increased speed, accuracy, flexibility, and reliability.

A difficulty arises, however, when attempting to reconcile the different *time domains* of computers and real-world machines. Computers are digital in nature, and perform their duties in discrete steps, changing from one state to another in a prescribed, time-determinate way. That is, a computer requires a specific amount of time, its *sample rate*, (τ) to change from one state to another, and cannot instantaneously change states.

This means that any external information that changes during the interval between times t and $t+\tau$ cannot be measured directly.[17] In contrast, real-world processes generally operate continuously in time with no notion of time-determinate steps. For example, a real-world variable may change dynamically and continuously as shown in Figure 3-2a, but a computer with a sample rate of τ will only 'see' (be able to sample) the variable at discrete times, resulting in approximation and loss of information (Figure 3-2b).

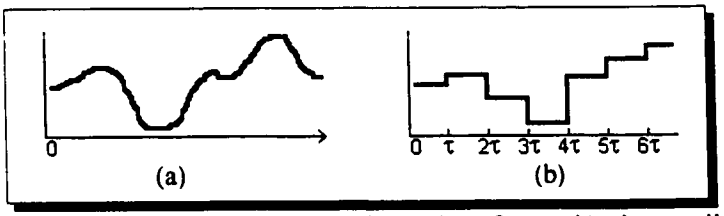


Figure 3-2. Comparison of time domains. Waveform (b) is a discrete-time approximation of the continuous-time waveform (a).

The disparity between the continuous and discrete time domains causes problems when a discrete-time computer is required to monitor and/or control a continuous-time process. These problems arise because it is necessary to *transform* information that is interchanged between the process and the computer. Process state information transferred from the process to the computer must be discretized (digitized). Likewise, control information transferred from the computer to the process must be transformed from the discrete-time digital domain to the continuous-time domain. Because the discrete-time (*digital*) domain is a subset of the continuous-time (*analog*) domain, there is approximation (information loss) in moving from continuous to discrete, and lack of specificity (precision) in moving from discrete to continuous. This means that measured information input to the computer is inexact, and generated information output from the computer is lacking in complete coverage of range. Such inexactness can occur in the *value* or *time* components of measured process state information and generated control information. The time component of measured process state information is the time at which the measurement was taken, whereas the time component of generated control information is the time at which the control action should be performed. In either case, transformation between time domains results in problems in both accuracy and timing.

[17] It is possible, of course, to employ instead some device whose sample rate τ' is less than the τ of the computer, but clearly there exists a time interval $[t,t+\tau']$ in which changes cannot be measured as τ' approaches 0.

Computer systems that are interfaced with real-world processes and that address these problems of accuracy and timing are called *real-time systems*. Several definitions of real-time systems have been presented in the literature. Four representative definitions are given below, and the reader should note that although they may seem similar, they are subtly different and offer alternative viewpoints as to the definition of a real-time system. For example, the *Oxford Dictionary of Computing* (as cited in [Bur90], p. 2) supplies this definition:

> "*Real-time system*: Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to the same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness."

In contrast, Lawson ([Law92], p. 2) defines a real-time system as follows:

> "By *real-time system* we mean a system that assures that controlled activities 'progress' and that stability is maintained and further, that the values of outputs *and* the time at which the outputs are produced are important to the proper functioning of the system."

Similarly, Young ([You82]) defines a real-time system as:

> "Any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period."

Finally, the German industry standard DIN 44300 (cited in [Hal92]) defines 'real-time operation' as:

> "The operating mode of a computer system in which the programs for the processing of data arriving from the outside are permanently ready, so that their results will be available within predetermined periods of time; the arrival times of data can be randomly distributed or be already *a priori* determined depending on different applications."

As described by Halang ([Hal92]), real-time computer systems are "associated with external processes. The program processing must be temporally synchronised with events occurring in external processes and must keep pace with them". Figure 3-3 captures the basic structure of a real-time system, showing how the system interacts with the real world process using sensors and actuators, and perhaps interacting with a human operator.

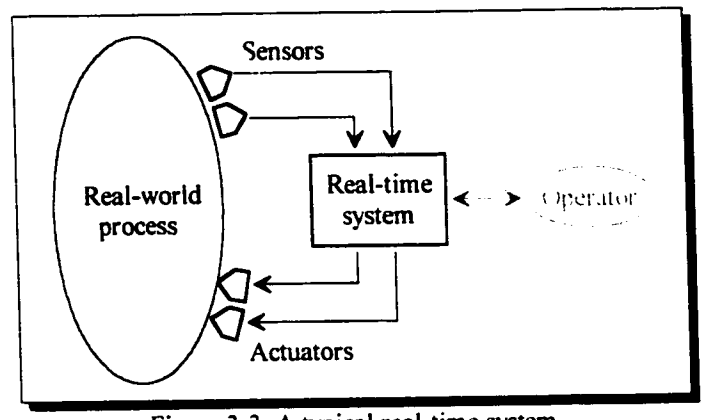


Figure 3-3. A typical real-time system.

Real-time systems generally fall into two classes of real-time environments: *hard* and *soft*. The former environment requires that the real-time system must "respond in time and cannot cease operation even for a moment", ([Hal92]). In contrast, a soft real-time environment requires that the "computer should on average respond quickly, but occasional delays are acceptable", ([Hal92]). Lawson ([Law92], p. 2) gives a similar definition of the two classes:

> "In *soft real-time systems*, the provision of a satisfactory degree of *service* is central and, while important, catastrophes will not result if the service is temporarily degraded. On the other hand, in the more restrictive *hard real-time system*, if a correct output is not available by a specific deadline, a significant, perhaps catastrophic result(s) may occur. In hard real-time systems *predictability* becomes the essential issue."

A special sub-class of real-time systems are those in which the computer system is physically incorporated into the real-world process, such as computer-controlled automobile ignition and anti-lock braking systems, videotape machines, and aircraft avionics. Lawson ([Law92], p. 3) describes these *embedded systems* as follows:

> "In many of the real-time system environments (particularly hard environments), there is an increasing trend to incorporate sophisticated processing directly into products (that is, *embedded systems*)."

Burns and Wellings ([Bur90], pp. 7-13) identify several basic characteristics of real-time systems in general:

- *Largeness and complexity*: Real-time systems tend to be large and complex because they need to deal with a wide variety of real-world events that are also complex and continuously changing.

- *Extreme reliability and safety*: Real-time systems need to possess a high degree of reliability and safety because they can affect real-world objects, including such things as expensive property and equipment, and human lives. Due to their inherent complexity, such real-time systems are highly vulnerable to software bugs that can result in damaging failure, or even catastrophe.
- *Concurrent control of separate system components*: A real-time system comprises a complex and simultaneous interaction between computers and real-world objects. Since the real-world objects naturally co-exist concurrently, the computers are also required to operate concurrently, calling for an implementation language that supports a high level of parallel processing.
- *Real-time response*: Because response time is a crucial requirement of real-time systems, they must be designed and implemented to produce the appropriate outputs at the appropriate times, no matter what the conditions.
- *Real-time control*: The real-time system's implementation language and run-time software are required to permit several important real-time control actions. The system must be able to perform and complete specified actions at specified times. Moreover, the system should respond properly when timing requirements change dynamically or cannot be met completely.
- *Manipulation of real numbers*: The implementation language should provide support for manipulation of floating-point (real) numbers because control algorithms can be mathematically complex and require high precision.
- *Interaction with hardware interfaces*: Because real-time systems are intimately coupled with real-world objects, they need to employ sensors for monitoring, and actuators for controlling, the real-world objects with which they are connected.
- *Efficient implementation*: The points listed above, especially that of real-time response, imply that the real-time system needs to be implemented, and perform its duties, in a highly efficient manner.

Real-time systems, in general, deal directly with real-world applications such as industrial process control and manufacturing, signal, image and graphics processing, and the so-called 'command, communication and control' (C3) applications, including computer operating systems. As such, these systems can involve such things as expensive equipment and property, or hazardous materials and environments, and they can therefore directly and profoundly affect human lives. Consequently, to be successful, real-time systems must rely heavily on the attributes of efficiency, timeliness, predictability, reliability, fault tolerance and safety, as will be discussed in the sequel.

3.1.1 Real-Time System Structure

The basic building blocks of real-time systems are *processes* (Figure 3-4a), which can either be physical real-world entities (hardware) or programmed mechanisms (software). The real-time system is formed as a collection of interconnected processes (Figure 3-4b), one or more levels of software built upon and interacting with the underlying hardware level. As mentioned previously, the concurrent nature of real-world processes requires that the real-time system also function concurrently, executing several different activities in parallel to accomplish the overall real-time task. These activities are performed by the software processes,[18] which are executed by the *processors* (computers). A real-time system, therefore, contains a collection of one or more processors that perform *multiprocessing* to execute the concurrent processes of the real-time application. If the system has only one processor, it must provide a multi-tasking scheduler to execute concurrent processes in time-shared parallelism. If the system has multiple processors, it must have mechanisms for inter-processor communication, process-to-processor allocation and processor load balancing. Additionally, the real time system must have mechanisms for inter-process synchronization and communication.

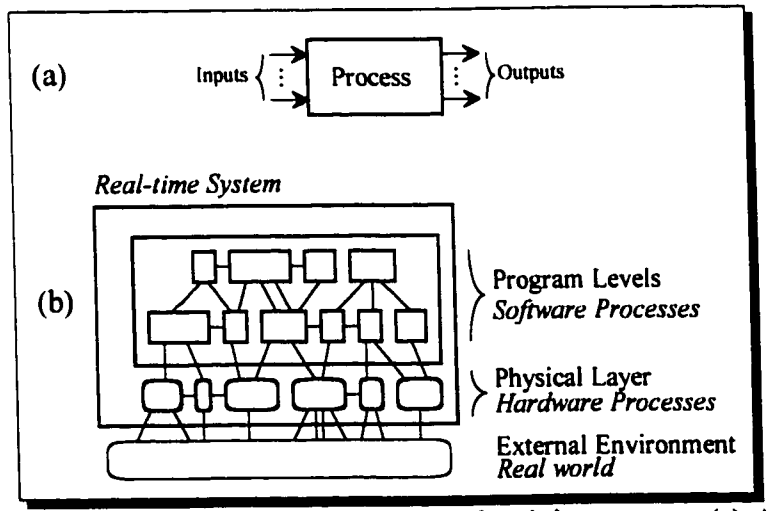


Figure 3-4. Processes are the building blocks of real-time systems. (a) A process generates outputs in response to inputs. (b) Software and hardware processes are interconnected to form a real-time system. *After [Law92], p. 15*

According to Lawson ([Law92], pp. 34-35), processes have several different properties. In general, a process possesses a *deadline* requiring that a certain operation (or

[18] The word *process* in this context is a computer science term not to be confused with the alternative use in this thesis of the word *process* used in such terms as *process management*, *complex process*, and *real-world process*. In the remainder of this section, *process* should be interpreted as *software process*.

set of operations) is completed (or started) at a certain point in time. Processes (and their deadlines) can have importance relative to one another: A *critical* process must meet its deadline or else possibly cause catastrophe; an *essential* process has a deadline important for system operation which will not cause a catastrophe if missed; finally, a *nonessential* process has a deadline which, if missed, does not affect the system's short-term operation, but may affect the system's long-term operation. Additionally, a process can be *periodic*, meaning that it is executed repeatedly at regular points in time, or it can be *aperiodic*, being executed at zero or more arbitrary points in time. Furthermore, a *static* process is one that is created when the system is started and exists for the duration of the system's execution; a *dynamic* process is created or destroyed as needed during system execution.

A real-time system as a whole should exhibit several qualities important for proper real-time performance, ([Law92], pp. 36-37). It must be *predictable*, meaning that processing time lies within maximum time limits in the worst case. The system may also be required to be *deterministic*, meaning that, under all possible situations, its behaviour is exactly predictable. Determinism, however, is in practice usually too difficult or costly to implement. In addition to predictability, the execution of system processes must be *timely* to ensure that processes are executed at the proper times. This subsumes efficient and effective multiprocessing that is free from such problems as process starvation (denial of execution) and deadlock (inter-process resource contention). Finally, the system may need to provide mechanisms for inter-process *synchronization* and *communication* to facilitate cooperative information flow and parallelism.

Figure 3-5 shows the components of a typical real-time system, both internal and external to the computer. A real-time clock is used for timekeeping and synchronization of events. A database may be generated and queried by the computer. External display devices, the operator's console, and perhaps a remote monitoring system are used for communicating and interacting with operators. The computer system and real-world process are connected through a real-time monitoring and controlling interface. Inside the computer system, different software modules are responsible for coordinating the external components and exchanging information between them. In this sense, the external

components can be thought of as *resources* that the computer system must utilize and manage effectively to accomplish its real-time duties.

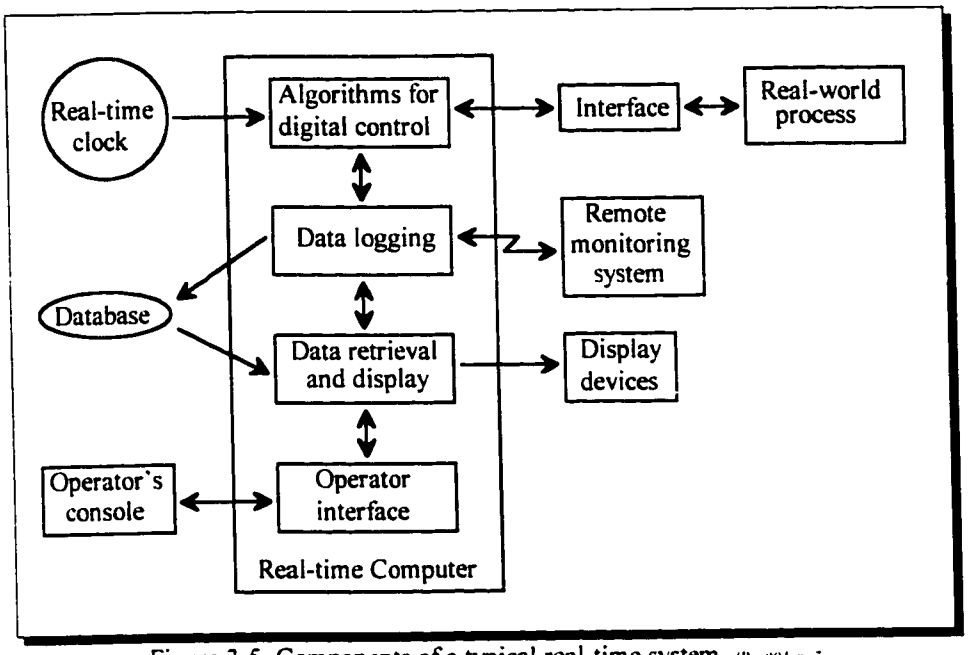


Figure 3-5. Components of a typical real-time system. [Burns, p.

In the past, real-time systems have been traditionally implemented in assembly language because assembly language easily facilitates the desirable properties of predictability, determinism and timeliness. System developers found assembly language convenient for analysis to prove system compliance to timing specifications, therefore proving predictability, determinism and timeliness. As real-time applications became larger and more complex, however, developers were forced to turn to higher-level languages to accommodate the demands placed by these more complex applications. Several languages have been specifically designed for real-time system implementation, such as Coral 66 and RTL/2, ([Ben88], pp. 315-318). Other higher-level languages, such as occam ([Ell91]), Modula-2 ([Ben88], pp. 318-337), and Ada (*Ibid*, pp. 337-338), incorporate features suitable for concurrent programming and embedded computer system development.

3.1.2 Reliability, Fault Tolerance and Safety

Over a nineteen month period in 1992-93, six patients undergoing radiation exposure for cancer treatment were inadvertently exposed to dangerously high radiation levels from Therac-25 radiation therapy machines. At least two of these people died as a direct result

of this over-exposure, and it is entirely likely that the lifespans of the other victims, although all were terminally ill, may have been needlessly shortened and their quality of life reduced. It was subsequently discovered, ([Lev93]), that the cause of the over-exposure was a programming error which rendered the machine's computer control system unreliable under certain rare and obscure circumstances. The problem was not recognized during system development because the computer programmer was unaware of salient mechanical engineering details of the machine's radiation exposure mechanism. The problem was not detected during routine testing because it could only arise through a specific sequence of actions by the machine operator which involved a data entry error. The unfortunate consequences of this design flaw were human suffering and loss of life. The Therac-25 accident is, therefore, a sad reminder of the extreme importance of requiring and ensuring that systems are engineered to be *reliable*, *fault tolerant* and *safe*.

Burns and Wellings define the *reliability* of a real-time system to be: "A measure of the success with which the system conforms to some authoritative specification of its behaviour", ([Bur90], p. 93). They also define *failure* to be: "When the behaviour of the system deviates from that which is specified for it, this is called a failure", (*Ibid*). Failures, they write, are the external manifestations of internal problems, called *errors*, which are caused by algorithmic or mechanical *faults*. Three types of faults can be characterized: A *transient fault* occurs at a particlular time, existing for a period of time before disappearing; a *permanent fault* occurs at a particular time and exists until remedied; an *intermittent fault* occurs aperiodically from time to time. Clearly, it is imperative that all three types of faults must be prevented from causing errors and system failure. Two approaches are commonly used for preventing errors and failures: *fault prevention* and *fault tolerance*.

Fault prevention entails two stages. First, *fault avoidance* is used during system development to limit inclusion of potentially faulty components. Second, *fault removal* is pursued, in which formal methods are applied for finding and then eliminating causes of error, ([Bur90], p. 95).

Despite the efforts of fault avoidance and removal, components of the system will eventually fail. That is, mechanical components wear down, and software systems frequently contain elusive bugs which eventually manifest themselves. Fault prevention, therefore, will not be successful if repair and maintenance efforts are irregular or insufficient. Consequently, system reliability relies largely on *fault tolerance*, the ability of a system to continue operation regardless of faults. Burns and Wellings ([Bur90], p. 96) list three general levels of fault tolerance that a system may employ, possibly in combination: With *full fault tolerance*, also called *fail operational*, "the system continues to operate in the presence of faults, albeit for a limited period, with no significant loss of functionality or performance". A system using *graceful degradation* (or *failsoft*) fault tolerance "continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair". Finally, a *failsafe* system "maintains its integrity while accepting a temporary halt in its operation". The requirements, nature and application of a specific system will dictate which combination of fault tolerance levels should be implemented.

The notion of system reliability is closely related to the general concept of *safety*, which Leveson ([Lev86]) defines as: "Freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm". Within the domain of software systems, Leveson further defines a *mishap* as "an *unplanned event* or *series of events* that can result in death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm".

Burns and Wellings ([Bur90], p. 120) contrast the possibly conflicting concepts of *safety* and *reliability* as follows:

> "Reliability has been defined as a measure of the success with which a system conforms to some authoritative specification of its behaviour. This is usually expressed in terms of probability. Safety, however, is the probability that conditions that can lead to mishaps do not occur *whether or not the intended function is performed.*"

In practice, safety and reliability are actually two aspects of system *dependability*, which Laprie ([Lap85]) defines as follows:

> "The dependability of a system is that property of the system which allows reliance to be justifiably placed on the service it delivers".

As illustrated in Figure 3-6, the notion of dependability encompasses the system qualities of reliability, safety and security. All three aspects should be addressed in the design and implementation of a real-time system.

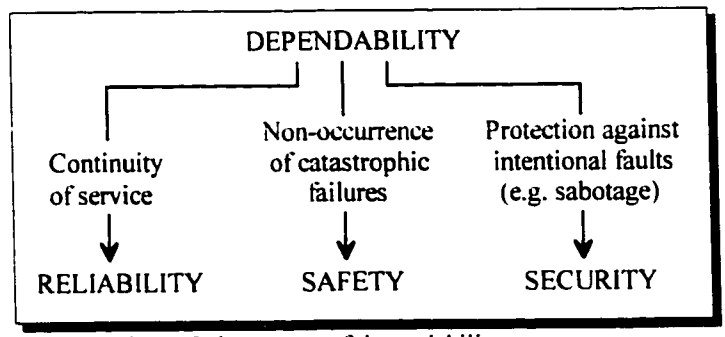


Figure 3-6. Aspects of dependability. *[Bur90], p. 120*

◆ ◆ ◆

This section has presented an overview of real-time systems and outlined the principal considerations that must be taken during their design and implementation. The next section introduces two forms of artificial intelligence that are suitable for use as reasoning and decision-making engines within real-time systems.

3.2 Knowledge-based Reasoning

Complex systems such as aircraft, nuclear reactors, and particle accelerators are difficult to control using conventional ('dumb') controllers because such controllers are not accommodating to large bodies of operational knowledge (human expertise). Although conventional controllers may be used to automate various *sub*systems, they are not well suited for overall control of complex systems. Historically, such systems have been supervised by human operators, yet increasingly larger subsystems have been relinquished to automatic control. As systems have become more complex, humans have become increasingly less able to maintain fast, error-free and all-encompassing supervision. This has led to the incorporation of computers into control systems to improve response time and reduce errors. But computers have lacked the human characteristics of knowledge-based reasoning, adaptive learning and experience. Only during the past 15-20 years have computer systems been given 'knowledge' to aid in their

control tasks. Such efforts have included techniques in *expert systems*, *neural networks*, and *fuzzy logic*.

Kosko ([Kos92]) has illustrated how these three types of 'artificial reasoning' are related within the framework of artificial intelligence (Table 3-1). All three paradigms are *model-free estimators* in that they are able to estimate a function without relying on a mathematical model of how the function's outputs depend on its inputs. Moreover, they are *adaptive* because they can 'learn' from experience.

	Symbolic	*Numeric*
Structured	Expert Systems	Fuzzy Systems
Unstructured		Neural Systems

Table 3-1. Types of artificial reasoning. [Kos92]

Fuzzy and neural systems are inherently *numeric* in nature, meaning that they use numeric operations to manipulate information stored as numbers. Expert systems, contrarily, are *symbolic* in nature, using logical operations to manipulate information consisting of symbols rather than numbers. On conventional computers, numeric systems typically execute faster, and sometimes more accurately, than symbolic systems, but the latter are sometimes more convenient to use for storing and processing the factual relationships that comprise complex information and abstract concepts. Expert and fuzzy systems share the feature of using *structured knowledge*, meaning that the 'knowledge' they possess is rigidly structured and delineated. It is possible, therefore, to use the term *knowledge-based system* (KBS) to encompass both expert and fuzzy systems. In contrast, neural systems use unstructured knowledge that is not required to hold any well-defined form, and thus neural systems need not be considered knowledge-based systems. A corollary of this distinction is that knowledge-based systems are inherently more amenable to inspection than neural systems. That is, it is easier to 'look inside' knowledge-based systems to see what information they store and how it is being processed. Neural systems have been likened to 'black box' functions that map inputs to outputs in a way not readily evident to inspection, (Figure 3-7a). A knowledge-based system, in contrast, can be analogized as a 'glass house' in that its internal information structure and processing are quite evident and inspectable, (Figure 3-7b). This property of 'transparency' is an

important one, and has great influence on the choice of which AI-reasoning paradigm to use for a given task. There are applications for which it is necessary that the decision-making process be available for observation and explanation at all times. In such applications, neural systems lack the needed transparency. It is for this reason that, although widespread success has been reported in using neural systems for control applications, they have not been employed in the present particle accelerator control system. Consequently, neural systems are outside the scope of this thesis, and will not be discussed further. The remainder of this section presents background material on knowledge-based systems for process control, namely expert and fuzzy systems.

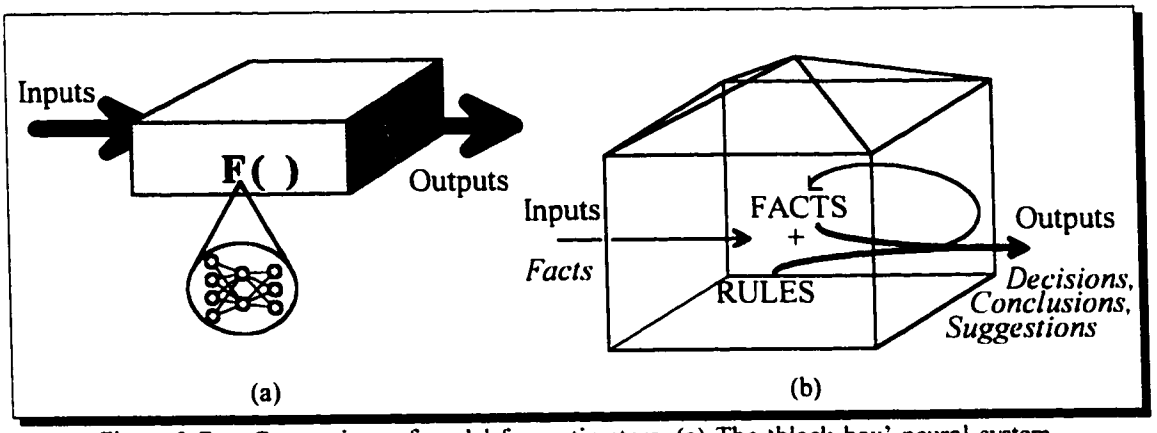


Figure 3-7. Comparison of model-free estimators. (a) The 'black box' neural system. (b) The 'glass house' knowledge-based system.

Since the infancy of AI, many research projects have investigated the prospects of using AI-based decision making for solving process control problems. A survey paper ([Tau89] cited in [Dri93], p. 14-17) reports on several experimental AI systems that demonstrate the applicability of knowledge-based systems to four major areas of process control: process monitoring, fault diagnosis, planning and scheduling, and supervisory control. Other papers report success in using knowledge-based systems for a variety of control problems, such as cement kiln operation ([Lar81], [Umb81]), high-speed subway train operation ([Yas85]), integrated circuit manufacturing, robot manipulator control, mobile robot navigation, and computer disk drive control ([IEE92]).

A general definition of knowledge-based systems applied to closed-loop control problems is taken from [Dri93], p. 1:

> "A KBS for closed-loop control is a control system which enhances the performance, reliability, and robustness of control by incorporating knowledge which cannot be accommodated in the analytic model upon which the design of a control algorithm is based, and that is usually taken care of by manual modes of operation, or by other safety and ancillary logic mechanisms."

Knowledge-based systems used in process control can be classified into four categories relating to their area of intended use: manual control assistance, plant-wide tuning, quality control, and expert control. A KBS used for expert control is called a *knowledge-based controller* (KBC), and is defined as "a highly specialized KBS designed for performing a specific task during a particular phase of the lifecycle of a process control system", ([Dri93], p. 17). The role of a KBC in applications for expert control of complex, human-tended processes is twofold: First, knowledge-based reasoning can be employed for control *per se*, be it open- or closed-loop; this type of system, shown in Figure 3-8a, is termed a *direct expert control system* (DECS). Alternatively, the KBS can be a *supervisory expert control system* (SECS), presiding over and directing a conventional controller in a supervisory capacity (Figure 3-8b).

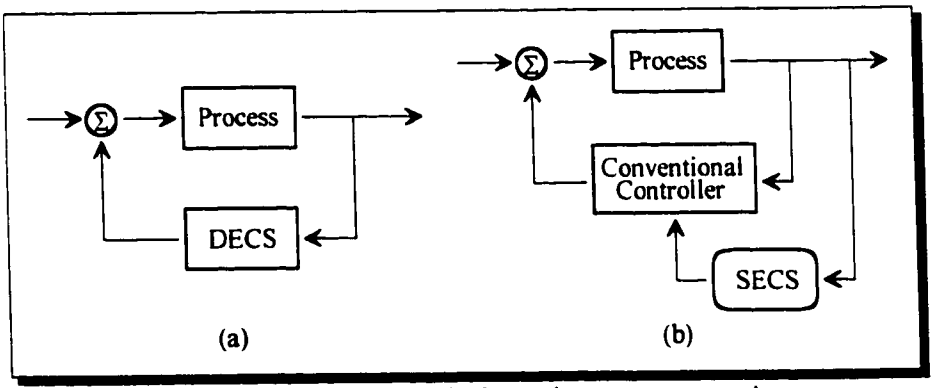


Figure 3-8. (a) Direct expert control system. (b) Supervisory expert control system. [Dri93], p. 19-20.

The following sections describe the two types of knowledge-based systems, the expert system and fuzzy logic system. As will be discussed in subsequent chapters, both of these forms of KBS have been employed in the particle accelerator technology insertion exercise for different aspects of the overall control task.

3.2.1 Expert Systems

An expert system has been defined as "a computer program that relies on *knowledge* and *reasoning* to perform a difficult task usually undertaken only by a human expert", ([Par88]). In essence, expert systems "enable computers to assist people in analyzing and

solving complex problems that can often be stated only in verbal terms", ([Har88], p. 3). Some researchers use the term *intelligent knowledge-based system* (IKBS) instead of expert system. Bader and Edwards ([Bad91], p. 383) prefer the more general term *knowledge-based system* because "it does not use the imprecise term *intelligence* and does not imply that KBSs possess human-intelligent capabilities". For the sake of brevity, the term *expert system* will be used in the sequel.

Expert systems have been studied since the early 1970s. Two of the first expert systems were MYCIN and XCON. The former was developed at Stanford University as a proof-of-concept system used by physicians for diagnosis of bacterial infections, ([Sho76]). The latter was developed at Carnegie-Mellon University and used at Digital Equipment Corporation for configuration of newly purchased computer systems, ([Bac84]). Since their initial debut, expert systems have diversified widely, proving successful at many types of tasks, including: diagnosis (problem analysis), consultation (advising), monitoring (on-line, off-line), configuration, designing, planning, scheduling, management, ([Har88]).

In essence, an expert system consists of two parts (as shown in Figure 3-9): a *knowledge base* and an *inference engine*. The knowledge base (sometimes called a *rule base*) is used to store 'information', typically of two types: ① *Facts* are stored pieces of information that are to be processed; ② *Rules* are used to combine *antecedent* facts to produce other, *conclusion* facts.

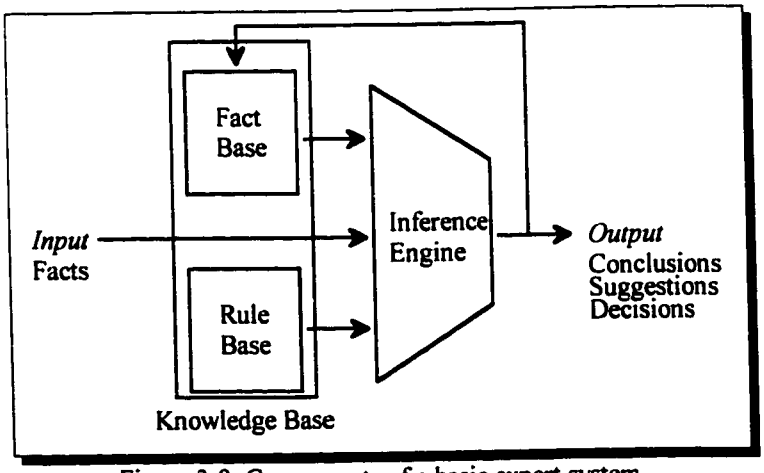


Figure 3-9. Components of a basic expert system.

Rules have a form resembling logical implications such as IF A THEN B, and associate consequent *actions* B with antecedent *conditions* A. The *inference engine* is used to process the rules in a prescribed manner to make decisions: Rules combine input facts with stored facts to produce output facts and new stored facts. For example, rules in an expert system used for control would combine input sensor data with stored state variables to produce output control commands and new state variables. Thus, "an expert system is an application program that includes both information about a particular problem and information about how to manipulate that information", ([Har88], p. 31).

The general structure depicted in Figure 3-9 benefits from three important principles: *knowledge representation*, *inferencing with heuristic search*, and *separation of knowledge from inference and control*, ([Har88], p. 7). These principles are elaborated upon in the following.

Knowledge Representation. The *knowledge* in a KBS is "a body of information about a particular topic that is *organized* to be useful", ([Har88], p. 7). This knowledge is usually stored symbolically, using words or symbols, rather than numerically. Knowledge-based systems attempt to mimic human decision-making processes in ways that make knowledge-based system development straightforward and transparent. It is the job of the *knowledge engineer* to collect, organize, structure and codify the KBS's 'knowledge'. Some of the basic concepts of knowledge engineering are summarized in Section 3.2.1.2.

Inferencing with Heuristic Search. *Inference* is the process of deriving new facts from established facts. In an expert system, the *inference engine* performs the inferencing, systematically deriving new facts from existing facts. Two forms of inferencing are typically used: *Forward chaining* combines input antecedents to determine output consequents. The rules are evaluated in a 'forward' manner. Forward chaining can be regarded as the *data-driven generation* of conclusions from evidence, and in this sense forward chaining can be thought of as asking the question 'what if?'. In contrast, *backward chaining* is the reverse (or *goal-driven*) process, in which input consequents are

used to derive output antecedents. Backward chaining forms hypotheses based on observations, and thus asks the question 'why?'.

An expert system's rule base is a multi-node acyclic graph, and inferencing, whether forward or backward, reduces to path enumeration. Therefore, complexity (time to perform inferencing) increases non-linearly with the number of rules. This means that inferencing can become combinatorially prohibitive for large rule sets. The remedy to this problem is to employ search heuristics (rules-of-thumb) or approximate reasoning strategies (*metarules*) for reducing the search space. Since both heuristics and metarules are not required to be exact in all cases, their use implies that the expert system will not always generate the *correct* result, but has a high certainity of producing a *good* result *most* of the time.

Separation of Knowledge from Inference and Control. In conventional computer programs, the information (program variables) is intermixed with the control engine (program instructions) so that the two are inseparable. Consequently, the information is not readily interpretable by anyone other than a computer programmer. Within the domain of expert systems, great effort has been expended to separate knowledge and information from the inference and control mechanisms, so that the knowledge engineer need not be concerned with the details of programming. Typically, an expert system starts out as an existing, proven inference engine combined with an empty knowledge base to which the knowledge engineer is able to add knowledge in a liguistic or symbolic manner without needing to understand how the expert system as a whole is programmed. This concept effectively frees expert system developers from having to be computer programmers, enabling nonprogramming domain experts to become directly involved in expert system development. It has been claimed that "the separation of knowledge from inference and control is probably the most important concept to come out of AI research", ([Har88], p. 8).

The basic expert system structure shown in Figure 3-9 does not capture the true structural extent of most expert systems. A more detailed architecture is captured in Figure 3-10, showing some of the important ancillary components of a conventional

expert system. The grey rectangle in Figure 3-10 outlines the basic components shown in Figure 3-9. There are four ancillary components added to Figure 3-10: a knowledge acquisition subsystem, an explanation subsystem, a user interface, and a collection of special interfaces. The knowledge acquisition subsystem is used initially to build the knowledge base and thereafter to maintain, update and expand the knowledge base. In some expert systems, this subsystem is automated, but in others it must be performed by humans. The explanation subsystem is employed by both the knowledge engineer and end-user to explain how the expert system applies its knowledge base to make decisions. Finally, the user interface is necessary for enabling the expert system to interact with the human user, and the collection of special interfaces are charged with 'connecting' the expert system to 'real world' devices such as sensors and actuators. Any or all of these ancillary components may be present (or absent) in a given expert system.

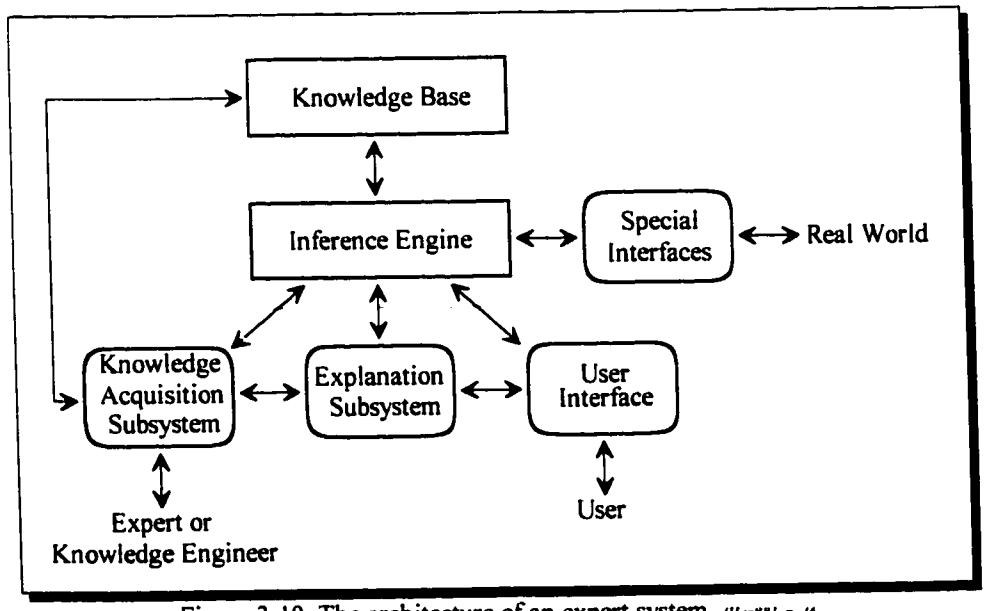


Figure 3-10. The architecture of an expert system. *[Hur88], p. 46.*

3.2.1.1 Expert System Development

A knowledge base is constructed, modified and used within a 'user community' (Figure 3-11), which is comprised of *experts*, *knowledge engineers* and *users*, ([Gai91], pp. 211-213). The experts are sources of the knowledge that is incorporated into the expert system's knowledge base. Experts are involved in acquistion, display and explanation of the knowledge base. Since experts are not necessarily knowledgeable of

computer systems and programming, the knowledge engineers cooperate with the experts to codify and transfer expertise to the knowledge base. Knowledge engineers participate in display, editing and validation of the knowledge base. Finally, the users (sometimes called the *end-users*) are the individuals with which the system is ultimately built to interact. Users take part in validation, explanation and application of the knowledge base.

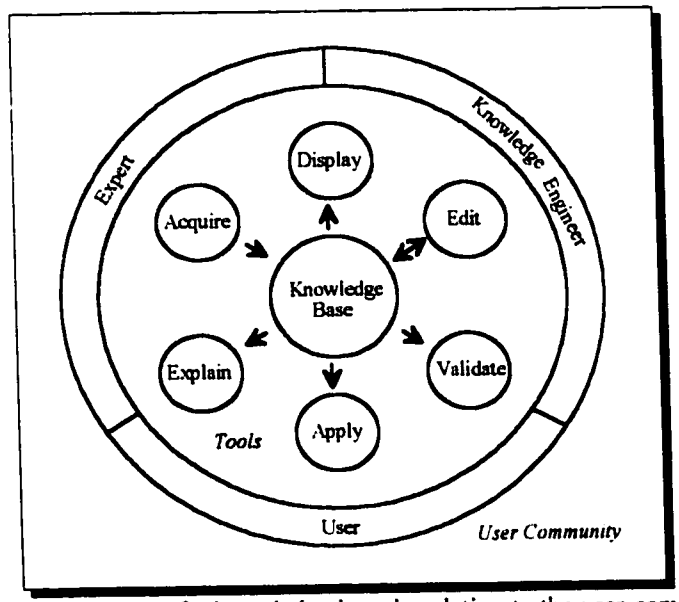


Figure 3-11. Components of a knowledge base in relation to the user community. *Adapted from [Giar91], p. 212*

There are two aspects to development of an expert system. The first involves programming the mechanisms of the expert system, that is the inference and knowledge base manipulation mechanisms which are collectively called the *shell*. The other aspect is concerned with the 'information programming', the knowledge engineering.

As illustrated in Figure 3-12, these two paths can be performed consecutively or concurrently. In Figure 3-12a, development of the expert system's mechanisms is performed prior to knowledge engineering, with perhaps a large timespan separating the two phases, and possibly different groups of people involved in each phase. This would be the case, for example, when using a commercial expert system shell developed elsewhere. There is a large number of commerical and public-domain expert system shells available, with many different characteristics such as cost, inferencing speed, target computer platform, knowledge base capacity, etc. Such shells are usually quite powerful and straightforward to use, but typically have limited flexibility because they are generic.

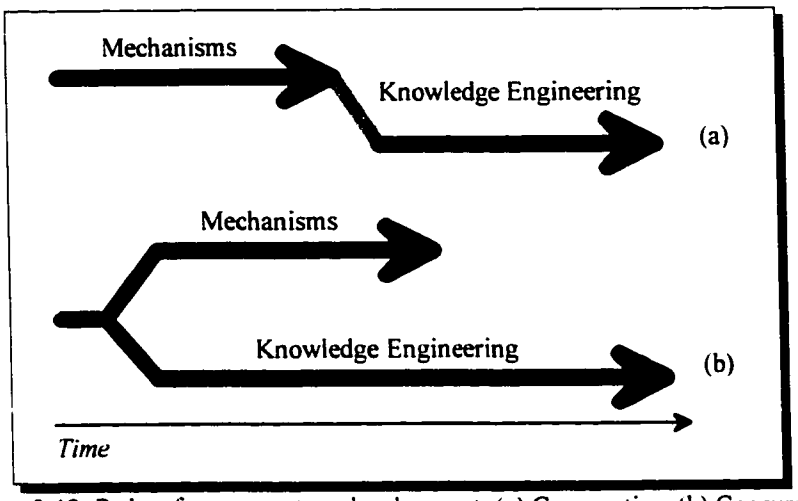


Figure 3-12. Paths of expert system development: (a) Consecutive, (b) Concurrent.

Alternatively, the concurrent approach of Figure 3-12b can be followed, in which all aspects of the expert system are developed in parallel. This would be the case with fabrication of so-called 'in-house' expert system applications. Here, the development team designs and implements the inference and knowledge manipulation mechanisms as well as performing the knowledge engineering. The programming can either be done from scratch, or by employing subroutines from existing expert system software libraries (*toolkits*). The developers also need to decide what programming language is to be used, and whether a numeric or symbolic language is warranted. Historically, expert systems have been written in AI-oriented languages such as LISP or PROLOG, but more recently systems have been implemented in C/C++, Pascal and other high-level numeric languages.

In any case, there are good reasons for choosing either one of these approaches, and the pros and cons of each should be weighed against the development team's abilities and resources, and against project constraints such as budget and timeframe. Harmon writes, "the question facing managers considering developing an expert system is whether they want to use a symbolic language or a tool written in a symbolic language, or whether they should stick with a conventional language they already know", ([Har88], p. 33). For some applications, use of a general purpose, off-the-shelf shell can yield a finished expert system rapidly and effectively, while in other cases a general-purpose shell may prove too rigid to suit the needs at hand; in these cases it may be more productive to follow the concurrent development strategy to accommodate the needed flexibility.

Once the expert system's shell is in place (whether built from scratch or 'taken off the shelf'), the next step is to imbue the expert system with its 'knowledge'. The process of *knowledge engineering* involves first acquiring knowledge and then rendering it in a form usable by the KBS. The knowledge originally exists in the form of *expertise* possessed by one or more *experts*. This expertise is sometimes available as well-defined information held in operating manuals, theory texts or procedural handbooks. But, most of the time, this expertise resides amorphously in the expert's mind, and may not easily be quantified in words or equations. For example, consider a veteran process controller who has amassed a large amount of operational expertise. This expertise is based upon training manuals and operating guides, but also arises from expertise passed on from trainer to trainee and from the expertise acquired through years of firsthand experience. In many cases, it is the latter form, *experiential knowledge*, which proves the most valuable and most difficult to express quantitatively. In the words of Harmon ([Har88], p. 31):

> "Inside the expert's head are facts, rules-of-thumb (heuristics), and various problem-solving strategies the expert uses when faced with a particular problem. To create an expert system [one] must transfer this knowledge to the computer..."

The process of transferring knowledge from expert to computer is called *knowledge acquisition*, which Harmon ([Har88], p. 266) defines as:

> "The process of locating, collecting, and refining knowledge. This may require interviews with experts, research in a library, or introspection. The person undertaking the knowledge acquisition must convert the acquired knowledge into a form that can be used by a computer program. Knowledge is derived from current sources, especially from experts."

Knowledge acquisition is performed by the *knowledge engineer*, whom Harmon (*Ibid*) defines as:

> "An individual whose specialty is assessing problems, acquiring knowledge, and building knowledge systems. Ordinarily this implies training in cognitive science, computer science and artificial intelligence. It also suggests experience in the actual development of one or more expert systems."

Knowledge engineering is a complex discipline, and training an effective and efficient knowledge engineer can take several years. The knowledge engineer must act as a liaison between the domain experts and the expert system builders. The domain experts are not likely to be skilled programmers; likewise, the expert system builders are not necessarily

experts in the domain of the expert system application. Therefore, the knowledge engineer serves as a bridge between these two groups, facilitating and organizing the flow of information between them to produce a functioning expert system.

3.2.2 Fuzzy Logic

The artificial intelligence paradigm of fuzzy logic was first developed during the 1960s by Zadeh ([Zad65], [Zad73]), who also outlined how fuzzy logic could be used in the analysis of complex systems. Fuzzy logic-based control seeks to combine the accuracy of numerical control with the flexibility and simplicity of heuristics (rules) based on natural language. Although such systems have found success in controlling a variety of processes, they have proved particularly advantageous for control of non-linear, dynamic processes which are difficult to model numerically. It has been reported that "a representation theorem, mainly due to Kosko ([Kos92]), states that any continuous, nonlinear function can be approximated as exactly as needed with a finite set of fuzzy variables, values and rules", ([Dri94], p. 3).

Driankov, Hellendoorn and Reinfrank ([Dri93], p. 2) define a fuzzy control system (FCS) as:

> "... a real-time expert system implementing part of a human operator's or process engineer's expertise which does not lend itself to being easily expressed in PID-parameters or differential equations but rather in situation/action rules."

Alternatively, fuzzy control systems can be thought of as "a heuristic and modular way for defining nonlinear, table-based control systems", ([Dri93], p. 3). Daugherity ([Dau92]) notes that fuzzy controllers have three advantages compared to conventional techniques such as PID control: ① They are cheaper to develop (with equivalent performance); ② They cover a wider range of operating conditions (i.e. are more robust); and ③ They are more readily customizable in natural language terms. Similarly, Driankov et al. ([Dri93], p. 4) claim three general advantages associated with employing fuzzy control: ① Implementation of expert knowledge for a high degree of automation; ② Robust nonlinear control; and ③ Reduction of development and maintenance time. In terms of deciding when to try using a fuzzy system for a specific problem, Driankov et al. ([Dri93], p. 8) suggest the following three steps:

1. If a similar fuzzy-based solution already exists, using a fuzzy controller is justified.
2. If there already exists a good PID-solution with good system performance, development and maintenance costs and market policy, then the PID-solution should be kept.
3. If the existing solution is unsatisfactory with respect to the criteria of (2) above, or there does not yet exist a solution, then a fuzzy system may be warranted if knowledge is available about the system or process that could be used to improve the solution but that is hard to encode in terms of conventional control such as differential equations of PID-parameters.

An early application of fuzzy logic to control ([Mam81a], [Mam81b]) involved a collection of fuzzy logic rules used to control the throttle speed and boiler heat input of a model steam engine. Subsequent research included fuzzy control of a full scale industrial process, specifically a rotary cement kiln, ([Lar81], [Umb81]). A review of recent literature on control systems indicates a sudden increase in the popularity of using fuzzy logic for control problems. For example, the 1992 IEEE Conference on Fuzzy Systems ([IEE92]) was host to a variety of researchers experimenting with fuzzy control systems. Areas of research described in [IEE92] include such problems as: control of a semiconductor manufacturing process, control of a gas-fired water heater, and motion control of a robot.

The diversity of current research in fuzzy control systems indicates that researchers are actively seeking control problems that are amenable to efficient, reliable and cost-effective hardware realizations of fuzzy logic-based controllers.

Fuzzy logic is based on fuzzy set theory. It has been shown that conventional (Boolean) set theory is a special case of fuzzy set theory, ([Kos92]). Whereas conventional set theory stipulates that an object x either is or is not a member of a set, fuzzy set theory defines a *degree of membership* for an object x in a fuzzy set. The characteristic function χ_A for a conventional set A and object x is expressed as $\chi_A(x) \in \{0,1\}$, implying that x can only be either a member or non-member of A. The characteristic (membership) function μ_A for a fuzzy set is $\mu_A(x) \in [0,1]$, meaning that x can be a member of A to any degree in the continuous range 0 to 1. Thus, any object in the universe of discourse is a member of a fuzzy set to some degree.

A fuzzy system (Figure 3-13) can be thought of as a combination of three 'mappings': ① A 'fuzzification' mapping of numeric inputs to linguistic terms (fuzzy input variables); ② An inferential mapping of linguistic antecedents ('if' clauses) of fuzzy input variables to linguistic consequents ('then' clauses) of fuzzy output variables; and ③ A 'defuzzification' mapping from linguistic terms (fuzzy output variables) to numeric (*crisp*) outputs. This arrangement provides a means of gaining numerical precision from descriptive, linguistic heuristics.

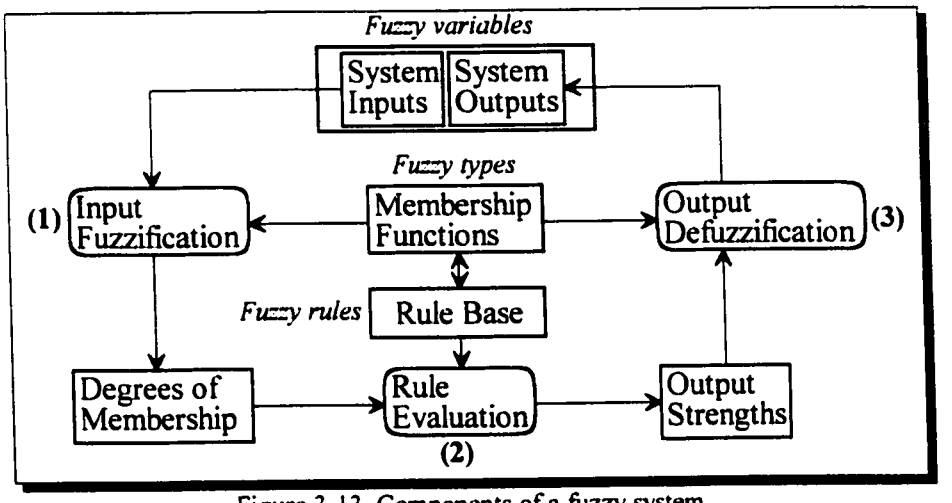


Figure 3-13. Components of a fuzzy system.

A fuzzy logic system contains a collection of *fuzzy variables*, each variable belonging to a specific *fuzzy type*. A fuzzy type is defined by a set of *membership functions* (characteristic functions) which span the range of possible values for the fuzzy type (see Figure 3-14). The set of membership functions serves to map a 'real world' (*crisp*) value into a fuzzy value (set of membership function fit values). For example ([Vio93]), the fuzzy type Temperature (Figure 3-14) may have fuzzy values *Cold*, *Cool*, *Warm* and *Hot*; each value is described by a membership function. A crisp temperature value of 63 °F, when fuzzified, is interpreted as *Cold* to degree 0, *Cool* to degree .8, *Warm* to degree .2, and *Hot* to degree 0. The fuzzy value v for a crisp value x can be written as a fit vector $v = [0,.2,.8,0]$, where $v_i = \mu_i(x)$.

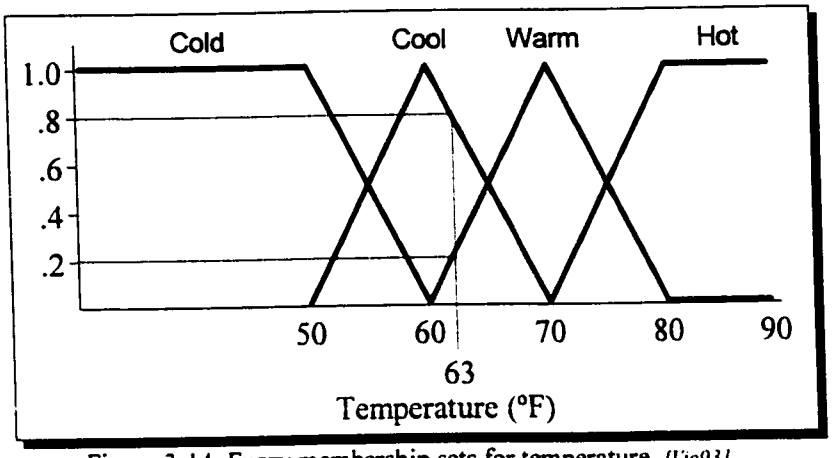


Figure 3-14. Fuzzy membership sets for temperature. *[Vio93]*

Fuzzy variables are combined in fuzzy rules to perform fuzzy inferencing. For example, a rule for controlling a household furnace might be of the form:

if RoomTemp is Cold and ThermostatSetting is High
then Furnace is OnHigh

The rule is inferenced as follows: Each antecedent is evaluated, using membership functions, to yield a 'fit value' in [0,1]. The fit values for the antecedents are then combined to calculate the 'fit value' for the consequent. If the rule uses an 'and' operator, the minimum antecedent fit value becomes the fit value of the consequent. For an 'or' rule, the maximum antecedent fit value is used. (Other methods of combining antecedents exist, [Kos92].) For example, given that *RoomTemp is Cold* has fit value .56, and *ThermostatSetting is High* has fit value .32, the fit value for *Furnace is OnHigh* would be .32.

Generally, a fuzzy system will have multiple rules which may have conflicting consequents (e.g. *Furnace is OnHigh* and *Furnace is OnMedium*). A common conflict resolution strategy uses an elementwise weighted sum of the rules' consequent fit vectors as the overall fuzzy value of the output variable, ([Kos92]).

The overall fit vector is a fuzzy value which must then be converted to a crisp output value using 'defuzzification'. Among a number of documented defuzzification methods, *centroid defuzzification* is commonly used. This method determines the area-wise centroid of the fuzzy fit vector as the crisp value, ([Kos92]). An example is shown in Figure 3-15, where *Furnace is OnHigh* has a value of .25, *Furnace is OnMedium* has a value .5, and the resultant furnace setting is calculated as the centroid of the shaded areas (59).

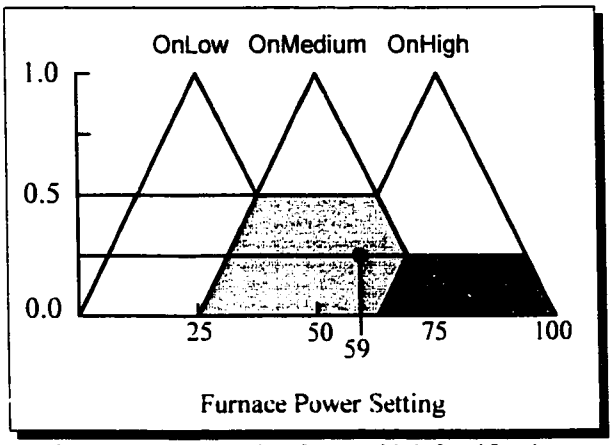


Figure 3-15. Example of centroid defuzzification.

Thus, the fuzzy system is 'executed' in a three-step process: ① Input variables are fuzzified; ② Fuzzy rules are evaluated; ③ Output variables are defuzzified. In a control application, the input variables hold sensor data or state information, and the output variables hold control actions. The fuzzy system is repeatedly 'executed' to yield continuous control.

◆ ◆ ◆

This section has served to furnish background information on two aspects of artificial intelligence suitable for real-time control applications, namely expert and fuzzy systems. The following section investigates aspects of human factors that are useful in forging an effective interface between software systems and human users.

3.3 Human Factors

During the second world war, so called 'pilot error' caused accidents in several thousand airplanes of the U.S. Air Force, directly as a result of design flaws. During a two year period, for example, some 2000 planes were involved in similar accidents because their flap and landing gear levers were "identical in appearance and located side by side", ([Sma94]). Airmen were confused by the location and appearance of the two levers, resulting in pilot error and perhaps even catastrophe and death. It became evident to aircraft engineers, and to systems designers in general, that any machine or system developed to interact intimately with humans would require careful consideration of 'human factors', those aspects of systems design specifically related to human characteristics such as physical ability, cognition, perception and emotion. Thus, the study

of the field of *Human Factors* gained prominence, seeking to study and solve the myriad problems manifest in the interaction of machines and humans.

Human factors (or *Ergonomics*[19]) is defined by Shackel and Richardson ([Sha91a], p. 3) as "the study of the relation between [humans] and [their] occupation, equipment and environment, and particularly the application of anatomical, physiological and psychological knowledge to the problems arising therefrom". Humans and machines form what Shackel and Richardson ([Sha91a], p. 6) call "a 'socio-technical system' in which [humans and machines] must be complementary components working to a common goal".

Ergonomics is concerned with the optimization of human-machine systems operation with respect to operator efficiency, safety, comfort and satisfaction, as gauged by the performance and opinion of actual human users. As illustrated in Figure 3-16, a human-machine system may be conceptualized as a composition of four components, the *user*, *task*, *tool* and *environment*. Within this conceptual framework, optimization of operation can be accomplished in two different forms. In 'fitting the system to the user', aspects of the task, tool and environment are improved, such as safety, usability and reliability. Additionally, in 'fitting the user to the system', personnel are improved through selection, training and adaptation to environmental and working conditions, ([Sha91a], p. 4).

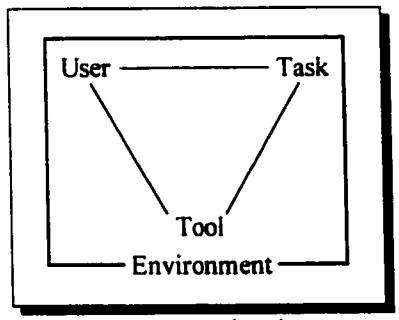


Figure 3-16. The four principal components in a human-machine system. *[Sha91b], p. 23*

Human factors researchers often lament that the value and importance of ergonomics is frequently downplayed, side-stepped and underestimated during system development, its principles being improperly considered by system designers as nothing more than common sense, ([Mil94]). On the contrary, ergonomics has been claimed by Shackel and

[19] The terms *Ergonomics* and *Human Factors* are synonymous in the literature, the former used predominatly in Great Britain and Europe, and the latter used in North America. The terms will be used interchangeably in this text.

Richardson ([Sha91a], pp. 4-5) to be essential to modern industry for several important reasons, including: ① The increasing complexity and sophistication of modern industrial technology places increasing demands on human operators, but also inhibits proper learning and utilization of human factors concepts by systems designers because they lack the proper training or are overburdened with technical problems; ② The increasing complexity of systems also creates a separation in time and space between designer and user, hampering the valuable feedback that facilitates design improvement; in this case, an ergonomics specialist (*ergonomist*) can remedy the problem by acting as an intermediary and promoting a "preventive and predictive feedback channel" between users and designers.

The field of human factors is mutlidisciplinary, stemming from several associated areas, as shown in Figure 3-17. Within its broad-ranging scope, however, two sub-domains of human factors are directly relevant to this thesis, namely the study of the *human-computer interaction* (HCI), and a subsection of HCI involving the design of the *computer-human interface* (CHI).

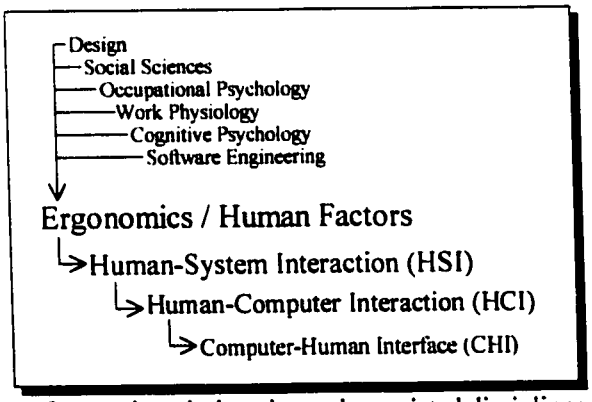


Figure 3-17. Human factors, its sub-domains and associated disciplines. *Adapted from [Sha91], p. 2*

The field of HCI involves studying how humans interact with computer systems to accomplish tasks. Shackel and Richardson ([Sha91a], p. 1) define HCI as follows: "Human-computer interaction deals with all aspects of the human use of computers, usually in the context of interactive informatics systems". HCI emphasizes *usability* of the computer system to facilitate *efficient* task performance. That is, in order to justify using a computer system to perform a given task, the computer system must be used *effectively*,

which in turn requires that the system be *usable* by humans. As Nickerson ([Nic86], p. 89) argues:

> "The challenge for human-factors people ... is to assure that ouput from the computer constitutes suitable input for the person, and, conversely, that input to the computer is something that is convenient for the human to put out."

HCI represents an extensive body of research, ranging from task analysis and ergonomic design of input/output devices to design of human-computer dialogues and the computer-human interface. An important issue in HCI relates to the input-output bandwidth of a human-computer system, what Nickerson ([Nic86], p. 106) calls the "impedance mismatch between computers and users". The computer can input and output information far faster than the user can produce or assimilate it, but the computer is not yet well capable of accepting information from the user in the way that people usually convey information to each other. Thus, a primary goal of HCI is to develop ways in which humans and computers can exchange information with speed, accuracy and convenience for both the user and the computer. This effort involves exploring physical methods of information exchange (computer input/output hardware) and cognitive methods (forms and content of information). These physical and cognitive aspects of HCI are embodied in the computer-human interface, the 'window' through which the human and computer interact. The following section elaborates on some of the issues of concern in the design, implementation and user acceptance of the computer-human interface.

3.3.1 The Computer-Human Interface

The computer-human interface (CHI) is the locus of interaction between the computer system and the human, frequently referred to as the *user interface*. In defining the term *interface*, Nickerson ([Nic86], p. 89) supplies the *Webster's New Collegiate* dictionary definition: "The place at which independent systems meet and act on or communicate with each other". The computer-human interface, therefore, consists of such things as, for example, the computer's video display, keyboard, mouse and printer, and the human's eyes, ears and hands. Nickerson ([Nic86], pp. 89-90) refers to two aspects of the computer-human interface. The *physical* interface involves the mechanisms of the human-computer interaction, the computer's input/output devices and the human's

sensory-motor system. The *cognitive* interface is concerned with the form and content of the information transferred between human and computer. Licklider ([Lic65]) prefers the term *intermedium* to *interface* because he feels that the latter term is insufficient to capture the intricacies of the human-computer interaction. Nevertheless, the term *interface* has gained widespread acceptance and will be used in the sequel despite its descriptive shortcomings.

Ziegler ([Zie91], p. 186) also describes two different 'views' of the CHI. The *psychological view* considers the CHI to be the "set of all physical, perceptual or conceptual elements structuring and mediating the user's interaction with the system". The psychological view of the CHI is based upon *cognitive models* of human thinking and problem solving, (cf. [Car83], [Bar87]). Cognitive models are used to "provide a basis for representing and understanding the cognitive consequences of particular [CHI] designs", ([Bar91], p. 169). For example, a cognitive model presented by Norman ([Nor86]) is shown in Figure 3-18.

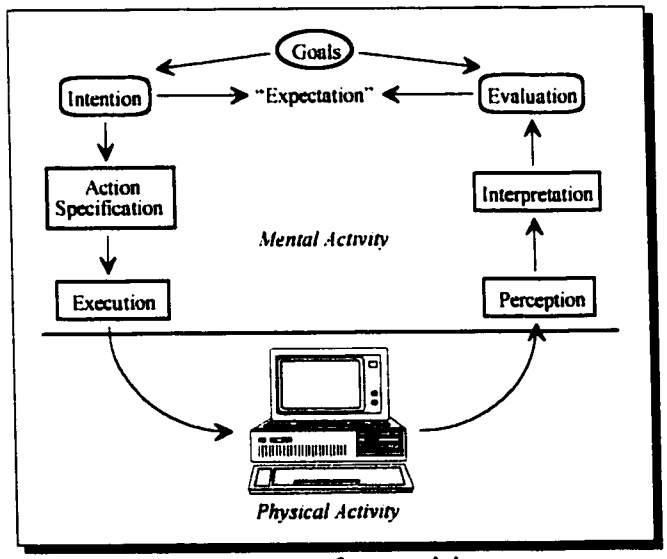


Figure 3-18. Seven stages of user activity. *Adapted from [Nor86]*

Norman's model consists of seven approximate stages of mental activity that occur during physical interaction with, for example, a computer system. Activity begins when the user forms a goal and an intention to act, causing specification of an action sequence followed by execution of the action sequence (e.g. typing on a keyboard or moving a

mouse). The effect of the action sequence is subsequently perceived, interpreted and evaluated.

The *computer science view* of the CHI is dramatically different from the psychological view in that it deals with *how* the CHI is designed and implemented. Ziegler ([Zie91], p. 186) characterizes the computer science view as:

> "all the components of a system which either map the physical user input ... onto internal data and functions of a so-called application component, or ... map changes in the internal representation of the application onto output operations perceivable by the user."

Whereas the psychological view is usually taken by the evaluator of the system, the computer-science view is usually taken by the system developer. In essence, the psychological view concerns how the *users* of the system view the CHI, while the computer-science view concerns how the *implementors* of the system design and build the CHI. Ziegler ([Zie91], p. 186) writes that "an integration of these two views in a common representation would be desirable but so far has not been achieved".

The computer-human interface has evolved greatly since the first days of computing. The earliest computers, such as ENIAC ([Dor77]), offered the user only banks of toggle switches and rows of indicator lights as a primitive user interface. Gradually, as computer systems became more advanced, new and valuable forms of user interface were developed, eventually producing the keyboard, mouse, high-resolution video display, laser printer and so-called 'windows' graphical user interfaces that are commonplace today. In the past, computer systems were scarce, highly specialized machines used by a few highly trained individuals; now, many computer systems are so affordable and usable that they are becoming integrated into many aspects of everyday life.

As computers have shifted from rare and specialized to common and generalized, the requirements of their user interfaces have changed. At one time it was more cost effective to 'fit the user to the computer', requiring the user to conform to the machine without much regard for such humanocentric concepts as comfort, usability or efficiency. In contrast, many contemporary computer systems can *only* be cost-effective if their user interfaces are 'fit to the user' by applying an ergonomically-oriented process of designing

the computer system to possess such characteristics as user comfort, ease of learning and ease of use.

Thus, an increasingly greater emphasis is placed on the user interface, and consequently an increasingly larger proportion of computer system design and development needs to be invested in the user interface, and indeed system-wide ergonomics in a general sense.

With regard to system development, Nickerson ([Nic86], p. 221) writes that "it is a popular misconception that products of engineering come into existence by means of a sequence of clearly demarcated phases, beginning with design". Instead, he suggests that it is more typical that the design phase occurs in parallel and in cycle with the implementation phase: design leads to implementation and testing, which may lead to re-design, re-implementation and re-testing. Nickerson ([Nic86], p. 231) summarizes this concept as follows:

> "System development is often thought of as requiring two qualitatively different tasks: design and implementation. When the functions the system is to serve are well structured and clearly understood, these tasks can be performed seriatim, and it may make sense to insist that the design be finished before the implementation begins. As the complexity of the system increases, however, it is more and more often the case that what is required of it cannot be entirely specified in advance but is best determined by a process in which the developer and user together explore various possibilities in an effort to converge on something useful."

Such cyclic development implies employing users to evaluate the system. Since the system is ultimately intended for the users, their involvement in its development is crucial, as are their needs of the system. Some researchers (cited in [Nic86], p. 222) claim that system design should, above all, begin with the users, but others ([Wri82]) report that it may be better to force the user to fit the system than to fit the system to the user.

Nevertheless, users will have some *a priori* perception of their *needs*, the properties of the system that they would deem desirable or necessary. Yet, it is not necessarily true that users' needs will remain static over time, ([Cav67], [She82]), and users ignorant of system design and development cannot be expected to say they need features or capabilities that they cannot predict as being necessary. Sheil ([She83]) has argued that "any attempt to obtain an exact specification from the client is bound to fail because ... the client does not

know and cannot anticipate exactly what is required". Moreover, as users gain experience with a system, they may suggest additional features that are needed or might prove useful, features they were unable to consider earlier due to ignorance. At the same time, users may eventually come to rely on system features that they otherwise might not have thought of as necessities. Obviously, both the users' needs and their perception of those needs change over time.

The problem, therefore, is that designers and users may not be standing on level ground. Whereas designers know *what* and *how* system features can be implemented, they may not know what is *needed* by the users. Alternatively, users have some idea of what they need, but do not usually know how such things are implemented, or the limits of what can be implemented. Consequently, it seems prudent to include prospective users on the design team, but it is cautioned ([Nic86], p. 224, [Nic77]) that "users who get involved in system development may not be representative of the average prospective user, simply by virtue of their involvement in the development process". This conundrum is difficult to resolve, and researchers are still evaluating methods of involving users in design and development without corrupting their capability to assess system usability.

Gould and Lewis ([Gou83], p. 50) have advocated four basic principles that should be followed in CHI design to produce systems that are useful, easy to learn, and pleasant and easy to use:

> · "Designers must understand who the users will be.
> · A panel of expected users should work closely with the design team during the early formulation stages.
> · Early in the development process, intended users should actually use simulations and prototypes to carry out real work, and their performance and reactions should be measured.
> · When problems are found in user testing, as they will be, they must be fixed. This means design must be iterative: there must be a cycle of design, test and measure, and redesign, repeated as often as necessary."

The above list of principles may seem obvious common sense, but Gould and Lewis report that system developers do not apply such principles liberally. In six surveys of over 400 system designers, "only 2 per cent of the respondents mentioned all four principles, 35 per cent mentioned only one, and 26 per cent mentioned none of them", ([Nic86], p. 227).

Needless to say, the 'mentioning' of a design principle says nothing of whether the principle was actually employed.

The problem with applying the principles of Gould and Lewis — that is, involving users in design and testing — is that, in some applications, there may not exist a large enough community of users from which to choose. Moreover, such user involvement must obviously take place on or near the location of the user community. Given that a user pool is readily available, the problem remains that different users do not necessarily exhibit equal appreciation of, or benefit from, the same user interface features. That is, impressions of system acceptability vary amongst users, making it difficult to assess user acceptance of a system in a general sense.

User acceptance is confounded by the fact reported by Walther and O'Neil ([Wal74]) that people with negative attitudes towards computers learn how to use them less efficiently than people with positive attitudes. Likewise, Zoltan and Chapanis ([Zol82]) report that many people's attitudes may present obstacles hindering them from becoming effective users. Another problem involved in system acceptance is the threat to job security, imagined or real, that some people may feel the computer system generates. For example, skill-oriented seniority can be threatened or devalued by introduction of computerized skill that can be used by less-skilled operators, ([Nic86], pp. 240-241). It is evident, therefore, that in addition to system-specific qualities, user acceptance or rejection of a system may also be influenced by positive or negative attitude towards computer systems in general.

The user community represents a continuum of user skill levels, ranging from novices to experts. Novice system users need a high level of instruction and guidance while learning to use the system. They prefer simplicity, but as they gain experience, they should eventually become experts. Experts require no instruction, and possibly are able to supply instruction to new novices. As novices evolve into experts, they demand the addition of greater system functionality. An increase in functionality implies an increase in complexity, which in turn implies increased difficulty for novice users. Thus, systems that are used by such a continuum of skilled users suffer from conflict between simplicity and functionality,

and it is difficult to provide both effectively. One approach is to partition the skill level continuum into several groups and provide different forms of system behaviour for different skill levels. This method relies on proper partitioning of the skill continuum, and may oversimplify the problem to the detriment of all skill levels, ([Nic86], p. 255).

Birnbaum ([Bir82]) suggests a remedy to such varying acceptability by users with differing attitudes and skill levels. He suggests that the user interface might benefit from individualized design, in which individual users are able to tailor and adapt the user-interface to suit their personal preferences. This feature would alleviate some of the problems arising from differing individual user preferences, but complicates system implementation. Similarly, Wixon, Whiteside, Good and Jones ([Wix83]) promote two principles for user interface design: ① "An interface should be built based on the behavior of actual users", and ② "An interface should be evolved iteratively based on continued testing".

Regardless of whether users are involved in system development, the CHI implementors need to decide how the user interface is to behave and what features and capabilities it should provide. There are five primary styles of interaction for conveying information between computer and user, ([Shn91], p. 326-335). These styles, and their advantages and disadvantages, are summarized below:

- *Menu selection*: The user selects items from a list displayed by the computer (e.g. Figure 3-19a). Menu selection can shorten training time, reduce number of keystrokes, promote structured decision-making, enable use of dialog management tools, and offer easy support for error handling. Menu selection may, however, slow frequent users, require screen space and a fast display rate, and tend to become complicated if many menus are needed.
- *Form fill-in*: The user supplies information by 'filling in' a set of labelled fields comprising a 'form' (e.g. Figure 3-19b). This style simplifies data entry, requires modest training, provides convenient assistance and context for activity, permits use of form management tools, and enables built-in automatic data validation and error handling mechanisms. In contrast, forms consume screen space and require typing skills.
- *Command language*: The user specifies commands using a well-structured syntactical formalism, permitting flexibility and speed, and encouraging user initiative. A command language is appealing to experienced users who are able to formulate command sequences mentally with little reliance on the visual

feedback associated with menus or forms. To their disadvantage, command languages require substantial training and memorization, may be difficult to remember, and can suffer from poor error handling mechanisms.

- *Natural language*: The user communicates with the computer using the user's natural language, thereby relieving some of the burdens associated with formalized command languages. Natural language for computer interaction suffers from ambiguity necessitating clarification, and may require more typing than command languages.
- *Direct manipulation*: The user interacts with the computer system in a highly visual manner, employing a graphics screen and pointing device (such as a mouse, light pen or touch-sensitive screen). Instead of typing in information or specifying commands, the user selects and manipulates objects to perform tasks (e.g. Figure 3-19c). In this interaction style, the task can be visually represented, making it easy to learn and retain system operation, encouraging user exploration, and reducing errors. Direct manipulation, however, requires specialized hardware such as graphics screens and pointing devices, and a high level of programming.

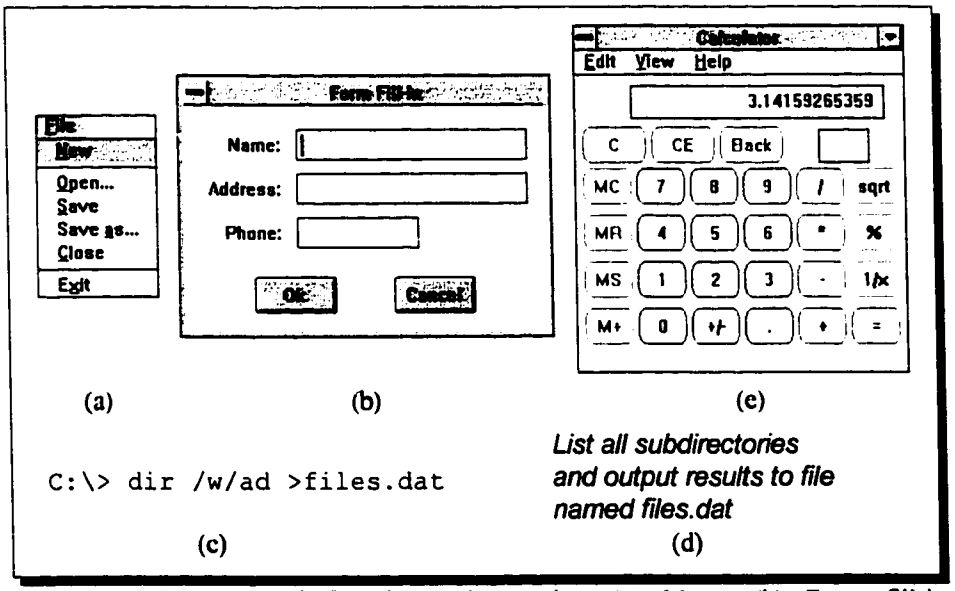


Figure 3-19. Examples of five interaction styles: (a) Menu, (b) Form fill-in, (c) Command language, (d) Natural language, (e) Direct manipulation.

It is important to choose the appropriate interaction style for a given task, and Norman ([Nor83]) posits that there are no straightforward, clear-cut solutions to the question of interface style, only trade-offs. For example, in a trade-off between menu size (number of items) and display speed, it has been found, contrary to intuition, that users prefer either large menu size *or* high display speed, but not a compromise between the two, ([Nor83]). Experimental results indicate that "trade-off functions are often concave upward

(U-shaped), suggesting that satisfaction — if it is a linear combination of the satisfaction with respect to both features — is maximized by an all-or-none solution", ([Nic86], p. 224). Moreover, the question of user satisfaction is confounded by the fact that satisfaction varies from user to user and from task to task. Although further research is needed, it appears that user satisfaction is an important issue to consider during user interface design; the problem remains as how best to measure user satisfaction and derive design principles therefrom.

Along this line, Shneiderman ([Shn91]) has developed a set of guidelines to aid system developers in selecting suitable interaction styles, which are reported as a set of three rule bases. The first rule base indicates which interaction style should be chosen based on task-related factors (Figure 3-20), and the second rule base infers interaction style based on user skill level (Figure 3-21). The third rule base determines a variety of aspects of interaction design based on user skill level (Table 3-2). Shneiderman writes that these rule bases are preliminary in nature and that further refinement may be warranted. Nevertheless, they serve to illustrate that systems developers should be able to make design and implementation decisions about the user interface depending on important factors such as user skill level and task requirements.

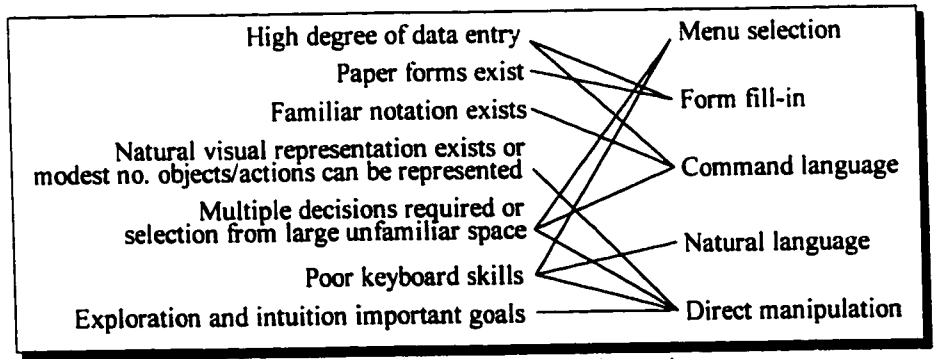


Figure 3-20. Task factors as determinants of interaction styles. *Adapted from [Shn91], p. 339.*

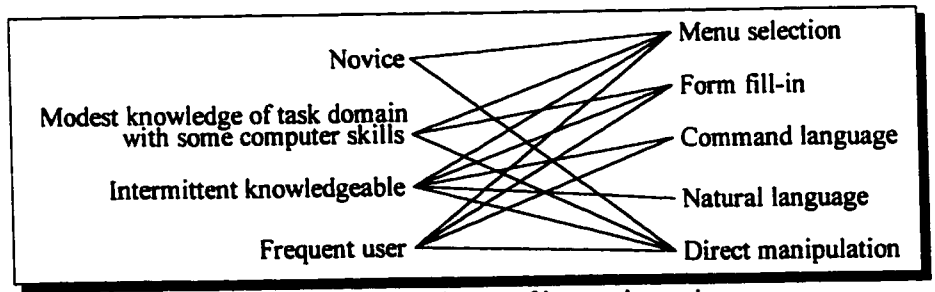


Figure 3-21. User skill factors as determinants of interaction styles. *Adapted from [Shn91], p. 340.*

	Novice	*Knowledgeable Intermittent*	*Frequent User*
Informative feedback	High density	Modest	Short, sparse or none
Pace	Slower	Moderate	Faster
Other	Introductory tutorial/demo. Limited subset of actions and functionality.	Online help. Chance to move up to more powerful actions, but protection from danger.	Online reference with elaborate search mechanisms. Abbreviations, shortcuts, user-defined macros, access to system internals.

Table 3-2. User skill levels for determining a variety of aspects of interaction design.
Adapted from [Shn91], p. 341

The term *friendliness*[20] is used by Nickerson ([Nic86], p. 149) to describe features of the computer system that "improve the quality of the interaction [with the user], reduce the probability of catastrophic mistakes, and make it easier for users to get the help from the system that they need". Nickerson ([Nic86], pp. 149-151) lists several features and capabilities that tend to promote computer system 'friendliness' as perceived by users:

- *Command confirmation*: Commands that have high-impact or widespread effect should cause the system to prompt the user for confirmation before proceeding. The user should be able to continue or abort the command, and possibly also ask for clarification or help. Examples are actions that cause irrevocable results, impose excessive time delays before completion, or cause some dangerous real-world action to occur.
- *'Undo' commands*: Commands that cause a change in the state of the system should have an 'undo' feature to reverse their effects.
- *'Your turn' signal*: The user should be given a clear indication of when the computer is busy or expecting user response.
- *'Forget it' command*: The user should be able to abort a command action, possibly undoing changes that have already been made.
- *'Enough' command*: The user should be able to halt a lengthy output operation and gain control.
- *'Help' facility*: The system should be able to respond to user requests for supplemental information on diverse system aspects such as commands or states of operation.

Although this list is not exhaustive, it is interesting to note that many of these features are already commonplace among modern computer systems, and most do not require a great amount of effort to implement. The 'undo' command, however, can be difficult to implement effectively in complex real-time or distributed systems because changes may be

[20] The term more frequently used is 'user-friendliness'.

wide-ranging or irreversible. Moreover, in order to properly implement the 'help' command on some systems, an inordinate amount of work may need to be done to provide good, comprehensive on-line documentation or advice.

Nickerson indicates that there is no formal standardization of such 'user-friendly' features. He observes that there does seem to be general consensus among system developers as to the types of capabilities that elicit system 'friendliness', but argues that system developers are the wrong kind of people to evaluate system usability because of the bias that their knowledge and experience impose. Developers tend to underestimate the difficulty that novice users will encounter. Consequently, researchers advocate that "clean separation of the user interface from the rest of the system should be an architectural objective ... to facilitate the making of improvements to the interface without requiring major revision of the underlying system", ([Nic86], p. 152).

3.3.2 System Acceptability

Ultimately, as Shackel writes, the system design must "start with the end users and be user-centred around them. Therefore, the human factors aspects become paramount", ([Sha91a], p. 8). Indeed, a human-computer system is only as successful as it is acceptable by the humans involved. *Acceptability* of a system is a balance (Figure 3-22) between the system's *cost* and its three *benefits* (sometimes called the '*x*-abilities'): utility, usability and likability, ([Sha91b], p. 22). A system has *utility* if, functionally, it does what it is required to do. The system has *usability* if users are actually able to work the system successfully. Usability, in turn, has two facets: how *easily* and *effectively* humans can use a system. Finally, the system has *likability* if the users *feel* that the system is suitable.

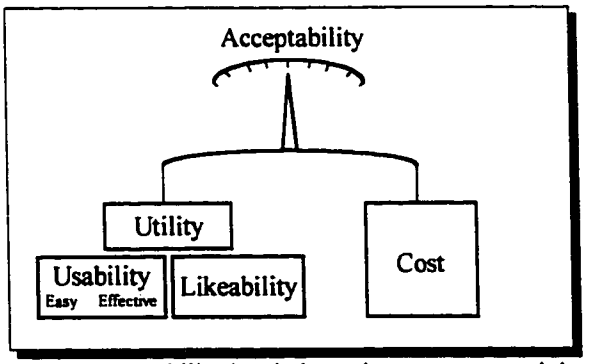


Figure 3-22. System acceptability is a balance between cost and the '*x*-abilities'.

It is obvious that utility is a simple quality to assess: the system either does or does not fulfill its functional requirements. Usability, however, is somewhat more difficult to evaluate, and involves measuring actual human performance, ranging from novices learning to use the system to experts using the system at the edge of its capability. In contrast, likability is quite difficult to gauge because it is concerned with human emotion, whether or not users *like* the system, are *willing* to use it and find it *worthwhile* to use. This exemplifies the close relation that ergonomics shares with psychology.

Shackel ([Sha81]) defines *usability* as follows:

> "The usability of a computer is measured by how easily and how effectively the computer can be used by a specific set of users, given particular kinds of support, to carry out a fixed set of tasks, in a defined set of environments."

This definition implies the use of experimental evaluation of a computer system by a group of users, and establishes a goal to which the software developers should strive. Unfortunately, as Willis and Miller ([Wil84], p. 30) point out, usability is sorely lacking from many existing computer systems:

> "Virtually every computer on the market today is advertised as a user-friendly computer, and many programs are sold as user-friendly products... Probably 50 to 80 per cent of the programs and computers claim to be user friendly but are not. Many of them are user surly or user hostile... The main problem is that many computer models and programs are not easy to use, no matter what the ads claim."

Chapanis ([Cha91], p. 359) wonders why computers, computer programs or manuals often turn out to be hard to use despite the deliberate efforts their designers have made to make them easy to use. He suggests three possible explanations: "Designers either (a) make no attempt to evaluate usability, (b) make an attempt to evaluate usability but don't do a good job of it, or (c) evaluate usability but don't think it is important enough to correct features that the evaluation shows are hard to use." He suggests that all new computer systems should be evaluated for usability because "every new computer system is, in some respects, novel. It may involve new technology or new combinations of old technology and one cannot always find principles and guidelines to cover these situations", ([Cha91], p. 360). That is, there does not exist a set of principles and guidelines that

would enable system developers to determine *a priori* the usability of a finished system. Since there is a lack of analytic methods, system usability must be evaluated empirically.

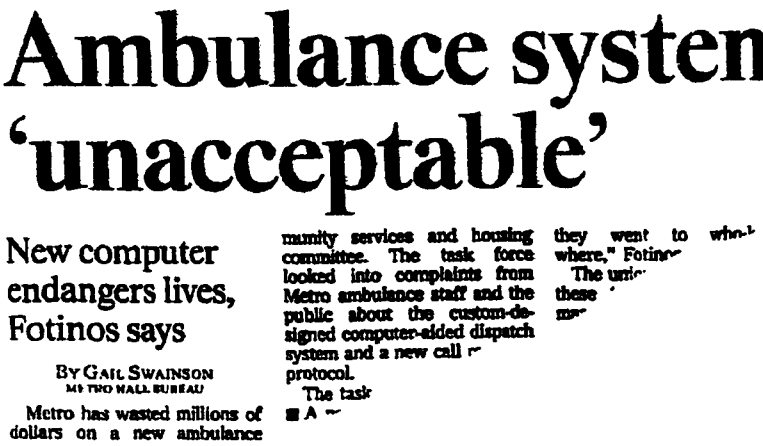
Clearly, the issue of computer system acceptability should be paramount in system design and implementation. System developers need to be aware of this necessity, and should endeavour to provide software that the end-users will find acceptable, in that the software possesses utility, likability and usability.

♦ ♦ ♦

This section has introduced the broad field of ergonomics/human factors and highlighted some of the concepts and issues that make it a valuable and necessary component of systems design. The following section presents an overview of the discipline of software engineering, whose principles can be applied to facilitate integration of aspects of real-time systems, knowledge-based systems and human factors into computer-centered technology insertion operations.

3.4 Software Engineering

A recent article (Figure 3-23) published in *The Toronto Star* ([Swa94]) reported that a new, multi-million dollar, computer-aided ambulance dispatch system has been malfunctioning to such a degree that some patients have had to wait up to 25 minutes for an ambulance to arrive. The article claims that the system has 'lost' calls, and that one of the these lost calls "involved a man who subsequently died, although it's not known whether [the lost call] contributed to his death". The system has also apparently been generating 'ghost calls' which dispatch ambulances to "locations where ambulances had responded exactly one year earlier". It remains to be seen whether this software system is actually malfunctioning as alleged, and whether its problems are due to poor software design. Nevertheless, this case illustrates the importance that *software engineering* plays in software design. The bottom line is that poor use of established software engineering principles and practices can result in software systems that have profoundly adverse and wide-sweeping effects on human lives, and society in general.



Ambulance system 'unacceptable'

New computer endangers lives, Fotinos says

BY GAIL SWAINSON
METRO HALL BUREAU

Metro has wasted millions of dollars on a new ambulance

munity services and housing committee. The task force looked into complaints from Metro ambulance staff and the public about the custom-designed computer-aided dispatch system and a new call
protocol.
The task
A

they went to
where," Fotin
The uni
these
me

Figure 3-23. Excerpt from the *Toronto Star*, [Swa94].

Software engineering is a pragmatic discipline that, in the words of Boehm ([Boe76]), involves "the practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them." Similarly, Fairley ([Fai85], p. 2) supplies the definition:

> "Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates."

Moreover, Fairley (*Ibid*) describes the roots of software engineering as follows:

> "Software engineering is a new technological discipline distinct from, but based on the foundations of, computer science, management science, economics, communication skills, and the engineering approach to problem solving."

Sage ([Sag90]) prefers the term *software systems engineering*,[21] and describes the field as encompassing three levels of activity, as shown in Figure 3-24. On a 'macro scale', software engineering activities involve *systems management* and *systems methodology and design*, dealing with teams of individuals, and large-scale goals and objectives. In contrast, the 'micro scale' level of software engineering is concerned with *software productivity methods and tools*, and deals with the work of individual people, the programmers.

[21] For the sake of brevity, the term *software engineering* will be used hereafter.

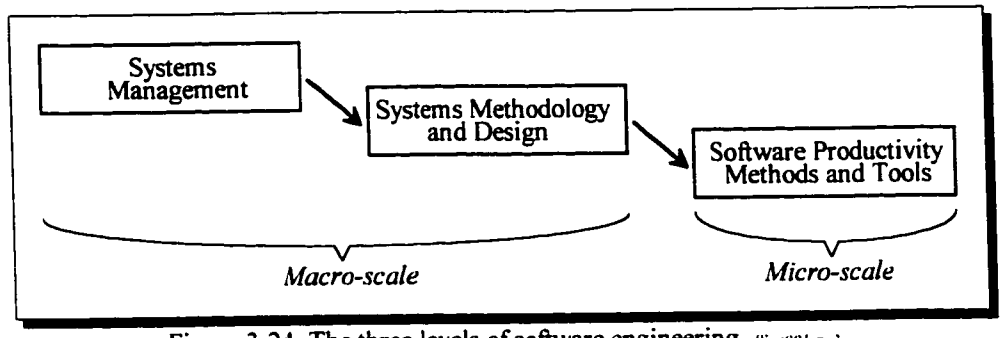


Figure 3-24. The three levels of software engineering. [Sag90], p. 2

It is important to note that software engineering is a sub-branch of the more general discipline of *systems engineering*, which involves the engineering of both hardware and software system components. Much of the material presented below on macro-scale software engineering is applicable to systems engineering in general.

As summarized by Sage ([Sag90], p. 8), studies of software productivity have indicated that many software systems are deficient in various ways, (Table 3-3), and it is generally the case that a large percentage of the overall system cost is expended on the software. Therefore, methods of improving quality and reducing the cost are valuable to software systems developers.

Software is expensive.	Software products often cannot be integrated.
Software deliveries are often quite late.	Software performance is often unreliable.
Software capability is less than promised and expected	Software is often cumbersome to use and system design for human interaction is lacking.
Software cost over runs often occur and are generally large.	Software often cannot be transitioned to a new environment or modified to meet evolving needs.
Software maintenance is complex and error-prone.	Software does not perform according too specifications.
Software documentation is inappropriate and inadequate.	System and software requirements often do not adequately capture user needs.

Table 3-3. Deficiencies of software systems. [Sag90], p. 8

Simultaneously, it is necessary for systems developers (or prospective buyers or end-users) to be able to measure the success of a software product. As indicated by Table 3-4, there are many different qualitative characteristics that a system developer (or others) might wish to use as gauges to determine whether the software meets its requirements.

Acceptable	Complete	Flexible	Precise	Timely
Accessible	Consistent	Generalizable	Reliable	Transferable
Accountable	Correct	Interoperable	Repairable	Understandable
Accurate	Documentable	Maintainable	Reusable	Usable
Adaptable	Documented	Manageable	Robust	User-friendly
Appropriate	Effective	Modifiable	Secure	Valid
Assurable	Efficient	Modular	Self-contained	Verifiable
Available	Error-tolerant	Operable	Survivable	
Clear	Expandable	Portable	Testable	

Table 3-4. Nonexhaustive list of desirable software attributes. *[Sag90], p. 16*

These attributes can be loosely organized into a 'software quality metric' hierarchy (Figure 3-25). The challenge, as pointed out by Sage ([Sag90], p. 17), is in being able to measure these software characteristics in a quantifiable manner. Some of these characteristics have been well defined and standardized ([IEE83]); a few examples are shown in Table 3-5.

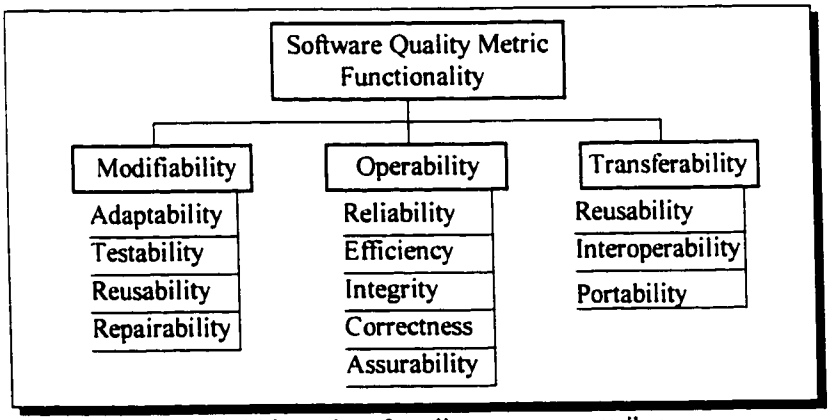


Figure 3-25. Partial hierarchy of quality assurance attributes. *[Sag90], p. 19*

Accuracy: ① A qualitative assessment of freedom from error. ② A quantitative measure of the magnitude of error, preferably expressed as a function of the relative error.

Correctness: ① The extent to which software is free from design defects and from coding defects; that is, fault-free. ② The extent to which software meets its specified requirements. ③ The extent to which software meets user expectations.

Efficiency: The extent to which software performs its intended functions with a minimum consumption of computing resources.

Portability: The ease with which software can be transferred from one computer system or environment to another.

Reliability: The ability of a program to perform a required function under stated conditions for a stated period of time.

Robustness: The extent to which software can continue to operate correctly despite the introduction of invalid inputs.

Table 3-5. Excerpts from the *IEEE Standard Glossary of Software Engineering Terminology*. *Adapted from [IEE83] cited in [Fai85], p. 35*

3.4.1 Macro-scale Software Engineering

Macro-scale software engineering is essentially the practice of applying management and design principles to software development projects involving groups of people. When computer programming was in its infancy, decades ago, software engineering was unnecessary because computer software was sufficiently uncomplicated that one or two programmers could accomplish the design and implementation singlehandedly. Yet, as increasingly larger and more complex software systems were called for, it became evident that the days of 'lone star' programmers were coming to an end. Instead, it was realized, software systems required foresight and planning, the application of sound design principles, and cohesive strategies for efficient implementation, comprehensive testing, on-time and on-cost delivery, and effective post-delivery maintenance.

The skeleton upon which macro-scale software engineering is based is the *software development lifecycle model*, a framework used for guiding and co-ordinating a software project. Fairley ([Fai85], p. 37) describes the software lifecycle model as encompassing "all the activities required to define, develop, test, deliver, operate and maintain a software product". Use of a software lifecycle model "enhances the productivity, quality and functionality of software through identification of a number of development phases that enable efficient and effective systems management of the software development process", ([Sag90], p. 48). Additionally, "a software lifecycle model that is understood and accepted by all concerned parties improves project communication and enhances project manageability, resource allocation, cost control, and product quality", ([Fai85], p. 37).

Table 3-6 summarizes how the use of a software lifecycle model affects a software development project. Although there is a diversity of software lifecycle models in the literature, it is important to note that "no single life-cycle model is appropriate for all software products", (*Ibid*). Sage ([Sag90], p. 55) notes that "each software development firm tailors the specific software development lifecycle process to meet the particular characteristics of the personnel of the firm, the needs of users for the software to be developed, and economic, legal, and time for development concerns".

· Enhances our ability to establish requirements to be satisfied by the proposed development.	· Provides cost information.
· Identifies and highlights potentially difficult problem areas.	· Lends itself to assignment of personnel.
· Permits the synthesis and evaluation of alternative solutions to difficult issues associated with each of the phases in the lifecycle.	· Lends itself to enforcement of standards.
· Enables selection of appropriate activities for each of the phases.	· Encourages the use of support tools.
	· Aids in the preparation of a quality product that is delivered on time and within budget.
	· Lends itself to management control.

Table 3-6. Benefits of using a software lifecycle model. *[Sug90], p. 50*

The original software lifecycle model is credited to Royce ([Roy70]), who introduced the 'waterfall' model shown in Figure 3-26a. Boehm ([Boe76]) subsequently refined the original model into that shown in Figure 3-26b. The waterfall model is so widely accepted that a form of it has become a standard in the U.S. Department of Defense, guiding projects that combine both hardware and software in defense-related systems (Figure 3-27).

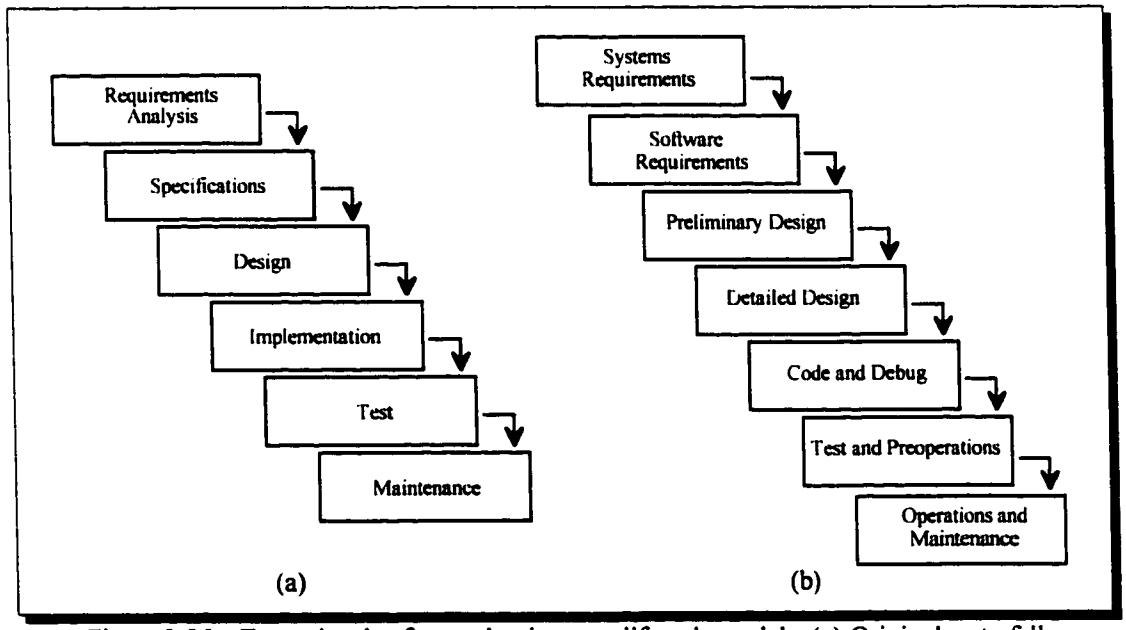


Figure 3-26. Two related software development lifecycle models. (a) Original waterfall model due to Royce, *[Roy70]*. (b) Modified waterfall model due to Boehm (after Royce), *[Boe76]*.

The basic idea of the waterfall model is that system development consists of several well-defined phases arranged in chronologically linear order, in the tradition of conventional 'top down' design. Generally, activities in one phase do not begin until all activities in the preceding phase have been completed. Ideally, in this way, it is reasoned,

problems or difficulties can be identified and corrected early in the development process. In practice, however, there is usually iteration and interaction between the phases, due to oversight or unexpected problems that become evident in phases subsequent to the phase in which they should have been taken into account. In fact, Sage ([Sag90], p. 56) indicates that iteration and interaction between phases is good, and suggests that "disadvantages associated with use of a lifecycle model include problems that occur if there is no iteration and feedback among phases." In a similar voice, Fairley ([Fai85], p. 41) writes:

> "Software development never proceeds in a smooth progression of activities as indicated in the waterfall chart. There is more overlap and interaction between phases than can be indicated in a simple two-dimensional representation."

In some cases, the iteration and interaction may become intense if software developers opt to 'fast-track' a project, usually due to tight time constraints. Fast-tracking involves deliberately accelerating or bypassing phases of the software lifecycle so that actual implementation (coding) can be performed quickly, sometimes even before the project's detailed design requirements are completely formalized. Fast-tracking is an inherently risky undertaking, and can lead to major complications if problems or errors are identified late in the development process that force the project to backtrack. Nevertheless, many developers feel that the potential benefits of fast-tracking (such as time and cost savings, and early delivery bonuses) outweigh the risks.

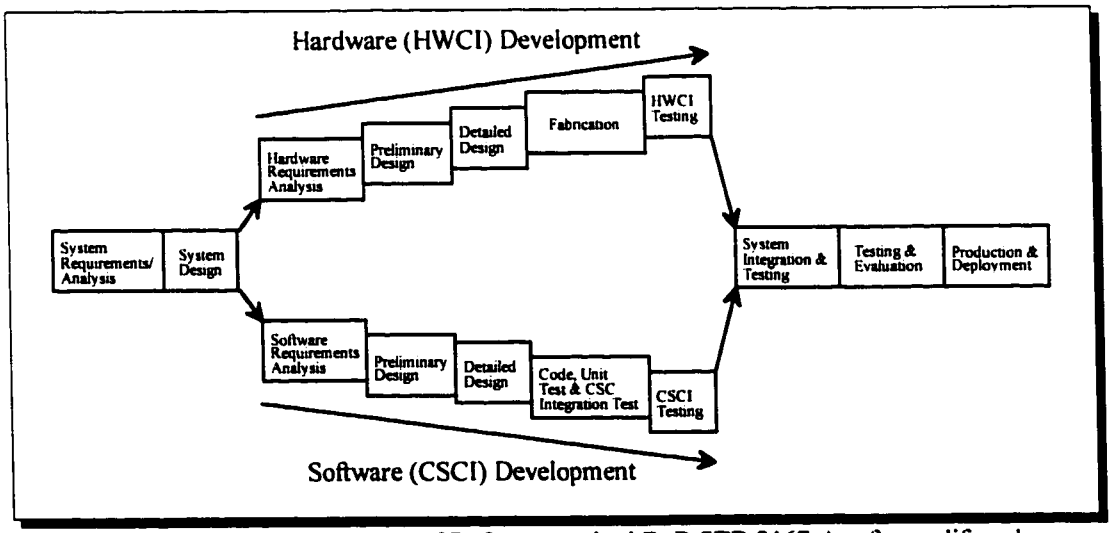


Figure 3-27. U.S. Department of Defense standard DoD STD 2167-A software lifecycle model. *Adapted from [Sag90], p. 53*

A concept related to fast-tracking is *prototyping*, in which preliminary versions of the software product are released to the prospective users for evaluation. Typically, the prototypes provide limited subsets of full product functionality, but still enable the users to acquire hands-on exposure to the developing product. The concept of a prototype is aptly expressed by Sage ([Sag90], p. 75):

> "... A prototype is viewed as an initial and possibly immature, incomplete model of the system that has been proposed for construction. The purposes of building a prototype are to enable identification of requirements specifications for a system; determine the feasibility of the overall project; explore particularly difficult tasks, such as designing a new algorithm; explore variations in requirements; or examine different requirements for impact assessment; or examine alternative strategies and approaches to solve problems."

Similarly, Jacobson ([Jac92], p. 27) writes:

> "A specific advantage of a prototype is that it can serve as a means of communication between the developer and customer. It is much easier to express a view about something that can be demonstrated and used, if only partially, than to express an opinion about a specification. A specification cannot capture the dynamics of the system in the same way as a working prototype."

Prototyping for small-scale projects is a relatively straightforward and cost-effective means of furnishing users with evaluational mock-ups, and can help to alleviate design problems early in development. Unfortunately, however, prototyping of large-scale software systems can become prohibitively expensive with relatively little pay-off to justify the expenditure.

Fast-tracking and prototyping are, therefore, two useful methods for accelerating and improving software development that can be integrated easily into the conventional lifecycle model if circumstances warrant. As mentioned previously, different software development groups will employ different lifecycle strategies and variations depending on the needs at hand.

Since its conception in the early 1970s, the basic waterfall model and its many variations have gained widespread acceptance. Software developers have discovered that the advantages of applying a waterfall model concern the organization and control of the software development project. Sage ([Sag90], p. 50) writes that "the use of the conventional (waterfall) lifecycle has demonstrated that better software will result from

the careful and systematic approaches that are called for through the careful use of a software development lifecycle". Additionally, he claims (*Ibid*, p. 56) that "the single most important methodological need associated with use of the waterfall lifecycle model is that of effective identification of user requirements". This notion ties in well with the previous section on human factors which stressed the importance of including the prospective user in the development of a software project.

The conventional waterfall model, however, is not without its problems. For example, rectification of problems and errors can be seriously impaired if no iteration or feedback occurs between the phases. At the other extreme, too much iteration and feedback can lead to expensive cost and time overruns. Additionally, the waterfall model may facilitate the 'second system syndrome' of building a system twice using iterative prototyping capability to correct problems resulting from oversight or bad planning during the initial or previous development cycles. Finally, it is possible that the rigidity of the waterfall model reduces the potential for individual creativity amongst programmers, perhaps leading to lack of inspiration and poor quality in the finished product.

Various researchers have attempted to improve upon the basic waterfall model, or replace it altogether. For example, McCracken and Jackson ([McC82]), and Gladden ([Gla82]), have suggested that the classical waterfall model should be scrapped and replaced. Along a similar line, Sage ([Sag90], p. 73) reports that:

> "It has been estimated that shortcomings in the application of the waterfall software development lifecycle model, such as, proceeding step-by-step through the design and development process without iterative feedback, have led to more than half the total life cost of a software product being expended in the 'maintenance' phase."

Sage (*Ibid*, p. 57) writes also that such attempts at revising the waterfall model usually include one or more of three general approaches. First, most newer models formally include feedback mechanisms between the phases to decrease the compartmentalization and isolation of the separate phases. Second, some models may involve the use of various computer-aided software engineering (CASE) tools designed to automate and streamline aspects of the software engineering lifecycle. Finally, some models are conceived to

introduce increased flexibility into the lifecycle to aid in development of large scale systems.

In the sequel, four different lifecycle models are described, illustrating how such models can simultaneously share common elements while seeking to emphasize and improve upon different facets of the overall software development process.

One such alternative is Yourdon's *structured software development lifecycle* model ([You82]), presented in Figure 3-28. The principal intention of this model is to facilitate the employment of structured development tools and techniques into software projects to improve organization and the development process. The structured software development lifecycle is essentially the classical waterfall model augmented with structured tools, such as data flow diagrams, data dictionaries, structure charts, and structured English. Yourdon's model offers advantages primarily in terms of the structured tools that it provides. To its detriment, however, this model suffers from the same disadvantages as the classical waterfall model.

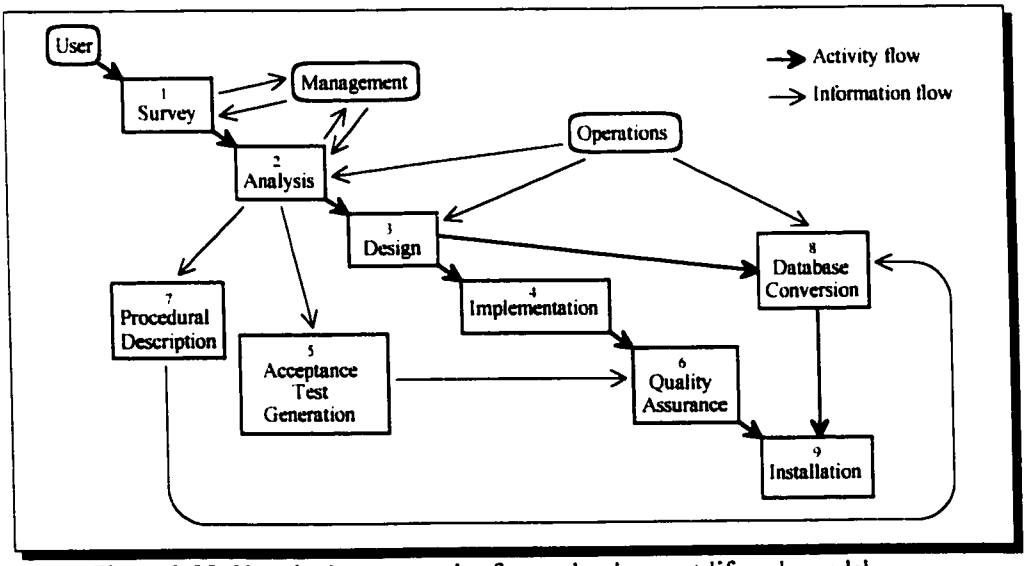


Figure 3-28. Yourdon's structured software development lifecycle model. [You82]

Another alternative model is the *spiral model* (Figure 3-29) developed by Boehm ([Boe86]). In contrast to the classical waterfall model, which is a 'specifications-driven' model, the spiral model is intended to "introduce a risk-driven approach into the development of software products", ([Sag90], p. 67). Iterative prototyping is an inherent aspect of the spiral model. Each iteration of the spiral involves a 'version' of the software

product which progresses through six phases of development (as shown in Figure 3-29). At the end of each cycle, the prototype is assessed in terms of various risk factors (such as cost, hardware requirements, and performance), and a decision is made whether to declare the project finished, continue development, or seek an alternative development path. The spiral model, therefore, combines the chronologically linear phases of the waterfall model with cyclic iteration and "many [other] modifications that have been introduced to the software development lifecycle during the past 25 years, including risk analysis", ([Sag90], p. 68).

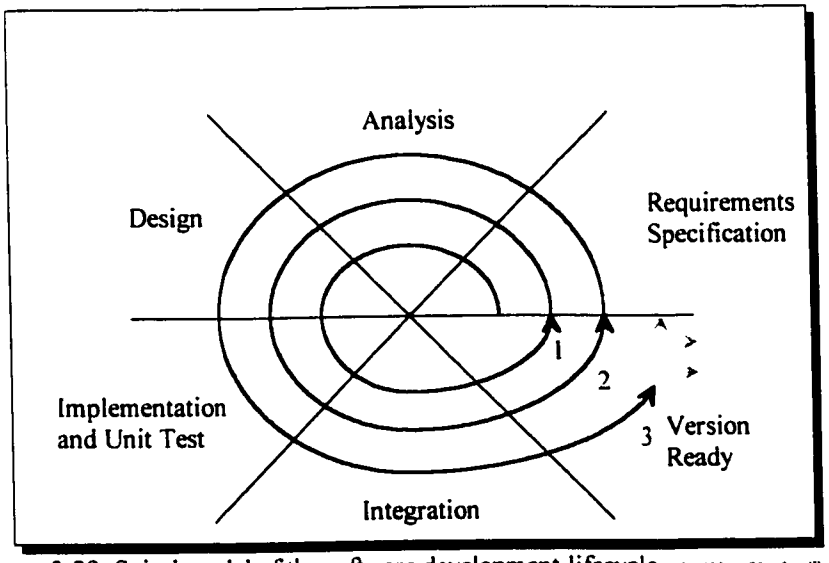


Figure 3-29. Spiral model of the software development lifecycle. *[Jac92], p. 73, after [Boe86]*

The *evolutionary* model (Figure 3-30), based on the work of McCracken and Jackson ([McC82]), and Gladden ([Gla82]), was developed to address the drawbacks of the waterfall-like class of lifecycle models. The evolutionary model is based upon prototyping as a means of providing prospective users with preliminary, hands-on software systems that the users can evaluate to provide developers better with timely, accurate requirements specifications. This method is iterative in nature: an initial, tentative set of system requirements gives rise to a first prototype, which is evaluated by the users to refine and elaborate upon the initial specifications, resulting in a second prototype, and so on until the users are satisfied with the software product. Clearly, this approach is of value when the users have vague or unclear *a priori* system requirements, or do not know what can be accomplished.

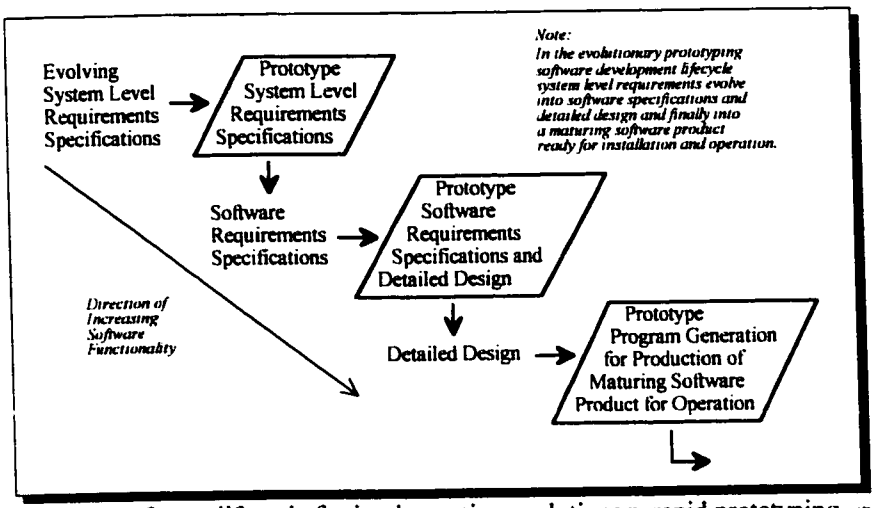


Figure 3-30. The software lifecycle for implementing evolutionary rapid prototyping. [Pag90], p. 74

The evolutionary model, like any other, has both advantages and disadvantages. To its benefit, the evolutionary model can improve system development through dividing the project into smaller, more manageable segments (modules) to make implementation and testing easier. Sage ([Sag90], p. 75) summarizes these advantages as follows:

- "Potential of bringing reality to the development process by building segments of the project and testing these for application fit.
- Working on particularly difficult issues, such as input/output (I/O) interfaces or algorithms through building and testing these portions of a system in small manageable parts.
- Better control of resource management, verification and validation.
- Increasing overall communications and productivity of the design process.
- Coupling which occurs between user group management and the development team through their direct involvement in the specification effort."

The disadvantages of the evolutionary model are common to any model based on incremental development, ([Sag90], p. 76). These include: ① Undetected errors can continue to propagate through the prototypes, making this model an expensive way to develop requirements specifications; and ② Production of prototypes may cause a significant increase in project management overhead, making resource allocation and estimation difficult.

A final software lifecycle model worthy of consideration concerns software engineering for knowledge-based systems (KBSs). Bader and Edwards ([Bad91], p. 384) claim that "despite some reported successes ([dAg87], [Pro87]), KBS technology is still thought by many in commercial computing to lack engineering credibility". They suggest

that "a software system's perceived credibility is based to a significant extent on the degree to which a [software development lifecycle] has been used in its development", and therefore, that "if KBSs are to gain widespread commercial and industrial acceptance, their development will also have to be based on a sound engineering method". Moreover, they write (*Ibid*, p. 384) that many purportedly successful knowledge-based systems are "relatively small, stand-alone systems which do not interact with other conventional systems", and these KBSs have been developed using unstructured, *ad hoc* approaches.

As a remedy to this problem, Bader and Edwards advocate a formal development model which they claim is "suitable for the construction of hybrid systems, containing both heuristic and conventional components, since it is unlikely that KBSs for commercial or industrial exploitation will solely contain heuristic elements", (*Ibid*, p. 385).

Bader and Edwards state (*Ibid*, p. 391) that "several authors characterise the prevailing AI methodology [for KBS development] as RUDE: **R**un-**U**nderstand-**D**ebug-**E**dit". This general KBS development model is shown in Figure 3-31, and accentuates the fact that many KBS development lifecycles follow a highly iterative, prototyping path which lacks both a method for implementing the initial prototype and a method of controlling the iteration. Consequently, such RUDE models can cause problems in the system's documentation, maintenance and testing, and perhaps result in a poorly engineered and unsuccessful KBS.

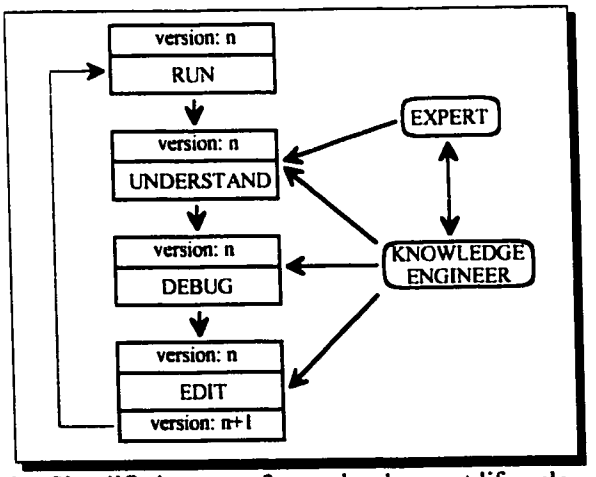


Figure 3-31. Simplified RUDE software development lifecycle. *[Bad91], p. 391*

The solution proposed by Bader and Edwards is the POLITE (Produce Objectives – Logical/physical design – Implement – Test – Edit) software development lifecycle model for engineering knowledge-based systems (Figure 3-32). The model is an extension of the classical waterfall model, as evidenced by the classical waterfall activities in the left half of its development phases, which are used to develop the conventional components (non-heuristic parts) of the system. In addition to the classical activities, the model adds parallel activities for development of the knowledge-based system components.

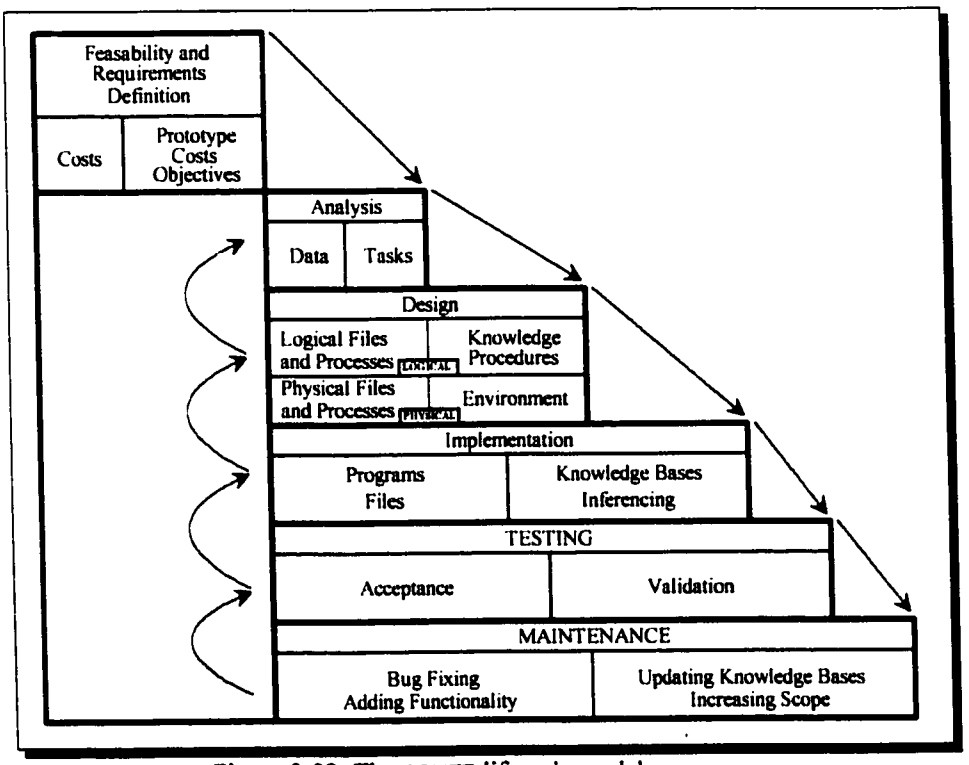


Figure 3-32. The POLITE lifecycle model. *[Bad91], p. 392*

Bader and Edwards ([Bad91], p. 405) summarize their conclusions about the POLITE model as follows:

- "KBSs must be engineered using a [software development lifecycle] if commercial and industrial standards of reliability and maintainability are to be met.
- Operational KBSs are likely to contain both conventional and heuristic components, necessitating the use of a hybrid development life-cycle.
- The RUDE AI methodology can be made POLITE by adapting the conventional waterfall model.
- The POLITE life-cycle, and accompanying standards, must be proved by using the model on commercial and industrial KBS development projects."

It is evident from the above that researchers have recognized that knowledge-based system development has been lacking in the application of accepted and proven methods for conventional software engineering, and further that it is likely possible that standard, conventional software development lifecycle models can be extended to incorporate development pathways for knowledge-based systems components.

◆ ◆ ◆

This section has served to introduce the broad field of macro-scale software engineering by describing several different software development lifecycle models, the essential backbones of macro-scale software development projects. It must be stressed that macro-scale software engineering involves much more than mere lifecycle models, but that much of this field, due to its immensity, is beyond the scope of this thesis. In the next section, a brief synopsis of micro-scale software engineering concepts is presented.

3.4.2 Micro-scale Software Engineering

Micro-scale software engineering is concerned more with *programming* than with project management. There is a wide range of concepts that fall into this category, including: ① Software tools for aiding and automating software development (CASE tools); ② A collection of programming techniques that can be referred to as *RIMS*: Reuse of code, Information hiding, Modularity and Specification; and ③ Object-oriented programming.

The realm of CASE tools is vast, and cannot be adequately covered in this thesis. Suffice it to say that many different types of CASE tools exist, including facilities for: managing source code files and monitoring revision tracking; automating source code generation from specifications; automating and rigourizing testing; aiding in user interface design; and generation of consistent documentation.

CASE tools research and development is currently a burgeoning field as software projects become increasingly larger in scope while developers seek to avoid the commensurate increases in workforce size. This enthusiasm in CASE tool development is a trend towards what Jacobson ([Jac92], p. 37) describes as the "industrialization" of software engineering. Jacobson (*Ibid*, p. 39) writes that CASE tools "can lead to massive

productivity improvements, but they are only part of a greater whole. The choice of a development technique has its roots in the basic philosophy chosen to govern the overall system structure of the designed systems, namely the architecture. To this base then is added a method, a process and, finally, computer-aided tools."

3.4.2.1 RIMS Programming Concepts

RIMS is an acronym coined by the author for a collection of valuable programming practices that are considered important aids for effective micro-scale software engineering: Reuse of code, Information hiding, Modularity and Specification. For the sake of conceptual clarity, these practices will be summarized in an order different from their enumeration in the RIMS acronym.

The term *modularity and specification* refers to the practice of designing and implementing a software product as a series of interconnected *modules*.[22] Sage and Palmer ([Sag90], p. 197) explain that "in most programming languages, a module is an entity in itself and operates on data in the way determined by the larger program of which it is part". The design of a modular program involves *specification* of how the modules should be organized, what information they should contain, how they should behave, and how they should interact. Interestingly, a proper specification does not indicate *how* the module's internal workings should be coded, that is, what algorithms are to be used internally; this detail is left to the discretion of the module's developer, with only the requirement that the module meets its specification. The specification formalizes the software design, enabling separate groups of software developers to code separate modules independently. The programmers are required to conform to the specification (to ensure that the pieces 'fit together' properly), but are free to choose how the module's inner workings are implemented. Furthermore, at a later time, the programmers are free to change the internal workings without necessarily being concerned that the rest of the program will cease to function, provided that the original specification is still followed.

Modularity lends itself to *reuse of code* by increasing software granularity. Proper modularization of a software product enables the implementation of generic or general-purpose software modules (libraries) which can be repeatedly used in other

[22] The literature uses several different words for a *module*, including *unit*, *library*, *monitor*, and *package*.

programs. Sage and Palmer ([Sag90], p. 330) write that "software reusability concerns the use of already constructed software 'parts', perhaps even 'conceptual parts', that are available from other software development programs, in new software development situations." Code reusability can significantly reduce the amount of effort and investment expended in re-creation of program code.

In addition to promoting reuse of code, modularity facilitates *information hiding*. According to Sage and Palmer ([Sag90], p. 193-194), information hiding is "the ability to conceal the existence of certain pieces of information and deny access to it except under direction of specific rules that have been predefined". Information hiding can endow software modules with localization of change by limiting the extent to which changes need to be made to program code. Typically, the module is coded so that only special, privileged subroutines are able to access or alter the information. By isolating (hiding) pieces of information in modules, and then tightly controlling and limiting access to only those modules for which the information is essential, the disruption is minimized (localized) when the pieces of information, or their manipulation subroutines, are changed. Hence, "most of the data and procedures are hidden from other parts of the software and the introduction of inadvertent errors is not likely to propagate to other locations in the software", (*Ibid*).

Together, the RIMS concepts, if applied properly, offer great benefit to software development by formalizing program design through the use of specification, and promoting efficient software implementation and maintenance through the use of modularity, reuse of code and information hiding.

3.4.2.2 Object-oriented Programming

Object-oriented[23] programming (OOP) has its roots in the SmallTalk programming environment, which has been in ongoing development at the Palo Alto Research Center of Xerox since 1976, ([Gol83]). The concept, which is a radical departure from the so-called 'conventional' realm of computer language design, is based on the premise that data objects take an active role in computing. Whereas regular procedural languages deal with

[23] The term *object-orientation* is sometimes used in this context, perhaps improperly, to describe the condition of something being object-oriented. Use of this aversive term is avoided in this thesis.

active procedures affecting passive data objects, object-oriented languages use data objects which affect themselves and possibly other data objects.

As a comparison, consider the general case action *print the value of x*. A procedural language would use a statement such as `print(x)` in which `print()` is an active procedure which is *passed* the passive data object `x` and prints it. An object-oriented language employs a statement which is almost an inversion of the `print(x)` statement, `x:print`, which says, essentially, *'Hey. x! Print yourself!'*. That is, the object `x` is *sent* the *message* `:print` which it recognizes as the name of one of its *methods* (actions that it knows how to perform). The contrast between these two different approaches to the same task is an important one, forming the waypoint from which object-oriented programming departs the road of conventionality.

The usefulness of OOP is something which can well be appreciated. In the past there was a pervasive semantic gap between problems or issues in the real world and their analogous computerized models. A large amount of effort was expended in order to simulate real events and objects using numerical processing engines. With the advent of OOP, however, the task of modelling the real world becomes a mere projection process from the concrete to the abstract. A data object in the computer is defined as an image or facsimile or a real object, possessing simulated characteristics and properties of the real object as the programmer sees fit. In such a way, the programmer discards the procedural approach of defining passive data structures, and active procedures that provide the modelling, in favour of data objects which are able to model their concrete counterparts in an active, participative manner.

It has been argued by Pascoe ([Pas86]) that a programming language requires four properties in order to be classified as object-oriented, and these properties, as described by Pascoe, neatly parallel the RIMS practices described previously. The first property is that of *information hiding*, which ensures both software reliability and modifiability by *modularizing* the software and minimizing inter-module dependencies and influences. Information hiding represents the application of a 'black box' approach to software development in which the internal details of modules are free to change without affecting

the functionality of the whole. Second is the property of *data abstraction*, the utilization of information hiding to design *classes* or *types* of data objects that possess informative attributes and inherent behaviours. Data abstraction offers the first step towards procedureless programming by offering a primitive form of 'objectism'. Third, the property of *dynamic binding* is required. Dynamic binding enhances programming by offering *polymorphism* to data classes; operations (behaviours) in different classes may be identified by the same names and the onus is placed on the data objects themselves to resolve the conflicting references at run time instead of compile time. This highly desirable property encourages semantic simplification of programs because operations from different classes which behave intuitively in a similar manner may be referenced by the same name, thereby reducing the complexity of the program as a whole. Finally, an object-oriented language must provide a mechanism of *inheritance*: classes of objects must be able to share operations in common. Inheritance permits the definition of *sub-classes* which are more-specific versions of their *parent class* (or *parent classes*) but still share (inherit) properties from the parent class(es). The property of inheritance represents the zenith of OOP by providing the highest level of real-world modelling. Inheritance facilitates code reusability and eliminates code redundancy since classes that share common characteristics may be enclosed under a parent class which holds their common qualities for all to use. Together, these four properties form the minimum requirements for an object-oriented language. The all-important properties of inheritance and dynamic binding rely on those of data abstraction and information hiding, so all are necessary qualities of an OOP language.

The issue of usefulness of object-oriented computing ultimately reduces to two significant benefits: ① The transition from the concrete to the abstract is simplified since object-oriented languages are *procedureless* and, therefore, are simpler, more accurate models of real-world events. This implies that the software developer should experience increased productivity and greater understanding of how the model achieves its required analogy to the real-world event; ② The reusability of software components through inheritance promotes a greater software life expectancy: code fragments may be used and reused as often as needed but only need to be developed once.

* * *

This chapter has introduced four areas of electrical and computer engineering that are relevant to the computer-centered technology insertion operation. Each of these areas is so large that a brief introspective does not do them proper justice. Nevertheless, the concepts presented in this chapter pave the way for the subsequent chapters, which explore the design and implementation of the Particle Accelerator Control Expert System, and suggest how its design and implementation principles can be applied to computer-centered technology insertion operations in general.

Chapter 4

Design and Implementation of PACES

This chapter introduces the reader to the Particle Accelerator Control Expert System (PACES), describing what it does, and how it is designed and implemented. Within the context of this thesis, PACES serves as a case study illustrating the advocated methodology for computer-centered technology insertion.

This chapter is divided into five sections: The first provides an overview of PACES, including its organization and operation. The second section describes how aspects of real-time systems design are utilized in PACES. The third section discusses principles of artificial intelligence which are used in PACES, particularly knowledge-based reasoning. In the fourth section, the incorporation of human factors concepts into PACES is presented. The fifth section deals with how software engineering principles have been applied in PACES design and implementation to integrate these different modalities into a working control system. As will be argued in Chapter 6, many of the decisions made during design and implementation of PACES can be applied to other computer-centered technology insertion operations.

4.1 Overview of PACES

PACES is a large, multi-module computer program which runs in part on one or more embedded controllers and part on a host computer (PC). With respect to the technology insertion problem described in Chapter 1, PACES incorporates aspects of three broad areas of computer science and computer engineering: real-time systems, artificial intelligence and human factors. As depicted in Figure 4-1, practices borrowed from these multi-disciplinary fields are combined in the present work into a cohesive hybrid system by employing principles of software engineering that promote modularity and integration of

program source code. Simultaneously, electrical engineering principles are used for interfacing with the accelerator, and computer-human interface principles serve to 'connect' PACES with the operator.

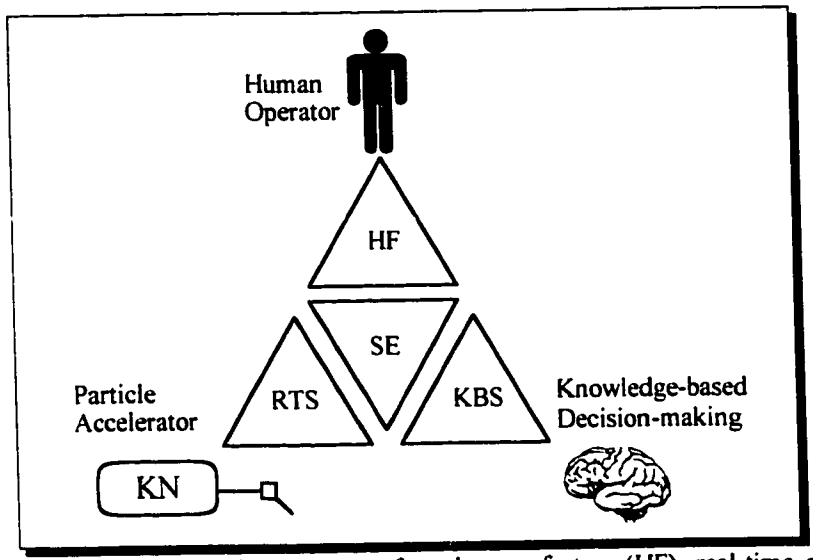


Figure 4-1. PACES comprises concepts from human factors (HF), real-time systems (RTS) and knowledge-based systems (KBS). Software engineering (SE) is used to meld these disciplines into a multi-disciplinary software system.

4.1.1 System Organization and Operation

PACES is a complex, distributed control program that runs on two or more hierarchically organized computers (Figure 4-2). The overall control task is divided between the processors to yield fast response times to both the operator and accelerator, ([Poe91]).

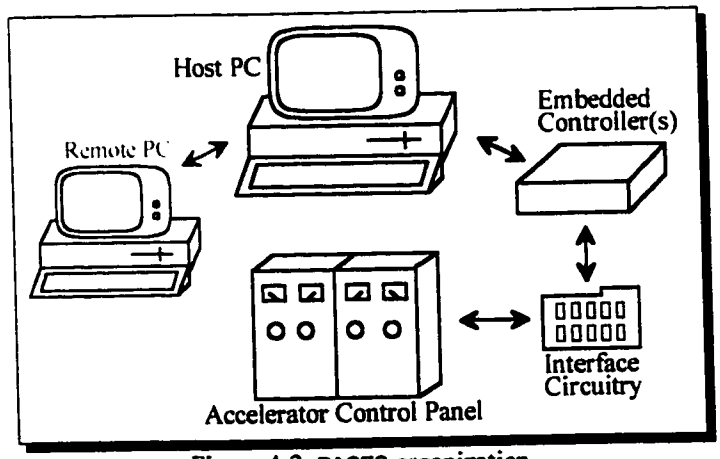


Figure 4-2. PACES organization.

The PC is connected to one or more single-board-computer (SBC) *embedded controllers* (see Figure 4-2), which are linked to the accelerator via interface circuitry. The SBCs are data acquisition and control systems. Presently, only one Intel 8051-based SBC is used, but PACES has been designed to accommodate other embedded processors as the system is expanded. The single SBC is responsible for acquiring and conditioning (filtering) accelerator operating parameters and relaying them to the PC. It also carries out accelerator control actions as directed by the PC, employing a fuzzy logic inference engine for low-level decision-making. As will be discussed in Section 4.2.1, additional SBCs can be added to the system as required to extend the capacity of the accelerator interface, such as, for example, inclusion of control over downstream beam line components.

The top level (*host*) computer is a PC with an 80486 or Pentium™ processor. A high resolution colour graphics monitor displays a facsimile control panel[24] (Figure 4-3). This graphical user interface (GUI) mimics the real control panel as closely as possible. Graphical meters, indicator lights and other visual aids popular in current GUIs display the accelerator's state. The operator uses the computer's keyboard and mouse to actuate the accelerator's controls (selsyns and switches). The PC incorporates a knowledge-based expert system for automated operation and fault handling.

Control of the accelerator must be performed in real-time, with response times of no more than one or two seconds. The control problem is complicated by the fact that both high resolution graphics (for the user interface) and knowledge-based reasoning (for automated operation) are computationally intensive. Since it is inappropriate to overload a single processor with such a tall order, the control problem is best solved in a parallel fashion by employing more than one processor. Thus, PACES divides the overall control problem to facilitate fast response time to both the accelerator and the user. Although such a parallel approach is well known, the novelty of the present solution centers on the use of knowledge-based reasoning for accelerator control. In this instance, the PC performs 'high level' tasks such as generating the user interface and performing rule-based reasoning, while the embedded controller performs 'low level' tasks such as data acquisition and direct control. An important issue is the *partitioning* of the overall control task to provide,

[24] Note that since the GUI for the AECL version of PACES is shown, the *extraction* selsyn is labelled as *beam bias* (cf. § 2.3.2).

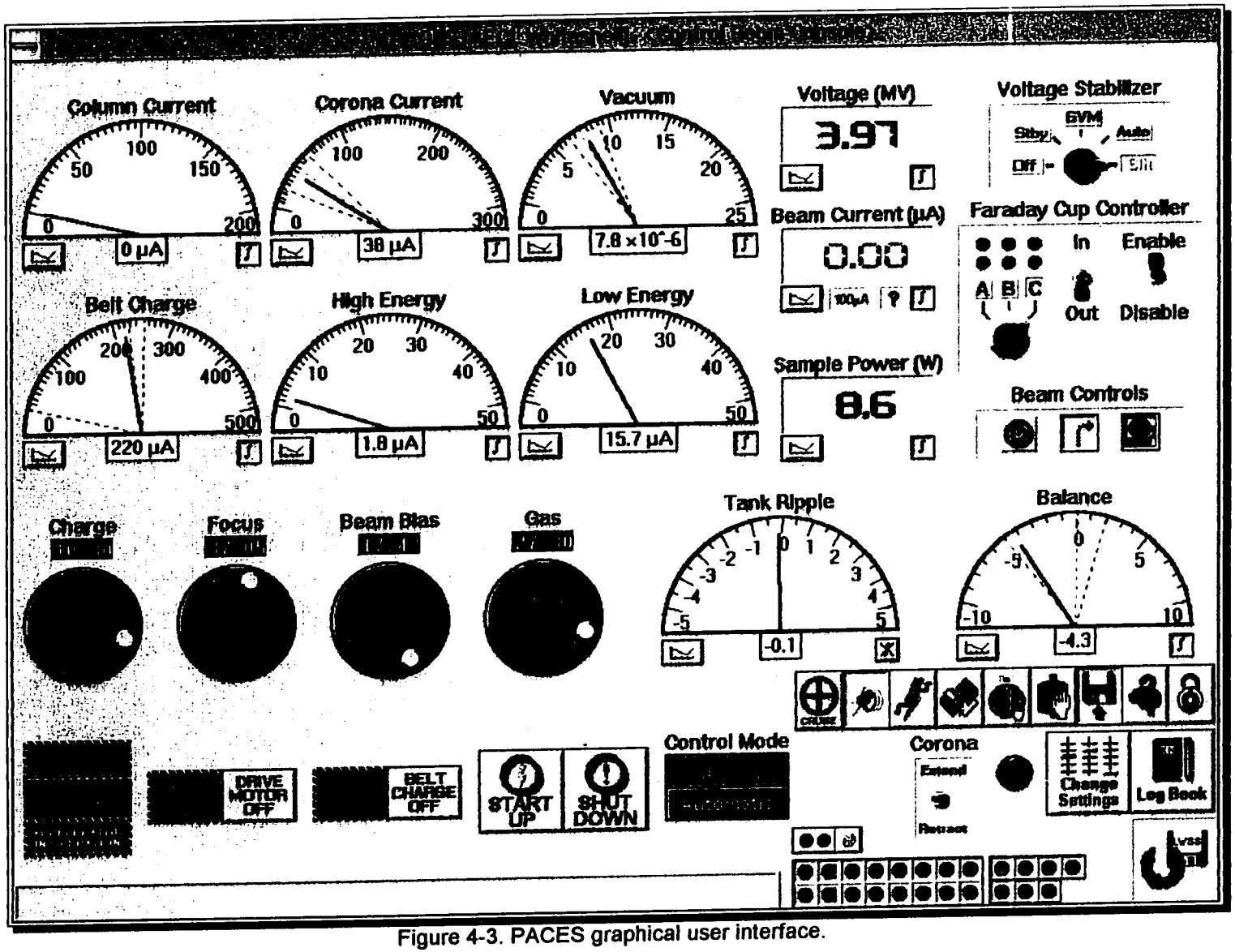


Figure 4-3. PACES graphical user interface.

simultaneously, fast response time and a high degree of 'intelligence' in the form of expert-based knowledge.

The host computer may also be connected to one or more remote consoles, typically IBM-style PCs. The remote consoles are connected to the host via RS-232 serial links, and may use modems for communication over telephone lines. The remote consoles act as functional extensions of the host computer's user interface, enabling the accelerator operator to monitor and control the machine from locations other than the main control room. This is useful during experiments when the experimenters spend most of their time in the target area but still wish to have control over the accelerator without having to travel to the control room. Additionally, the use of a modem over telephone lines enables a person at another location, perhaps many miles away, to operate the accelerator remotely for expert diagnosis or troubleshooting purposes.

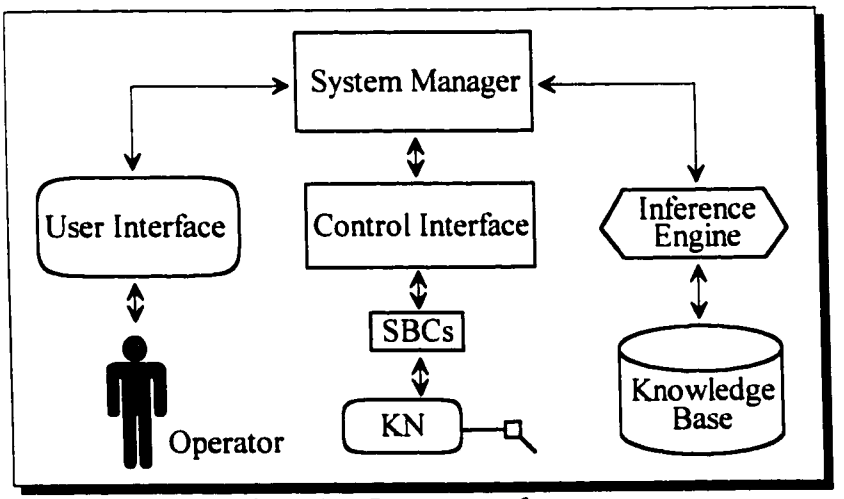


Figure 4-4. Components of PACES.

The overall system can be conceptually divided into five parts (Figure 4-4). The *system manager* is the top level of the program that oversees and co-ordinates the other parts. The *user interface* is the system's link with the operator. The *control interface* is responsible for uploading telemetric data from the SBCs and sending them control commands. Finally, the *inference engine* and *knowledge base* are used for making heuristic decisions.

The most substantial portion of PACES runs on the host computer (PC). The PC program is a Microsoft Windows[25] application[26] written in Borland Pascal for Windows[27] (BPW). It is concerned with maintaining the graphical user interface (GUI) and performing high-level reasoning for controlling the accelerator. PACES is written in a highly modular, object-oriented manner to promote software engineering (as will be discussed in Section 4.4). The heart of the program is the 'control panel' (cf. Figure 4-3), a graphical facsimile of the real accelerator control panel through which the user interacts to operate the accelerator manually or to initiate automated operation.

4.1.2 Manual Operation

PACES provides the capability for the operator to run the accelerator manually in a fashion that closely mimics manual operation using the real control panel. The operator is free to follow the prescribed start-up and shut-down procedures, and to perform whatever control actions are warranted during the accelerator run. During start-up (or shut-down), the operator uses the mouse to turn on (or off) the control power, drive motor and belt charge switches (Figure 4-5).

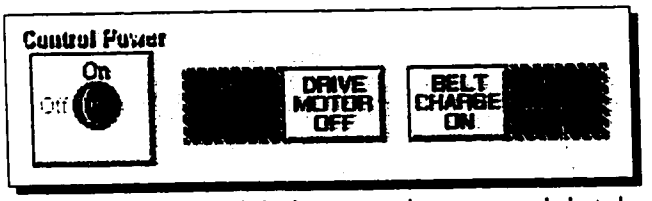


Figure 4-5. Switches used during manual start-up and shut-down.

During start-up, the operator next turns the four selsyns to settings appropriate for the day's run. This can be accomplished using the *selsyn controller* window (Figure 4-6). The controls of this window allow the operator to select a selsyn and turn it up or down by a specific amount, or to command a selsyn to turn to a specific position. The selsyns can also be manipulated, without using the selsyn controller, from the main control panel window by placing the mouse over the desired selsyn and using the left or right mouse button to decrease or increase the selsyn's position, respectively. The operator is also able to control the voltage stabilizer's mode and Faraday cups (Figure 4-7).

---

[25] Microsft Corp., Redmond, WA.
[26] The term *application* refers to the overall program in a general sense; PACES is one instance of an application.
[27] Borland International, Scotts Valley, CA.

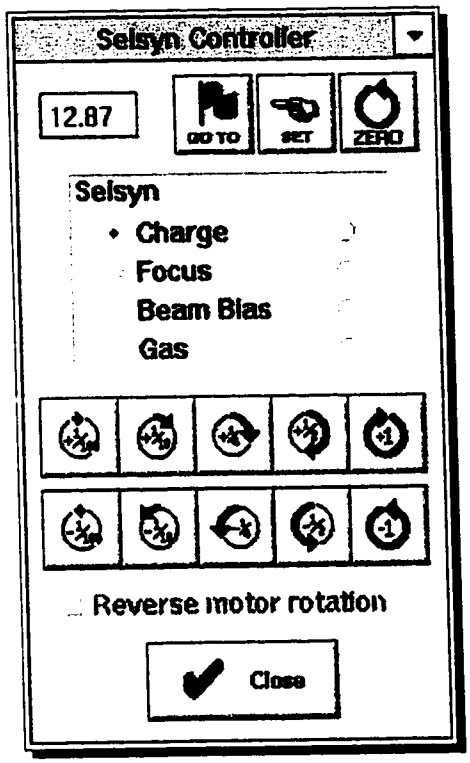


Figure 4-6. Selsyn controller window.

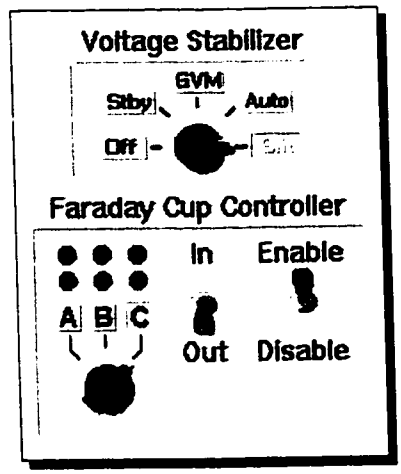


Figure 4-7. Voltage stabilizer mode and Faraday cup controls.

4.1.3 Automated Operation

The manual mode of operation is augmented by automated operation which the operator can to choose to invoke during start-up, shut-down or beam maintenance modes. The buttons shown in Figure 4-8 are the main controls for invoking automated operation. The Start Up button is used to begin automated start-up, and automated shut-down is initiated using the Shut Down button. The Control Mode button is used during beam maintenance operation to switch between automatic and manual beam maintenance. Collectively, these buttons control the PACES knowledge bases responsible for automated operation. The Start Up and Shut Down buttons trigger knowledge-based decision making that mimics the established start-up and shut-down procedures, while the Control Mode button activates (or deactivates) knowledge-based control loops that optimize and stabilize the accelerator during beam maintenance mode.

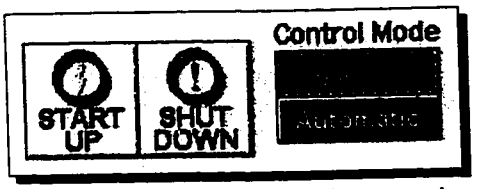


Figure 4-8. Automated operation controls.

When automatic start-up is initiated, the operator is given the opportunity to select the accelerator's initial settings from a database of past settings (Figure 4-9), or to specify target settings for the start-up procedure (Figure 4-10). In either case, these settings are used by PACES to establish set-points for controls and target values for machine parameters such as terminal voltage and beam current. Furthermore, the operator is able to specify *tolerance intervals* (upper and lower limits) for these target parameters (Figure 4-11).

Automated accelerator operation is discussed further in Section 4.3.

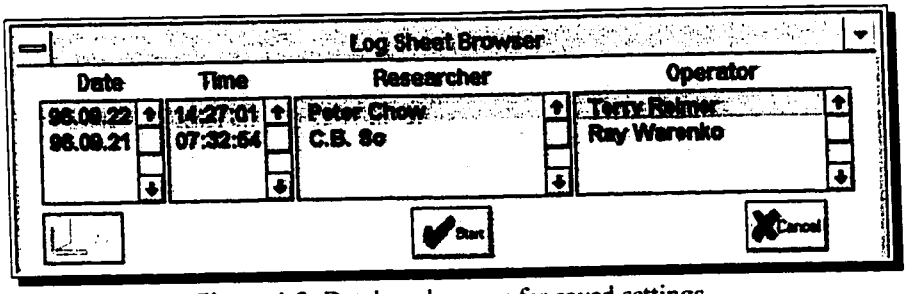


Figure 4-9. Database browser for saved settings.

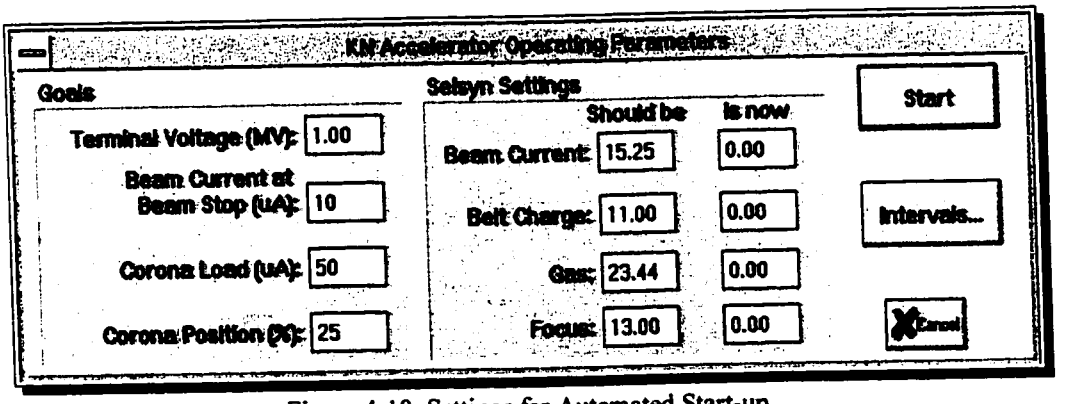


Figure 4-10. Settings for Automated Start-up.

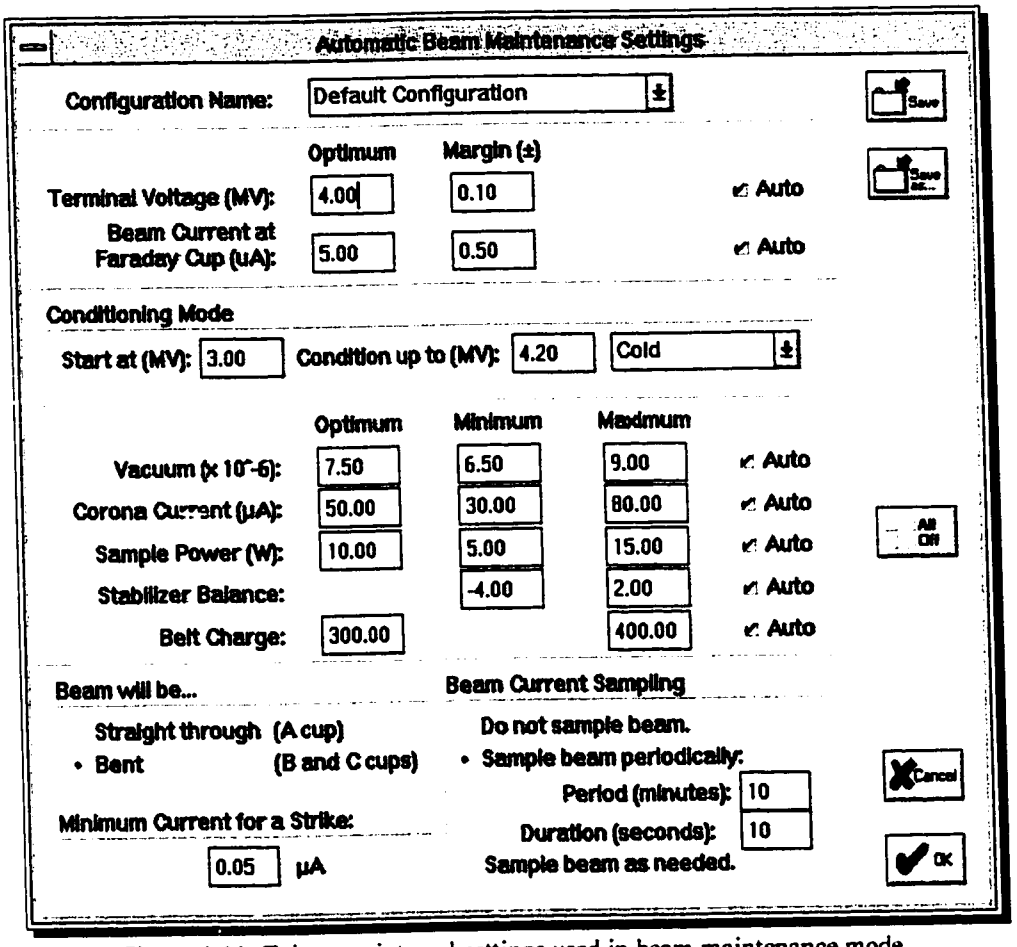


Figure 4-11. Tolerance interval settings used in beam maintenance mode.

4.1.4 Miscellaneous Features

This section briefly summarizes some of the additional features of PACES to illustrate how the program can be used to aid users in accelerator operation. These features are referred to as *tools* (shown in Figure 4-12). The tools not described in this section are detailed in the sequel.

The *selsyn controller* tool (Figure 4-12a) was mentioned in Section 4.1.2. The *data logger* tool (Figure 4-12b) is used to turn on or off the program's data logging facility, which writes time-stamped accelerator telemetry data records to a disk file. The *stepper motor power* tool (Figure 4-12d) enables the user to turn off power to the selsyn stepper motors so that the selsyns can be turned manually. The *decisions explainer* tool (Figure 4-12e) opens a window that displays knowledge-based decisions as they are made (cf. § 4.3.2.5). The *SBC reset* tool (Figure 4-12f) can be used to reset the embedded controller manually. The *voltage conditioner* tool (Figure 4-12g) is used to initiate

terminal voltage conditioning manually if the operator decides that the accelerator is unstable. The *audible warning control* tool (Figure 4-12i) turns on (or off) the audible warning sounds of the 'spark' and 'beam lost' alarms. The *SBC settings update* tool (Figure 4-12j) is used to alter the SBC's record of selsyn positions after any selsyn has been adjusted manually without using the software.[28] The *beam maintenance settings* tool (Figure 4-12k) is used to change beam maintenance settings when needed (cf. Figure 4-11). The *kiviat graph* tool (Figure 4-12l) opens/closes the accelerator kiviat graph (described in § 4.4.2). The *analyzing magnet controller* tool (Figure 4-12o) gives the user control over the analyzing magnet. Finally, the *remote operation* tool (Figure 4-12p) controls remote operation of the accelerator: the user can activate this tool on either the control room console or remote console to switch accelerator control between the two locations.

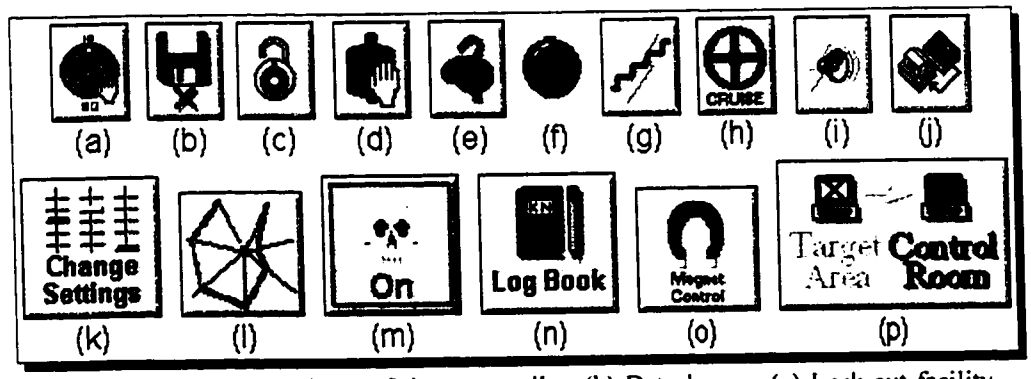


Figure 4-12. PACES tools: (a) Selsyn controller. (b) Data logger. (c) Lock-out facility. (d) Stepper motor power. (e) Decisions explainer. (f) SBC reset. (g) Voltage conditioner. (h) Cruise control configuration. (i) Audible warning control. (j) SBC settings update. (k) Automatic beam maintenance settings. (l) Kiviat graph. (m) Idle watchdog. (n) Logbook. (o) Analyzing magnet controller. (p) Remote operation control.

4.1.4.1 Lock-out Facility

The *lock-out facility* (Figure 4-12c) is a safety mechanism for the accelerator control room's exit door which is used during unattended operation to detect unauthorized entry to the control room. If the exit door is opened during unattended operation, the accelerator is placed in safe mode unless the proper password is entered before a time limit expires.

[28] This is necessary because PACES has no sensors to detect when a selsyn has been adjusted manually.

4.1.4.2 Logbook

The *logbook* tool (Figure 4-12n) is an electronic version of the operators' logbook that is kept on hand at the control panel (cf. Section 2.5). This tool consists of five sub-tools as shown in Figure 4-13. The Save Settings tool is used to save the current accelerator state variables in the settings database for future use during start-up for restoration of accelerator state. The Browse tool opens the accelerator settings database (cf. Figure 4-9) for selecting past run settings for replication during start-up. The Operator Notes tool accesses a free-form, text-based memo pad that the operator can use to make notes about the accelerator, run or experiment in progress. The Service Memo button is used to issue service requests to be processed by maintenance personnel. Finally, the Journal button permits viewing of PACES' *significant event journal* (see Figure 4-14), a database which stores descriptions of events occurring during accelerator operation which are worthy of logging for future reference.

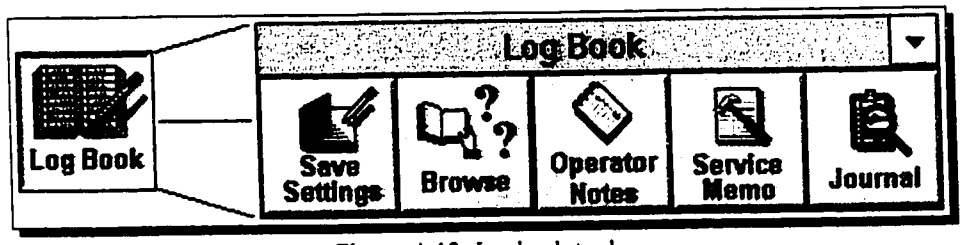


Figure 4-13. Logbook tools.

When the user activates the Save Settings tool, a logbook database entry form is presented that closely resembles the format of the regular logbook (Figure 4-15). PACES fills in all the fields it is able to, and the user is expected to complete the remaining information before committing the entry to the database.

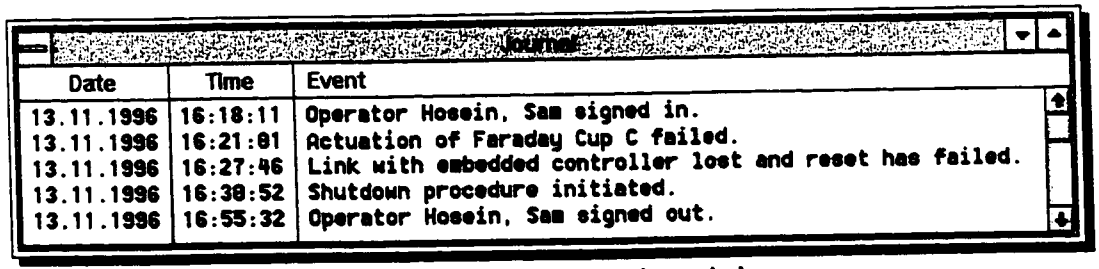


Figure 4-14. Significant events journal viewer.

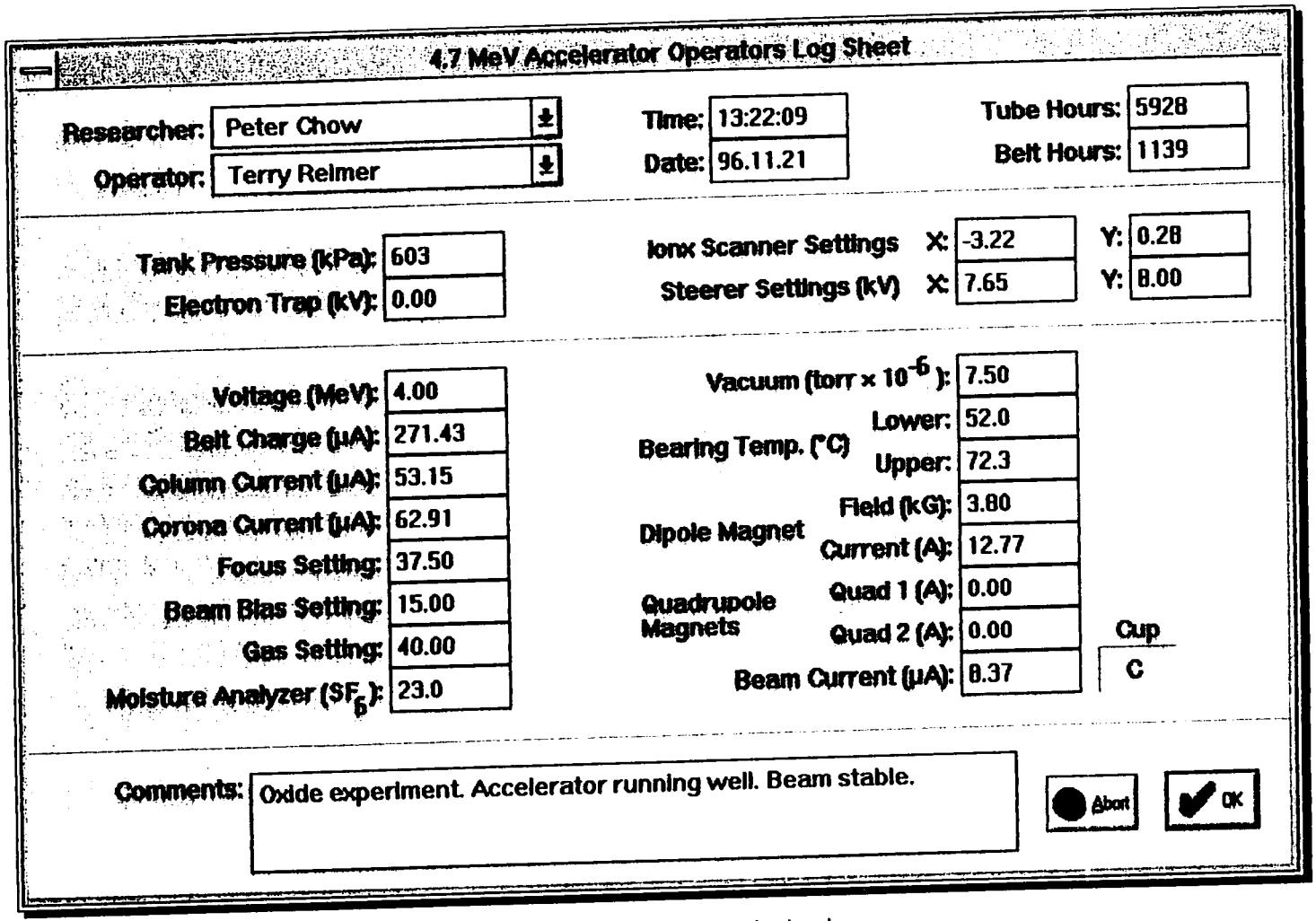


Figure 4-15. Accelerator logbook.

4.1.4.3 Idle Watchdog

The 'idle watchdog' is a simple safety feature intended to protect both the accelerator and personnel during automated accelerator operation. The idle watchdog monitors operator activity (keyboard and mouse) to determine whether an operator is present at the computer console while PACES is operating the accelerator, and will invoke safing action if user inactivity exceeds a programmed time limit. Consequently, this feature forces an operator to interact periodically with the computer, and it is assumed that such interaction involves periodic monitoring and supervision of the control system and accelerator. If the idle watchdog detects operator inactivity, it assumes that the accelerator is running unsupervised and initiates safing action or shut-down for safety.

4.2 Aspects of Machine Interfacing and Real-Time Systems

As mentioned previously, the 'low end' of the interface problem is concerned with effectively connecting the computer system with the accelerator. This involves two steps. First, the physical link between computer and accelerator must be realized in the form of data acquisition and control hardware. Then, built upon the physical interface layer, the conceptual link for information interchange (using software) is implemented. Construction of the physical layer relies upon established electrical engineering techniques. Likewise, the conceptual layer is developed using real-time systems concepts.

The following subsections deal with some of the issues prevalent in the design and implementation of both the physical and 'informational' levels of the accelerator interface.

4.2.1 Multi-processor Environment

As described in the section on real-time systems in Chapter 3, real world processes are inherently concurrent, with multiple aspects existing and operating simultaneously. Consequently, a real-time system built to interact with a real-world process typically mirrors this real-world parallelism in the form of multiple (software) processes, possibly operating on multiple processors. This principle of parallelism has been applied to PACES in the interests of promoting fast response time to both the user and accelerator. The overall control task is partitioned between two or more processors of two basic types. The so-called 'high level' tasks of user interaction and knowledge-based reasoning are

performed by the host PC, while the 'low level' tasks of data acquisition and accelerator interfacing are performed by one or more embedded controllers.

The PC is a (relatively) expensive, high performance, general purpose computer with input devices (keyboard and mouse) and an output device (high resolution colour graphics monitor); it stores the bulk of the PACES software and knowledge base. In this way, it can be likened to a human's head which comprises a brain, an assemblage of input devices (eyes, ears, nose), and an output device (mouth).

The embedded controllers are, in comparison, simple and cheap single-board computers with limited memory and processing ability. They do not communicate directly with the user, and thus do not possess 'conventional' input/output devices. Instead, they communicate with and are controlled by the PC, which presides over them as a higher-level co-ordinator; and, on behalf of the PC, they acquire data from the accelerator and perform direct control actions on the accelerator. In this sense, therefore, the SBCs can be compared to the human body's sensory/motor system (sense of touch and motor muscles).

The embedded controllers are imbued with limited ability to make control decisions within limited bounds set by the PC. Situations that the embedded controllers are unable to handle are relegated to the PC. This approach serves to shift some of the system's overall duties from the PC to the embedded controllers, thereby relieving the PC of some of the workload. Continuing the body analogy, it is possible to imagine the processor hierarchy within PACES as being similar to the organization of *reflex arcs* in the body's nervous system. Gleitman ([Gle81], p. 28) describes a reflex arc as:

> "...the reflex pathway that leads from stimulus to response. Some reflexes represent a chain of only two components, as in the case of an afferent neuron which contacts a motoneuron directly. More typically the chain is longer, and one or more interneurons are interposed between the afferent and efferent ends. We can also ask whether the reflex arc also involves the brain (and if so, which part)..."[29]

Figure 4-16 illustrates an old conceptualization of reflex arcs: When a person's limb strays too close to an open flame, a reflex arc pulls the limb away. The 'decision' to pull

[29] The term *efferent nerve* can be interpreted to mean 'motor' (output) nerve, and *afferent nerve* to mean 'sensory' (input) nerve.

the limb away is made outside the brain by a *flexion reflex* which is formed by a chain of neurons. The chain originates at the thermal sensors in the limb's skin, travels to the spinal cord, and then back to the muscles of the arm. The 'decision' to retract the arm is an *involuntary reflex* made in this reflex arc, without any assistance being needed from the brain. Indeed, by the time the sensation of pain reaches the brain, the flexion reflex has already acted and pulled the limb away. The involuntary reflex system is paralleled by the *voluntary action system*, in which muscle action is caused by 'voluntary' motivation from the brain instead of from a reflex action.

Figure 4-16. Reflex action as envisioned by Descartes. *[Des62] cited in [Gle81], p. 16*

The nervous system of the human body, and higher-order animals in general, is organized in this manner because there is a propagation delay imposed in sending signals from sensory nerves up through the spinal column to the brain and then back down to the motor nerves. This delay would cause poor response time (and perhaps injury or death) were it not for the presence of reflex arcs, which serve to provide the body with localized, fast (but limited) decision-making capability. In essence, therefore, *reflex arcs improve system response time by adding an element of local parallelism.*

The PACES processor hierarchy is organized in a similar fashion (Figure 4-17). Some (simple) decisions can be made by the SBCs without intervention by the PC, while other (more complicated) decisions require the PC's more-sophisticated decision-making. Such distribution of the workload serves to decrease response time to the accelerator by shifting some of the workload to the embedded controllers, and freeing the PC from some of this

burden, thereby improving response time to the user. The issue of workload partitioning is explored further in Section 4.3.

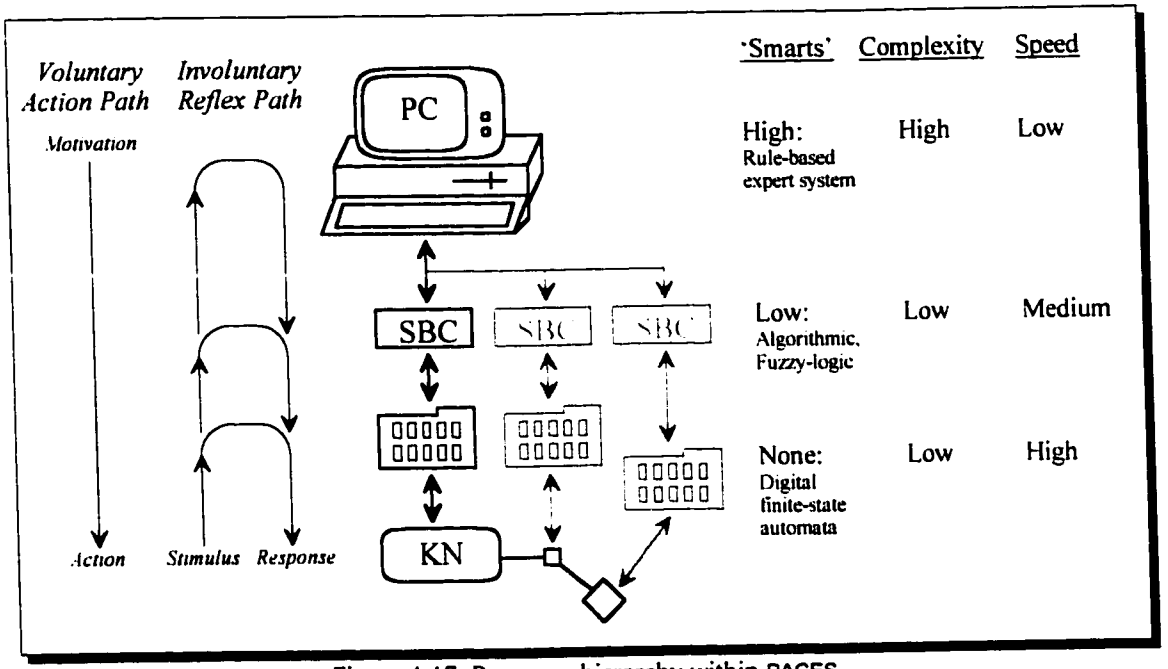


Figure 4-17. Processor hierarchy within PACES.

It is worth noting that the accelerator interface circuitry can be considered to have a rudimentary level of decision-making capability in the form of digital finite-state automata (DFSAs). There are presently two DFSAs in the interface circuitry: one is responsible for counting motor steps and stopping a selsyn stepper motor after a specified number of steps; the other is incorporated into the analog-to-digital converters, which use successive approximation to convert their analog inputs into digital outputs. Both of these DFSAs are 'mindless' in terms of artificial intelligence, but do possess some simple decision-making capability, and are able to function autonomously to some degree. Thus, the DFSAs can be thought of as lower-order aspects of the system's sensory/motor reflex arc structure.

4.2.2 Accelerator Interface

The hardware underlying the PACES accelerator interface is designed with several key principles in mind. Above all, *simplicity* is the main issue: The circuitry should be simple in design to facilitate rapid prototyping and construction, and to promote ease of understanding and troubleshooting by people other than the system designers. This maxim

is merely a form of Occam's razor: *Entities are not to be multiplied without necessity*, ([Pea64], p. J32). Other important principles involved include:

- *Frugality*: The hardware design and implementation should be low-cost, making use of existing interface circuitry whenever possible .
- *Modularity*: The hardware should be modular to permit straightforward reconfiguration, extension and repair.
- *Flexibility*: The hardware should be flexible in its ability to be extended and modified, in response to both changes in the accelerator site and in the higher levels of PACES software.
- *Utility*: The hardware should, where appropriate, perform its duties autonomously without the need for constant direction by the embedded controller.
- *Passivity*: The hardware should minimally disrupt normal (manual) accelerator operation, in order to maintain accelerator operability during PACES development and during times when PACES is not functioning.

The accelerator interface is performed in a non-invasive, 'piggy-back' manner to permit easy installation and removal, and to minimize disruption of regular accelerator operation. Since the McMaster accelerator was used as a testbed for PACES early in its development, much of the hardware that distinguishes the McMaster accelerator from its DREO counterpart was irrelevant; the main control points (the selsyns and switches) are identical on the two machines, enabling much of the early development to be performed at McMaster. The accelerator interface circuitry was designed to be portable bewteen these two KN-3000 accelerators with minimal reconfiguration. (The field version of PACES later developed for the Whiteshell Labs KN-4000 incorporated several improvements over the original accelerator interface circuitry; some of these enhancements are described hereafter.)

At its lowest level, PACES is connected to the accelerator's control panel via custom data acquisition and control circuitry operated by one or more single-board computer (SBC) embedded controllers. A block diagram of these connections is shown in Figure 4-18a. The inputs to the control system are the signals from several analog meters on the control panel, and a series of digital inputs for sensing the states of control panel switches and indicators (such as the control power, drive motor on/off, states of Faraday cups, and voltage stabilizer mode switch). The outputs from the control system actuate

switches (using relays) or turn selsyns on the control panel (using stepper motors). Figure 4-18b shows how the accelerator fits into the PACES system as a whole: The outputs of the accelerator are fed back into PACES to facilitate closed-loop control. This scheme represents a non-linear, multi-variable control system that is impractical to model, and, therefore, the achievement of stable control is intractable using conventional control methods, ([Sin83]).

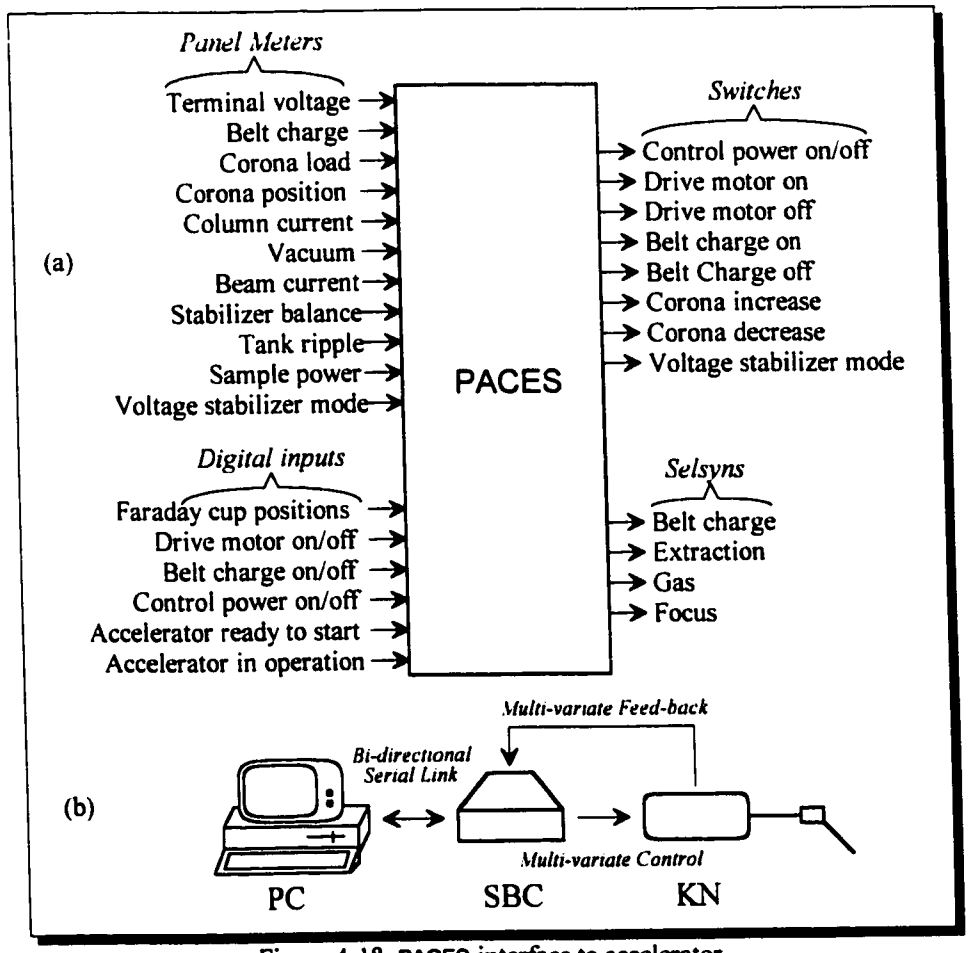


Figure 4-18. PACES interface to accelerator.

Figure 4-19 shows how the SBC is connected, through a memory-mapped input/output (I/O) space to subsystems for data acquisition and control point actuation.

The system uses two 10-bit analog-to-digital converters (ADCs) for data acquisition. Each ADC receives its analog input from an 8-to-1 analog multiplexer (AMUX). Computer control is used to select each ADC's input channel and also to start its conversion cycle. A ready signal is output from each ADC when its conversion cycle is complete.

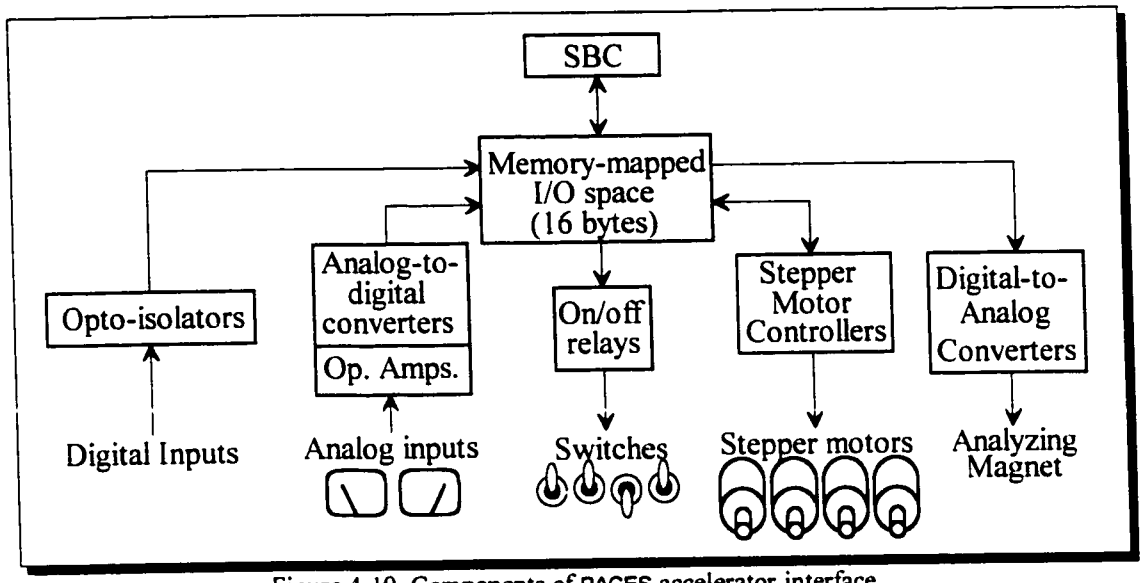


Figure 4-19. Components of PACES accelerator interface.

The first ADC samples eight voltage channels connected to control panel meters (Figure 4-20). These meter signals are amplified using three stages of linear operational amplifiers (unity-gain differencer, unity-gain filter, and adjustable-gain amplifier) before being fed to the AMUX.

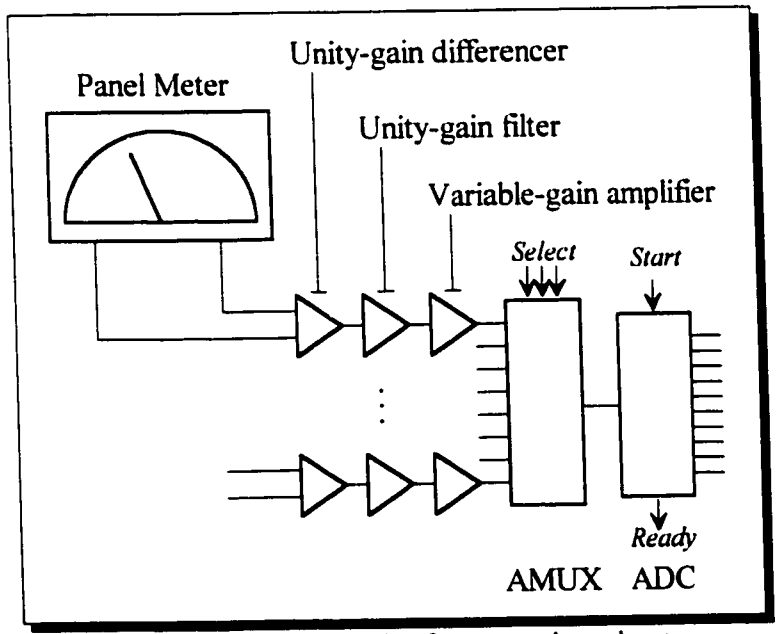


Figure 4-20. Signal acquisition from control panel meters.

The second ADC samples up to eight beam current channels, such as from the Faraday cup, target chamber or analyzing magnet chamber (cf. Figure 2-5). The beam current signals are amplified using logarithmic amplifiers so that the signal can range over several decades from nA to mA (Figure 4-21).

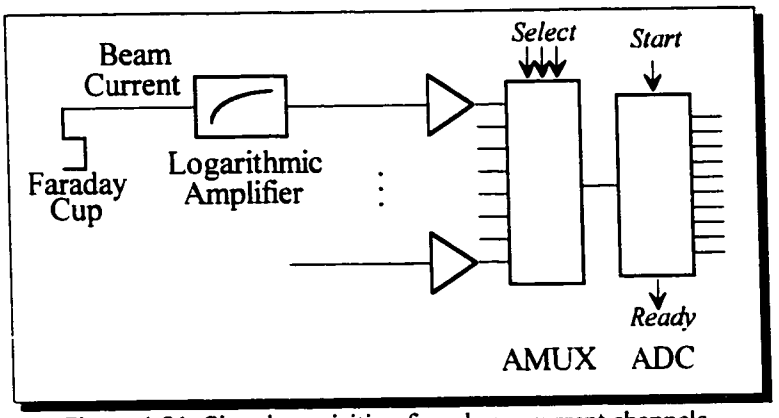


Figure 4-21. Signal acquisition from beam current channels.

Active control of the accelerator is accomplished using relays to actuate switches, stepper motors to turn selsyns, and digital-to-analog converters for controlling the analyzing magnet's power supply (cf. Figure 4-19). The relays are controlled by simple gating circuitry, and are connected to the control panel wiring to mimic manual switch activation; that is, when the control system is inactive, the control panel switches function as normal. Moreover, the control power switch is never bypassed, so manual intervention is always required to activate the accelerator's control power, and the entire system can always be manually deactivated at any time simply by turning off the main key switch.

Special circuitry is used to drive the four stepper motors which are mechanically connected to the control panel's selsyns. When the stepper motors are de-energized, the selsyns can be turned manually as normal.[30] The circuitry can be commanded to turn a specific motor an exact number of steps clockwise or counterclockwise, one step being 1/100th of a complete turn of 360°. Once a motor is turning, the circuitry is able to start another motor turning, enabling simultaneous manipulation of multiple selsyns. Additionally, the circuitry counts the number of steps applied to a turning motor, and stops a motor when its prescribed number of steps has been reached.

4.2.2.1 Enhancements at Whiteshell Labs

The accelerator interface circuitry installed on the Whiteshell Labs KN-4000 has several enhancements over the prototype circuitry installed at DREO. The analog inputs employ opto-isolation amplifiers to improve noise filtering and protect the circuitry from high voltage transients which sometimes propagate through the control panel. Also, since

[30] As explained in Section 4.4.2, the selsyns behave slightly differently when the motors are connected.

the beam current signals do not generally span 6 decades of range as they do at DREO, logarithmic amplifiers are not needed; consequently, the second ADC is not dedicated to sampling beam currents, and operates the same as the first ADC. The two ADCs are driven in parallel, so that two analog signals (out of 16) are sampled simultaneously.

One additional enhancement is the addition of 16 opto-isolated *digital inputs* which are used for sensing such things as the presence of control panel power, 'accelerator ready to start' indicator, 'accelerator in operation' indicator, state of the drive motor (on or off), state of the belt charge power supply (on or off), positions of the three Faraday cups (in, out or in transition), and the exit door interlocks (cf. § 4.1.4.1).

4.2.3 Non-invasive Machine Interface

As mentioned, interface to the accelerator is non-invasive. This is important and desirable for several reasons. The control system must be easy to connect and disconnect from the accelerator because the accelerator cannot suffer from long periods of down-time during system development. The system must not involve any alteration of existing mechanical, electrical or safety systems that would render the accelerator manually inoperable at any time. It is crucial that the operator is able to assume immediate, complete manual control of the accelerator whenever needed while using the computer system. Details of this 'piggy-back' approach to machine interfacing are published in [Lin91] and [Lin93b]. In essence, care is taken in PACES to ensure that run-away control conditions are avoided and that the operator can quickly assume complete, direct manual control at any time.

4.2.4 Embedded Controller

The embedded controller (Figure 4-22) used by PACES is a small, uniprocessor single-board computer[31] (SBC) built around an Intel 8051 CPU. The SBC communicates with the PC via an RS-232 serial link. The PC sends the SBC commands similar to *remote procedure calls* for requesting accelerator data or ordering control actions. The SBC presides over an ensemble of sub-systems that collectively form the data acquisition and control circuitry of the accelerator interface.

---

[31] HiTech Equipment Corporation, San Diego, CA.

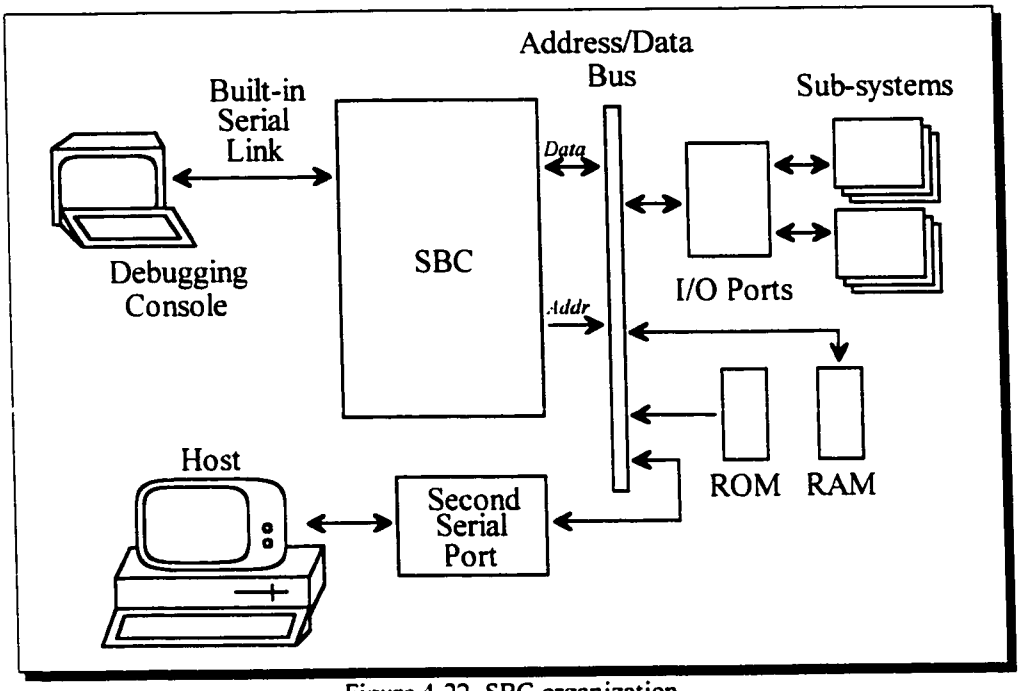


Figure 4-22. SBC organization.

The SBC possesses 8 kb of ROM which holds the firmware bootstrap monitor,[32] and 32 kb of RAM which is used for the PACES embedded controller real-time kernel (RTK). The SBC has two serial ports: one built into the CPU which is connected to a debugging terminal for feedback from the RTK during debugging, and another memory-mapped serial port which is used for the command/data link with the PC.

The CPU has four 8-bit 'multi-purpose' parallel I/O ports, but two of these are required for accessing the SBC's ROM and RAM. Of the two remaining ports (16 bits), 10 bits have special functions, leaving just 6 bits for general use. Since this renders the built-in I/O ports effectively useless, the accelerator interface is implemented using memory-mapped I/O channels (Figure 4-23).

Each sub-system of the accelerator interface is connected to the SBC's address/data bus, has a unique range of addresses, and includes decoding logic for detecting when the sub-system is being addressed by the SBC. Data sent to/from the sub-systems are passed from/to the data bus via tri-state latches. Conventional memory accesses (reads and writes) are used to input data from or output data to the sub-systems, respectively. This method makes it easy to code SBC software since I/O devices can be treated as memory

[32] HiTech Equipment Corp., San Diego, CA.

variables in the program source code. Moreover, this method provides for easy addition of sub-systems: an available range of memory addresses is simply allocated for the new sub-system to use, without need for concern about addressing conflicts. Figure 4-24 shows how the SBC's subsystems are connected to the accelerator for data acquisition and control.

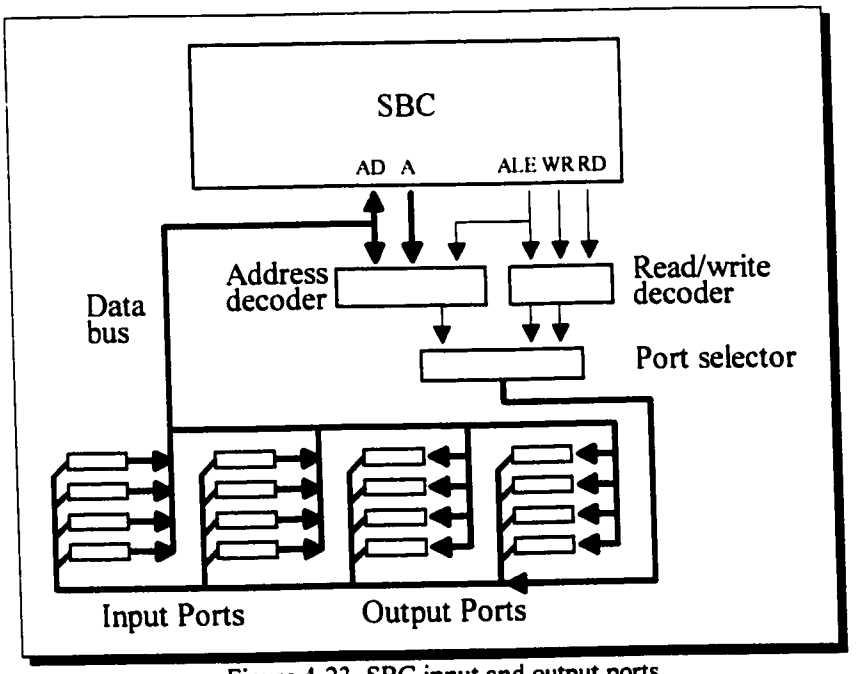


Figure 4-23. SBC input and output ports.

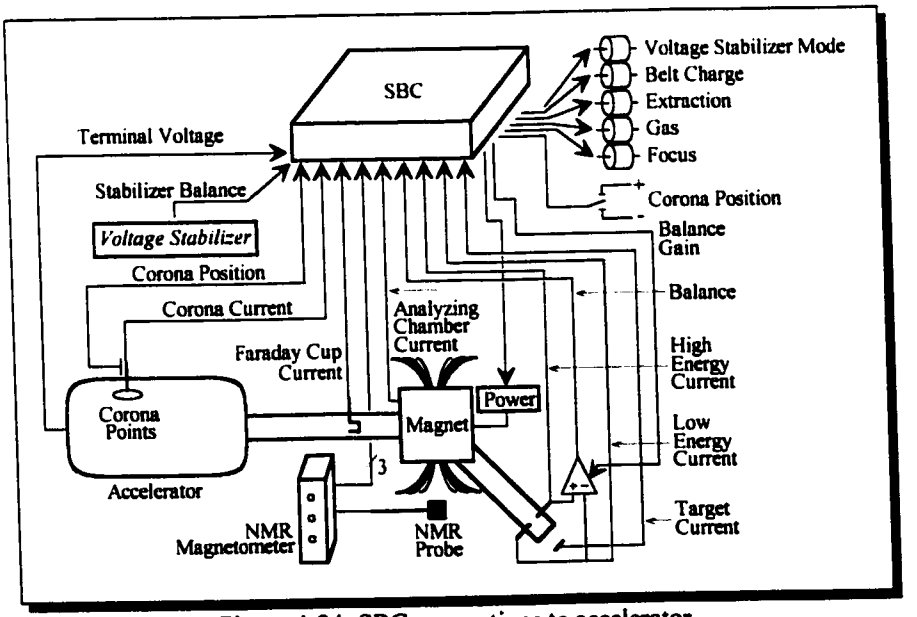


Figure 4-24. SBC connections to accelerator.

4.2.5 Real-Time Kernel

The RTK running on the SBC (Figure 4-25) is written in C and assembler code. It is responsible for receiving host commands, interpreting them and performing the requested actions. The kernel is also charged with several 'housekeeping' tasks, such as data acquisition and filtering, and stepper motor control. The RTK is held in object code format on the PC and downloaded into the SBC's RAM during system initialization.[33]

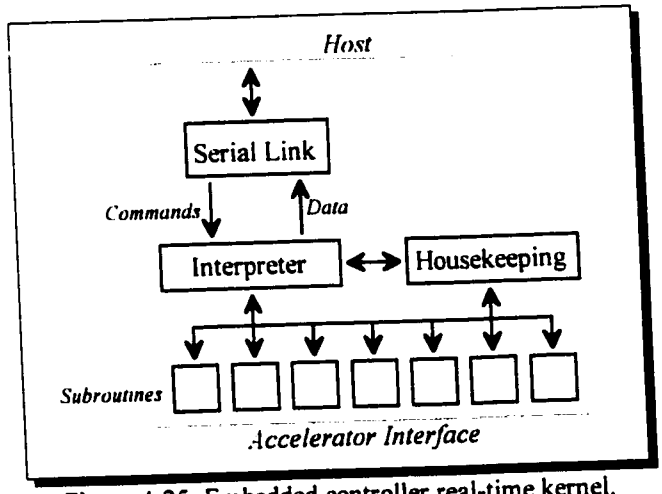


Figure 4-25. Embedded controller real-time kernel.

The main execution loop of the RTK is shown in Figure 4-26. The RTK loops continuously, polling the PC serial link for incoming commands (①). These commands resemble remote procedure calls (RPCs), consisting of an opcode (remote procedure ID) and zero or more parameters. If no RPC has been received, the RTK performs its housekeeping tasks before polling the serial link again (②).

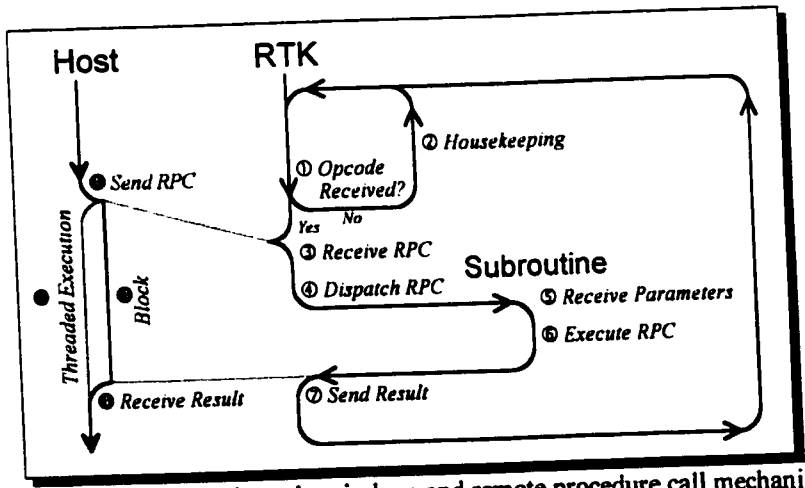


Figure 4-26. Real-time kernel main loop and remote procedure call mechanism.

[33] Later versions of the SBC will have the RTK placed in program ROM so that downloading is not necessary.

4.2.5.1 Real-time Kernel Housekeeping Tasks

There are several housekeeping tasks performed by the RTK. In the 'data acquisition' task, the RTK samples its digitized inputs from the accelerator, computes first-order differences and accumulates running averages. These data are transmitted to the PC in the form of a telemetry packet (see below).

For purposes of motor position feedback to the PC, motor turn commands are carried out as a sequence of 32-step stages. In the 'motor compliance checking' housekeeping task, the RTK determines whether a motor has completed its most recent 32-step turn command. When this happens, the RTK updates its internal record of the motor's position and starts the motor turning again if more stages are required to complete the turn command.

Corona points positioning adjustments are also carried out in a sequence of steps as a safeguard measure to prevent the points from being extended or retracted too far. Any active corona points position adjustment is automatically terminated if the RTK does not receive confirmation from the host PC within a certain period of time. This prevents the occurrence of a run-away condition in the event that the host PC crashes during corona points adjustment (cf. Section 4.3.4.3). Likewise, adjustments to the analyzing magnet set-point are performed in a stepwise fashion to prevent run-away and accommodate for hysteresis and slew rate in the analyzing magnet's power supply.

Another important task carried out by the RTK is the failsafe *command loss watchdog* mechanism, which automatically initiates accelerator safing action if a preset period of time passes without reception of a command from the host PC. This mechanism safeguards against the possibility of PACES crashing while the accelerator is running unattended and leaving the accelerator in an uncontrolled state.[34]

4.2.5.2 Real-time Kernel Remote Procedure Call Mechanism

As shown previously in Figure 4-26, when the PC sends an RPC (❶), the RTK receives an RPC opcode (③), housekeeping is suspended, and the appropriate subroutine is called (④). The subroutine in turn receives its expected parameters from the serial link

[34] The embedded controller's CPU also has a built-in watchdog timer which resets (reboots) the CPU if the software being executed crashes or becomes stuck in an endless loop.

(⑤), performs its function (⑥), and perhaps sends a result back to the PC (⑦). Meanwhile, the PC may either continue execution using parallel code threads (❹) or block (❸) until the result is received (❹).

One of the most frequently used RPCs is responsible for uploading accelerator parametric data to the PC. When the PC sends a telemetry request RPC, the RTK forms a telemetry packet from the information shown in Table 4-1. The PC requests telemetry packets from the SBC every 200ms, using a Windows event timer as the timebase. When the timer expires, a telemetry request RPC is sent to the SBC, and PACES blocks until the packet is received completely or a time-out occurs. In the event of a time-out, any partial packet received is discarded, another telemetry request RPC is sent, and a telemetry failure event is counted; after a certain number of telemetry failures, the SBC is rebooted. If a complete telemetry packet is received within the time-out period, its leader and trailer fields are validated. These fields should contain specific values, and the packet is discarded if they are incorrect, or if the exclusive-OR checksum is incorrect. Once the packet is validated, it is unpacked and its fields are used to update the selsyn positions and meter values. If data logging is active, the packet is time-stamped and written to disk. If a remote console is currently logged in, the packet is also forwarded along a separate serial link to the remote console.

Packet Header
Status Flags
Error Flags
Active Motor Flags
Idle Count
Selsyn Positions (4)
Switch Positions (16)
Averaged Meter Readings (16)
Digital Inputs (16)
Exclusive-OR checksum
Packet Trailer

Table 4-1. Telemetry packet.

The PC is able to program the RTK to monitor accelerator performance data and notify the host via a *service request* if certain operating parameters exceed prescribed limits (termed a *breach*). As the SBC is repeatedly sampling the analog signals from the control panel, it is able to compare each value to a pair of limits. The SBC can be instructed to watch for any combination of the following types excessive values:

- *Time-averaged*: To avoid noisy data and unimportant transients, this type is used to detect lasting deviations.
- *Integrated*: The SBC watches for an excessive integrated measurement (i.e. total measured value over a specific time interval).
- *Instantaneous*: Certain types of faults (e.g. sparks) are characterized by sharp, momentary changes that exceed the limits.
- *Differential*: The SBC monitors the rate of change of a certain value.

Upon detection of a breach, the SBC immediately notifies the PC, indicating the exact nature of the event. In response to the service request, the PC shifts its chain of reasoning to determine the nature of the breach, and perhaps initiate fault detection, diagnosis and recovery. This facility is analogous to a thermostat in a house that is set by an 'expert' (the human) to operate the furnace within a certain temperature range, and generating an event (turning the furnace on of off) when the temperature drops below, or rises above this range, respectively.

It is sometimes necessary for the PC to be able to gain control of the SBC, mainly during PACES initialization when the program needs to be sure that the RTK is functioning properly. To accomplish this, the serial link's DTR line (data transmit ready signal) is used as a reset strobe. The PC can pulse this line high to reset the SBC. When a reset occurs, the SBC responds by starting execution from a fixed program address, and the RTK reboots. Since resetting the SBC causes some of the SBC's CPU registers to lose their data, the PC will only reset the SBC if necessary. The PC first 'pings' the SBC by sending it a 'ping' RPC and a random 16-bit value. If the RTK is functioning properly, it will respond to the 'ping' command by returning the ones-complement of the random value received from the PC. If the PC does not receive the expected value, or a time-out occurs, the PC assumes the RTK is not running, and resets the SBC. Once the reset occurs, the PC again pings the SBC. If the ping again fails, the PC assumes that the SBC does not have valid copy of the RTK, and forces the RTK to be downloaded onto the SBC and started. Such downloading is typically only necessary when the SBC is powered up or when the RTK has changed and the new version needs to be placed on the SBC.

◆ ◆ ◆

This section has presented the 'low level' of PACES, consisting of the accelerator interface and embedded processor. In the following section, the use of knowledge-based reasoning in PACES is explored, illustrating how artificial intelligence reasoning techniques can be used to perform 'high level' decision making.

4.3 Aspects of Knowledge-based Reasoning

Knowledge-based reasoning forms the 'brain' of PACES, that part of the program responsible for making decisions about how to control the accelerator. There are two levels of decision making in PACES. The 'high level' decision maker is the expert system running on the PC. The 'low level' decision maker takes the form of algorithmic and fuzzy control systems running on the embedded controller. The expert system presides over the fuzzy controllers, deciding when they should be used, and how they should be configured.

4.3.1 Decision-making Requirements

Four key areas of accelerator control rely on knowledge-based inferencing. As described below, these areas pose different requirements on the decision-making system, and require that this system has capability for at least three types of 'reasoning': the heuristic forward chaining and backward chaining methods, and the deterministic, algorithmic (temporal) method.

Start-up: During start-up, the decision-making system is used to start the accelerator and produce a stable particle beam with specified physical characteristics. The established start-up procedure followed by the operators is used as a template for rule-based inferencing, implying that the decision-making process is algorithmic. Yet, the start-up procedure also requires a degree of heuristic decision making. Consider an abridged version of the procedure used to start-up the accelerator from a 'cold' state, shown in Table 4-2.

1. Turn on control power.	9. Pull out Faraday cup. (Beam now passes through analyzing magnet to target.)
2. Insert Faraday cup.	
3. Set gas and focus selsyns to previous settings.	10. Look for target current.
4. Increase terminal voltage to reach desired energy.	11. Monitor high/low slits, balance and annular ring current. If annular ring shows current, beam is out of focus. If balance signal is non-zero, adjust terminal voltage to shift balance toward zero. As balance reaches zero, increase balance gain to measure smaller balance errors.
5. Wait for "strike" (current on Faraday cup).	
6. Adjust extraction and gas selsyns to maximize beam current on Faraday cup.	12. When beam is stabilized, switch to built-in stabilization.
7. Set magnet field strength using NMR.	
8. Set balance gain to minimum.	

Table 4-2. General steps involved in accelerator start-up.

Despite the fact that this procedure appears at first glance to be quite deterministic, it is important to note that this algorithm excludes almost all of the 'expert knowledge' that the operator draws on while executing the procedure; the operator is monitoring the control panel at every step, intent on detecting any problems as they arise. At any stage, the operator may have to alter the procedure using some heuristic. To this end, the above algorithm captures only the *skeleton* of the start-up procedure, and the decision-making system must be able to rely on a large pool of knowledge in order to complete the start-up procedure successfully with an acceptable level of reliability. The shut-down procedure is similarly confounded, but to a lesser degree.

Shut-down: Shut-down involves the accelerator being deactivated in an expedient manner. As described in Section 4.3.2.8 (p. 41), the decision-making system may be called upon to perform any of the following: ① Normal shut-down; ② Rapid safing (in order to recover from a possibly dangerous anomaly); or ③ Panic shut-down (in event of a catastrophic fault). Here again, the established shut-down procedure followed by the operators is used as a skeleton template for knowledge-based inferencing.

Beam Maintenance: A more complicated duty of the decision-making system is that of *beam maintenance* during the period after start-up and before shut-down, when the accelerator must remain in some 'steady state' (that is, maintain beam stability on target and maintain certain accelerator operating parameters within specific tolerance margins). This mode of automated operation is called *cruise control mode*.

The accelerator is a tool used for conducting scientific experiments, and, as such, must be reliable, accurate and stable in its behaviour. Therefore, beam maintenance requires that the particle beam be maintained within specified operating limits. The conditions under which the accelerator is to produce a particle beam vary from experiment to experiment and from run to run. In a given experiment, however, it may be necessary to maintain the beam current with a maximum deviation of only a few nano-amperes, or perhaps keep the beam aimed at a specific location in the target area to within a few millimetres. At other times, the operator may need to draw on expert knowledge several different times during a run to alter particle beam characteristics. This implies that the concept of beam stability is ill-defined and transient, and therefore difficult to incorporate into a conventional knowledge-based system. Nevertheless, the decision-making system is charged with maximizing beam stability while affording the operator some capability to alter the accelerator's behaviour as required.

The approach being taken is to partition the control problem in such a way that those aspects that can be handled without much 'higher level' reasoning are isolated from the high-level expert system and attended to at a lower level. The SBCs (and their fuzzy-logic controllers[35]) are responsible for what amounts to stupid, narrow-minded fine-tuning. This low level of the system can be 'programmed' by the higher level expert system to consider the 'stable state' to be defined by a certain set of parameters, as determined by the required beam characteristics. The expert system determines the acceptable range for these parameters and transmits them to the embedded controller(s) where they are interpreted as the (new) definition of a 'stable' beam.

Fault Detection, Diagnosis and Recovery: The decision-making system is used to detect, diagnose, and perhaps recover from accelerator faults. Detection involves heuristics to determine that the accelerator is exhibiting a fault. Diagnosis involves deciding, based on accelerator operating parameters, what type of fault or aberration is occurring (or has occurred). Recovery involves selection of appropriate means to overcome the problem, be it a simple control adjustment, immediate shut-down, or request for operator intervention.

---

[35] See Section 4.3.3.

Perhaps the most difficult and important aspect of the entire expert system is its ability to detect on-line faults dynamically from the accelerator's behaviour and respond to them in a timely, effective fashion. A substantial portion of the knowledge base needs to deal with detection of faults based on telemetric performance data uploaded from the embedded controllers. Moreover, the knowledge base must include a diagnostic facility that the user can invoke when the accelerator is perceived by the operator to be functioning anomalously. That is, as the non-expert user's expertise increases, the user becomes more intuitively attuned to the quirks and peculiarities of the accelerator and may wish to diagnose anomalies that do not necessarily constitute faults and, therefore, are not directly detectable by the knowledge-based system. This facility represents a compromise between the difficult-to-attain "ideal" knowledge base and an adequate but incomplete knowledge base; the expert system will evolve over time to match the abilities and expertise of seasoned operators so as to provide an overall benefit for the operator community.

Fault diagnosis involves periodic analysis of logged accelerator performance data as well as relying on the SBCs' ability to detect breaches of operating limits quickly and notify the PC. The former, analysis of logged data, is performed to detect gradually developing (chronic) and/or intermittent anomalies (e.g. loss of beam line vacuum due to leakage), and requires both a large expenditure of computation and the utilization of the expert system's knowledge base. The latter, response to limit breaches, also involves use of the knowledge base to make rapid assessments of abnormal operating data, detect acute anomalies (e.g. sparks) and effect immediate remedial action. A related issue is that of periodic verification of the accelerator's safety interlock network: violations of safety interlocks must result in immediate 'safing' of the accelerator to prevent machine damage or personnel injury.

4.3.2 Expert System Considerations

The expert system is the highest level of decision-making capability in PACES. The following sections describe the requirements for, and design and implementation of the expert system. As will be explained, the requirements of PACES' decision-making

capability lead to a specialized and distributed hybrid knowledge-based system, which functions in a manner analogous to the reflex arcs described in Section 4.2.1.

4.3.2.1 Expert System Requirements

Given the decision making requirements outlined above, the 'expert system' component of PACES is, consequently, required to possess several important characteristics. First, the accelerator control system requires a tiered organization (cf. Figure 4-4) in which the system co-ordinator oversees the inference engine, deciding when knowledge-based reasoning should be employed and how it should proceed. Thus, the expert system must not be a stand-alone top-level shell, but rather be *embedded* in the control program as a whole. Due to the real-time nature of the control system, it is necessary to relegate the expert system to a subservient level; the expert system is invoked by the system manager when needed, but never assumes complete control. The expert system is periodically called upon to detect operating faults or anomalies and to perform heuristic decision making.

Second, the expert system must provide both forward and backward chaining. During start-up and shut-down, data-driven forward chaining is utilized to achieve the goal of a stable particle beam or deactivated accelerator, respectively. Fault diagnosis, conversely, requires goal-driven backward chaining to reach diagnostic conclusions based on accelerator performance data. Finally, beam maintenance mode operation involves both forward and backward chaining, depending on how the accelerator is performing and how the operator wishes to alter the accelerator's behaviour. Clearly, PACES must be able to switch freely between goal- and data-driven inferencing as events warrant. This requirement leads to the need for a flexible inference engine that allows for external control.

Finally, the expert system must be amenable to 'stop and start' inferencing. The real-time nature of the control problem requires that the inference engine be able to suspend its current reasoning path in order to follow some other reasoning path as events warrant. Typically, this situation arises when the operator decides to alter the accelerator's

state, which could happen at any time during the expert system's decision-making process, or when an anomaly occurs in the accelerator's behaviour.

4.3.2.2 Early Attempts at Choosing a Shell

Two early attempts were made at integrating an off-the-shelf expert system shell with PACES, and both met with disappointing results. The Personal Consultant Plus (PC+)[36] expert system shell was considered first, but was found to be almost impossible to embed into a larger application, ([DeM91], [Lin91]). Its slow inferencing speed, high demand on system resources and complicated external language interface mechanism made it unwieldy as an 'on line' real-time embedded inference engine. Kappa-PC[37] was also briefly evaluated since it offers many features that PC+ lacks. Kappa-PC is designed as a Microsoft Windows-based extensible expert system shell. Although Kappa-PC was, in many ways, ideal for the requirements at hand ([Lin91], [Lin92b]), it was initially only extensible using Microsoft C, which was found to possess a somewhat hostile user interface aversive to rapid software prototyping. A desirable form of Kappa-PC would be that of a Windows-format dynamic link library (DLL) that could easily be integrated into the control system regardless of the programming language used for development.[38]

4.3.2.3 Inference Engine Design

It was finally decided that the best approach was to use a custom-made expert system shell that would be designed from the outset to possess the desired qualities, ([Lin92a], [Lin92b]). This shell, the *Windows Application eXpert system* (WAX) shell, is designed to be embedded in a larger software package, and provides several important features:

- *Object-oriented programming*: Knowledge base objects (such as inputs, outputs and intermediate conclusions) can be linked to program objects (such as graphical elements displayed on the user-interface, or control/sense points that are directly connected to external devices).
- *Global information registry*: A blackboard-like repository for shared knowledge is available, and knowledge base objects can be tied to this information registry as producers or consumers, facilitating real-time information flow into and out of the knowledge base.
- *Two-way inferencing*: The knowledge base can be inferenced using forward- or backward-chaining as desired. This ability is of great utility during accelerator

[36] Texas Instruments, Austin, TX.
[37] Intellicorp, Mountain View, CA.
[38] The newest version of Kappa-PC does indeed possess this capability, but became available too late in PACES development to be used.

beam maintenance control, when forward chaining is used while the beam is in a quasi-stable state and backward chaining is used during fault recovery or beam alteration directed by the operator.

- *Demand-based inferencing*: The inference engine and knowledge base are re-entrant to allow for on-demand re-focusing of the inferencing process. This is useful for handling situations such as, for example, when a fault occurs during the start-up procedure. The inferencing process used for start-up must be suspended (temporarily or permanently) while the fault is diagnosed and corrected.
- *Source-code level compatibility*: The knowledge base structure is completely compatible with the implementation language (BPW – Borland Pascal for Windows), so that the knowledge base can be combined with the rest of the software package at a source code level.

The WAX shell is written as a collection of objects in a BPW *unit* (separately compiled module). Figure 4-27 shows how WAX is integrated into the application as a whole. The WAX unit does not contain any 'knowledge', but only the mechanisms for manipulating (inferencing) knowledge. The 'knowledge bases' are stored in separate units, and dynamically linked to the inference engine at run-time.

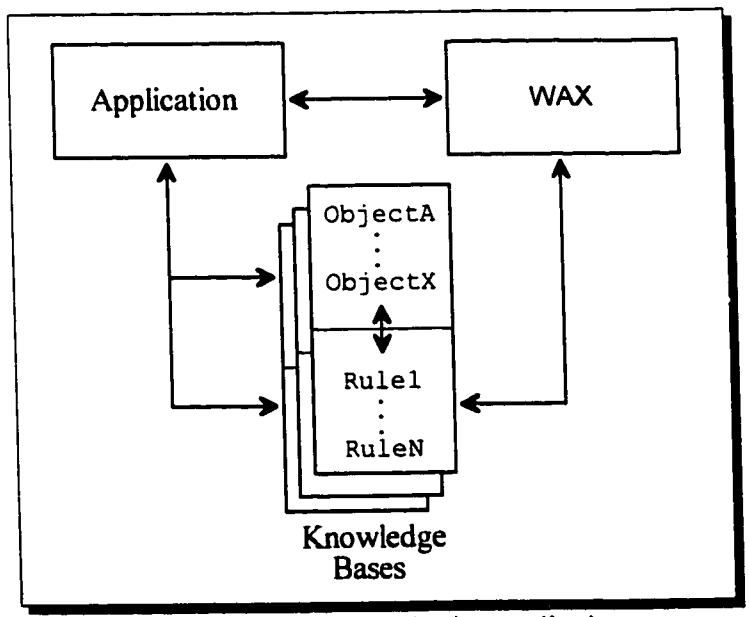


Figure 4-27. WAX integration into application.

The principal WAX object is the *inference engine* used to evaluate rules. Rules are passed to the inference engine in prioritized rule sets that represent domains of knowledge. The inference engine is re-entrant so that it can be interrupted while inferencing one rule set in order to inference a different rule set. Rules are able to change the course of

inferencing by calling WAX subroutines that start, suspend or stop inferencing chains, or alter the membership of rule sets.

WAX knowledge base units, which achieve the useful software engineering property of *information hiding*, contain *rules* (heuristics) and *information objects* (object-oriented data structures). In developing knowledge objects, the knowledge engineer (programmer) is free to use as much of BPW's extensible OOP capability as desired. The knowledge base rules are written by the knowledge engineer to process the information objects to derive new information objects or cause side effects in the application. (Examples of side effects include a pop-up message prompt asking the user to input a piece of information, or the assertion of a control action which causes actuation of an external device.)

WAX knowledge bases are inferenced in parallel using a multi-threaded execution object called the *thread sequencer*. Each knowledge base consists of one or more rule sequences (*threads*), and a collection of data elements (the 'knowledge'). The data elements are privately-stored data structures which the rules can access. Since data elements are global to all rules within a knowledge base, they form a type of 'blackboard' or data exchange local to the knowledge base.

Knowledge base rules are implemented as functions that return pointers to rules. When a rule is inferenced (executed), it decides how inferencing should continue, and returns a pointer to the next rule to be inferenced (possibly itself). In this way, the interconnected structure of the rule set provides for sequential or multi-branched inferencing, with rules being able to activate or deactivate rule threads dynamically.

Typically, the top level of the application initiates inferencing in response to a request from the user, or after some external condition has occurred (such as a timed event or interrupt). The application dynamically creates an inference engine object and passes it a rule set, thereby starting the inferencing mechanism. The inference engine evaluates the rules in its rule set until the rule set is empty or the engine is suspended or stopped. Inferencing proceeds under the Windows multitasking environment so that the inferencing can be performed 'in parallel' with other operations.[39]

---

[39] The version Windows used (3.1) performs non-preemptive round-robin multitasking.

The WAX implementation results in a flexible inferencing system that places a large burden on the programmer/knowledge engineer with regard to implementing the knowledge base. In certain situations, this approach is unsuitable since it requires that the knowledge engineer be a proficient programmer. Nevertheless, WAX is intended primarily as an embeddable inferencing system, and thus, is inherently tied with software development and programming. WAX is not meant to replace conventional shells that offer ease of knowledge encapsulation without requiring a high level of programming ability on the part of the knowledge engineer. Instead, WAX is designed to sacrifice generic ease-of-use and high shell overhead in favour of an extreme amount of low level flexibility and minimal shell overhead.

4.3.2.4 WAX Knowledge Base Structure

PACES uses several WAX knowledge bases for different stages of operation. These knowledge bases are implemented as descendants of WAX's generic knowledge base object called `TKnowledgeBase`, shown in Table 4-3.

```
TYPE  PSequencer = ^TSequencer;
      PThreadList = ^TThreadList;
      PRule = ^TRule;
      TKnowledgeBase = object(TObject)
                         sequencer: PSequencer;
                         threads: PThreadList;

                         constructor Init (ASequencer: PSequencer);
                         destructor  Done; virtual;

                         function  RuleSet (Rule: PRule; Thread: integer;
                                            Action: kbActionTyp) : PRule; virtual;
                       end;
```

Table 4-3. The `TKnowledgeBase` object.

The `sequencer` field of `TKnowledgeBase` points to the application program's thread sequencer, which is responsible for execution of threaded code sequences. The field `threads` points to a list of execution threads which are rule sequences to be inferenced in parallel using a prioritized round-robin schedule. Threads can be created, blocked, delayed, pre-empted, suspended or terminated by making calls to the thread sequencer.

The `constructor Init` and `destructor Done` are necessary components of all objects, and are used for creation and destruction of the object. The parameter `ASequencer` passed to `Init` is a pointer to the thread sequencer. During execution of `Init`, the knowledge base registers its rule threads with the thread sequencer for future

sequencing. When thread sequencing is activated, the knowledge base's `RuleSet` method will be executed.

The `RuleSet` method of `TKnowledgeBase` contains no rules; it is overridden in descendant objects to implement actual rule sets. This method is passed three parameters: ① A pointer to the next rule to be executed (`Rule`); ② A thread identifier (`Thread`), that indicates which parallel thread is being inferenced when the rule is executed;[40] and ③ An action flag (`Action`), used to specifiy whether the rule set should initialize itself (`kbInit`), terminate (`kbTerminate`) or perform inferencing (`kbExecute`). The `RuleSet` method returns a pointer to the rule that should be executed the next time it is called. The actual rules are coded as subroutines nested inside the overridden `RuleSet` method.

Knowledge bases are descendant objects of `TKnowledgeBase` that may contain data elements in the object's `private` section. An example knowledge base is shown in pseudocode in Table 4-4. Although it contains nonsense rules, it serves to illustrate WAX knowledge base structure. There are two knowledge elements: `balanceGain` and `balanceError`, both pointers to variables that are monitored and updated by the control system. The rule set consists of three rules in a single thread. `Rule1` is executed first, and makes a decision based on `balanceGain` whether to execute `Rule2` or `Rule3` next. Rules `Rule2` and `Rule3` make similar decisions affecting knowledge base data elements and rule set inferencing sequence.

This approach to knowledge base implementation has several advantages. It is easy for the system developer to write rules and declare data elements because they are coded in the application's programming language instead of an expert system shell. Likewise, the overhead of execution time and storage space associated with a shell is eliminated. The rigidity of a shell is also eliminated, enabling the developer to use the full capabilities of the programming language. Conversely, such an implementation is virtually impenetrable to all but experienced programmers. There is also a lack of built-in high level inferencing techniques, such as certainty factors and decision tracing, common in commercial expert system shells.

---

[40] This information is generally used for rule tracing diagnostics (q.v. § 4.3.2.5).

```
TYPE  TExampleKB = object(TKnowledgeBase)
                    function  RuleSet (Rule: PRule; AThread: integer;
                                       Action: kbActionTyp) : PRule;
                   private
                    balanceGain,balanceError: float;
                   end;

Function TExampleKB.RuleSet (Rule: PRule; AThread: integer;
                             Action: kbActionTyp) : pointer;

    function Rule1 : PRule;
       if balanceGain > 0.1 then Rule1 := @Rule1 else Rule1 := @Rule2;

    function Rule2 : PRule;
        if balanceError < 0.0
           then balanceGain := balanceGain * 10.0;  Rule2 := @Rule1;
           else Rule2 := @Rule3;

    function Rule3 : PRule;
       if balanceGain = 0.0 then Rule3 := @Rule1 else Rule3 := @Rule2;

    Begin
     case Action
          of
           kbExececute: RuleSet := Rule;       { Execute rule pointed to by Rule }
           kbInit: threads^.AddThread(@Rule1);     { Initialize rule set thread }
           kbTerminate: ;                                  { Do nothing }
          end
    End;
```

Table 4-4. Example knowledge base.

4.3.2.5 Decision Explanation Facility

Although WAX does not directly implement a mechanism for performing explanation of decision making, it is straightforward to add this capability to WAX knowledge bases. The approach used in PACES employs *input/output streams* (I/O streams) for providing decision explanation. An I/O stream is a object-oriented input-output file that can be linked to a window on the program's GUI. PACES defines a special, globally accessible output stream which is tied to the *decisions explainer window* (cf. Section 4.1.4). The knowledge bases are able to access this output stream, and their rules can contain code to send text-based information along the stream to the decisions explainer window.[41] When this window is visible on the GUI, the information is viewable by the user. Figure 4-28 shows the decisions explainer window opened during accelerator start-up. In this figure, each sentence beginning with a bullet (•) is the explanation of a 'decision' made by a rule in the start-up knowledge base. Because of the highly flexible nature of this decision explanation mechanism, 'decisions' displayed in the window can relate any information that the knowledge base programmer deems appropriate.

[41] This textual output can optionally also be simultaneously output to a disk file for later, off-line analysis.

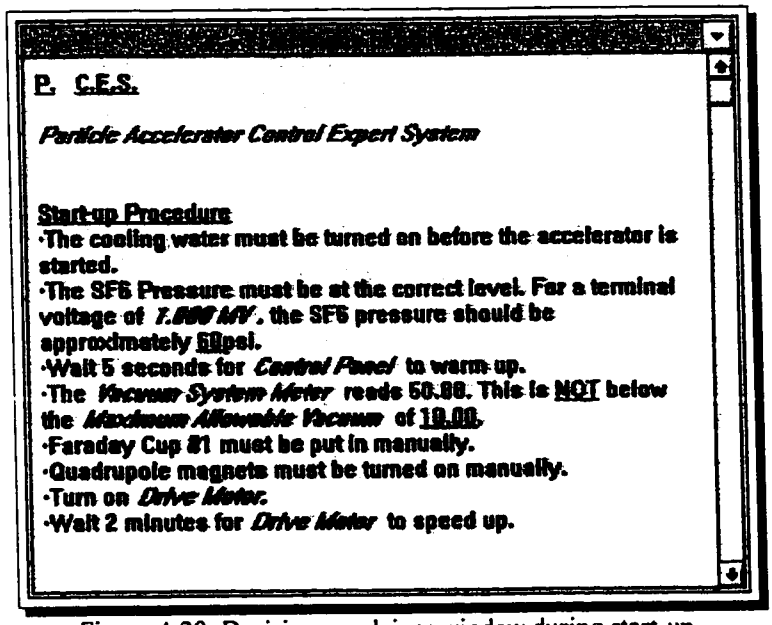


Figure 4-28. Decisions explainer window during start-up.

4.3.2.6 Algorithmic Control vs. Heuristic Decision Making

As already discussed, the control system as a whole is partitioned into several layers. The SBCs form the lowest level, possessing limited heuristic knowledge in the form of fuzzy-logic controllers, and a large amount of algorithmic 'knowledge' in the form of deterministic procedures. The SBCs respond to control and data acquisition commands, but have little volition of their own. At the top level, the host PC strikes a balance between complete heuristic activity and algorithmic activity. Algorithms are used for well-defined control procedures in order to increase efficiency and response time, whereas heuristics (inferencing) are used for expert-like decision making.

Certain aspects of the accelerator control problem are more suited to being implemented as algorithms. The shut-down process, for example, almost always follows the same, step-by-step procedure. Beam maintenance during an accelerator run, however, calls for a mix of heuristics and algorithms: control decisions are made using inferencing, but execution of the control decisions (e.g. feedback loops) requires algorithmic processing.

The WAX shell provides a simple means of melding procedural activity with heuristic reasoning: Since WAX knowledge base rules are merely sequences of embedded subroutines, they are unconstrained as to the scope and depth of their effects on the

application as a whole. A rule might, for example, initiate a completely algorithmic feedback control loop, succeeding only when the control loop meets some convergence criterion.

For example, as the expert system works its way through the start-up procedure, the stage is reached when the belt charge selsyn must be turned up until the required terminal voltage is exceeded slightly.[42] Although this could be implemented as a purely deterministic feedback loop, PACES instead incorporates inferencing into the control loop in an effort to detect any problems that might arise during start-up. This is analogous to operators who follow a general, well-established start-up procedure while simultaneously using their expertise to watch for unexpected problems (such as non-nominal terminal voltage slew rate and instability, poor vacuum, or improper SF_6 pressure). Hence, the algorithmic nature of the start-up procedure is augmented by diagnostic heuristics to produce a smart, flexible and robust automated start-up process.

4.3.2.7 Knowledge Engineering for PACES

The problem of controlling the accelerator presents many complications, ranging from real-time control considerations and controller robustness to educational and safety issues. Thus, the PACES knowledge base needs to encapsulate as much expert knowledge as possible, especially in the area of fault detection and diagnosis, while still preserving a high degree of real-time response.

Knowledge engineering for PACES was performed in conventional manner. Various expert accelerator operators were interviewed and observed over several sessions, notes were taken by hand and the interviews were taped using an audio cassette recorder. Following preliminary interviews, a prototype knowledge base was constructed that implemented the start-up and shut-down procedures. A development cycle ensued in which testing was followed by expert evaluation and knowledge base modification and further testing. The bulk of this development was performed while the knowledge engineer was off-site — that is, away from both the target accelerator and the essential experts — and, consequently, the development time was lengthy and complicated by numerous

[42] During the later stage when the particle beam is actually produced, the terminal voltage tends to drop, so the machine is initially set to a terminal voltage higher than necessary.

expensive (but inadequate) field trips to the accelerator site for testing and debugging.[43] Such off-site development of similar automated systems is not recommended.

Subsequent development involved knowledge engineering for the beam maintenance process. The majority of this work was performed on-site, in close consultation with accelerator operators and technicians.[44] Since these experts were readily available during this phase of knowledge engineering, and the accelerator was almost exclusively at the knowledge engineer's disposal, it was straightforward to iterate rapidly through consultation, coding, testing and debugging. This facilitated a quick development time for the beam maintenance knowledge base when compared with that of the start-up procedure's knowledge base.

4.3.2.8 Fault Detection and Diagnosis

The most important duty of the PACES expert system is the timely detection of faults followed by expedient remedial action. This fault detection and diagnosis relies on heuristic inferencing because there is a distinct lack of sensors that could be used to detect directly the many types and locations of faults. Consequently, rather than easily, repeatedly polling hundreds of sensors, the control system must rely on inferring anomalous behaviour from the available telemetric data (derived from the control panel's meters). This is just the way that skilled operators diagnose problems: they are drawn to abnormal readings on the control panel meters, reason about causes and then try to rectify the problem.

Accelerator faults are classified using a continuum of severity (Figure 4-29). Low severity faults cause minor problems (such as slight beam instability) that can be ignored by the control system unless they become more severe. During such low severity conditions, the expert system will limit operation of the accelerator in certain ways, such as imposing an upper bound on the permissible terminal voltage and/or beam current. (For example, poor vacuum would 'de-rate' the maximum terminal voltage to 1MV rather than the normal 3MV until the vacuum improves. This problem can arise after the accelerator has been opened to atmosphere for maintenance, and requires several hours after

[43] The accelerator used during this phase of development was located at DREO, several hundred kilometres from McMaster University.
[44] This phase of development was performed on-site at AECL's Whiteshell Laboratories, Pinawa, Manitoba.

re-sealing to achieve optimal vacuum; during this time the machine is still operable, but not at maximum rating.)

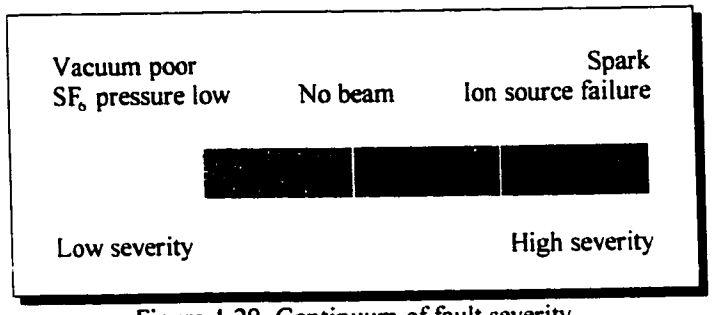


Figure 4-29. Continuum of fault severity.

High severity faults may require the control system to initiate safing action (cut off the particle beam) or complete shut-down, or may cause a hardwired safety interlock to trip. In the latter case, the control system must recognize the trip and, if appropriate, attempt recovery.

The fault continuum is dominated by a wide range of severity that represents faults that are too severe to ignore but not severe enough to trigger a shut-down. It is for this wide variety of cases that the expert system is utilized to analyze the anomalous symptoms and produce a diagnosis. Due to the large number of possible faults, and the low number of indicators, the expert system is charged with finding the most plausible explanation for the anomaly. As an example of this diagnosis, consider the problem of 'no beam' (Figure 4-30): The start-up procedure has progressed to the point at which, under normal circumstances, a beam current should be measured at the first Faraday cup, but no beam current is measured.[45]

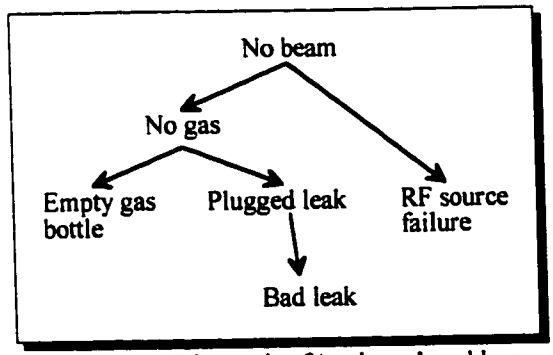


Figure 4-30. Diagnosis of 'no beam' problem.

[45] The 'no beam' problem can also arise during steady state beam maintenance when the beam is suddenly cut off.

Given the above description, the 'no beam' problem occurs when there is no measurable beam current on the Faraday cup even though the terminal voltage is correct and the gas and extraction selsyns have been adjusted properly (cf. § 2.2). The lack of beam is most likely caused by either failure of the RF source (due, perhaps to a failed electronic component such as a vacuum tube), or a lack of gas to be ionized. To begin the diagnosis, the expert system consults the knowledge base to determine when the gas bottle was last changed. If the gas bottle has not been changed for some specified time, the expert system can suggest that the bottle needs to be re-filled. If the gas bottle was recently changed, the lack of beam is probably not caused by an empty bottle.[46] Given that the bottle is not empty, the expert system can hypothesize that the problem probably lies in the thermo-mechanical leak mechanism. This hypothesis can be tested automatically: The gas selsyn is zeroed, the gas source is switched to the alternate gas bottle, and the gas selsyn is turned back up. If this test succeeds in producing a particle beam, the expert system can conclude that the original gas bottle's leak mechanism is faulty. If the test fails, the expert system can infer that there is an electrical problem shared by both thermo-mechanical leaks (e.g. failure of their shared power supply).

Although overly simplified, the 'no beam' example illustrates the extent to which PACES uses knowledge-based inferencing to diagnose accelerator faults and resolve conflicting symptoms. In many situations, the expert system can be engineered to perform explorative troubleshooting (such as switching gas bottles) to reduce its search space during fault diagnosis. In some cases, the expert system is able to overcome the problem and continue accelerator operation; other problems result in shut-down and the posting of a diagnostic message that the operator can use to effect repairs.

4.3.3 Knowledge Base for Accelerator Start-up

As mentioned in Section 4.1.1, the start-up procedure possesses a high degree of determinism in that it is essentially a well-established, step-by-step process. Typically, one instance of accelerator start-up differs from another only with respect to its goal parameters (e.g. terminal voltage, analyzing magnet setting, beam current on target) and the possible occurrence of unexpected complications during start-up (such as faults or

[46] This alternative is still possible (i.e. the gas was left on high for a long period and the bottle emptied prematurely), but is less likely.

failures). Were it the case that the goal parameters never changed, and the accelerator never experienced faults or failures, there would be little need for artificial intelligence-based reasoning during start-up.

Consequently, accelerator start-up is verging on straightforward, and this was the first aspect of automated accelerator operation to be implemented in PACES. Consultation with seasoned operators at DREO and McMaster, coupled with observation of their activities during start-up, resulted in a basic skeleton procedure for start-up (cf. Table 4-2). A similar skeleton procedure was later developed for the Whiteshell Labs accelerator.

Once the skeleton start-up procedure was determined, it was implemented as a WAX knowledge-base, whose flowchart is shown in Figure 4-31. The basic start-up process involves five basic stages (① to ⑤ in Figure 4-31):

Accelerator power-up (①): The accelerator's control power is switched on, the van de Graff generator's belt drive motor is started, and the belt charging system's power supply is activated. For safety's sake, this step always involves human intervention to apply power to the accelerator.

Terminal voltage set-point acquisition (②): The belt charge selsyn is increased to apply charge to the accelerator's high voltage terminal until the desired terminal voltage is achieved. The focus selsyn is adjusted to an approximate setting appropriate to the desired terminal voltage. If the accelerator tank has recently been been opened for maintenance, or due to other factors (such as tank internal moisture level or SF_6 pressure), the van de Graff generator may be unstable at high voltages, and may need to be *conditioned* (gradually brought up to voltage). Two other knowledge base threads may be called during this phase: the voltage set-point pilot (q.v. § 4.3.4.2)and the voltage conditioner (q.v. § 4.3.5).

Initial establishment of a stable particle beam (③): The source gas flow into the ionizing chamber is adjusted (using the gas selsyn) to provide sufficient gas for ionization into plasma. The extraction selsyn is then adjusted until a particle beam is detected on the first Faraday cup (a *strike*). The focus and extraction selsyns may then be adjusted to maximize the beam current.

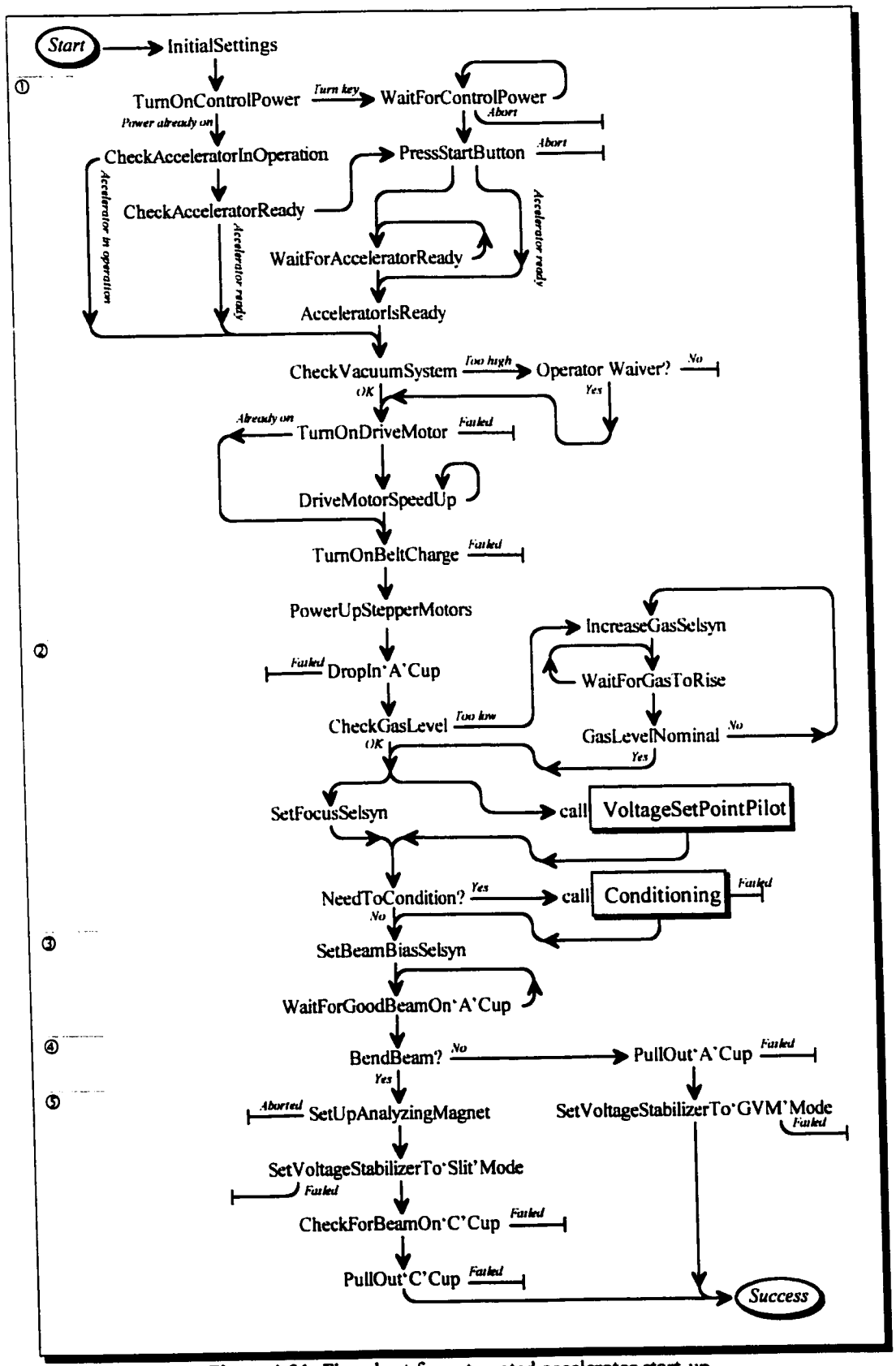


Figure 4-31. Flowchart for automated accelerator start-up.

Beam line set-up (④): The beam line is configured by setting up various beam steerers, quadrupoles, analyzing magnet(s), etc. This step also requires the voltage stabilizer to be activated in order to maintain beam energy finely; once the voltage stabilizer is functioning properly, PACES can relinquish control of terminal voltage until such time when the voltage stabilizer loses control (see Section 4.3.4). This stage of start-up involves a large amount of human intervention because current versions of PACES have little control over the accelerator beam line. At present, only the analyzing magnet is configurable and controllable by PACES, but human intervention is still required to apply power to the magnet.

Delivery of particle beam to target (⑤): The particle beam is now striking the last Faraday cup upstream of the target. The focus and extraction selsyns can be adjusted to optimize the beam current. Finally, all Faraday cups are pulled out of the beam line, and the particle beam is conveyed to the target.

The automated start-up procedure is considered successful if it reaches the end of Stage 5. When this occurs, inferencing of the start-up knowledge base terminates, and PACES switches to *cruise control mode* in which the *beam maintenance knowledge base* is inferenced.

4.3.4 Knowledge Base Subroutines

Several inferencing threads are implemented as 'subroutines' (subsidiary, re-entrant knowledge bases) which can be called from other threads during the course of inferencing. Typically, the inferencing thread performing the subroutine 'call' is blocked until the subroutine terminates. There are two main knowledge base subroutines: the *ripple loop* and the *voltage set-point pilot*.

4.3.4.1 Ripple Loop Knowledge Base

The ripple loop (Figure 4-32) is executed whenever a period of terminal voltage instability is detected (usually after a spark has occurred); it can be called from either the voltage set-point pilot (q.v. § 4.3.4.2) or the voltage conditioning knowledge base (q.v. § 4.3.5). The ripple loop returns a Boolean flag to its caller: `false` indicates there was little or no voltage instability; `true` indicates that sufficient voltage instability is

occurring that remedial control action should be performed by the caller (typically implying that the belt charge selsyn should be decreased slightly to reduce the terminal voltage, thereby reducing the instability).

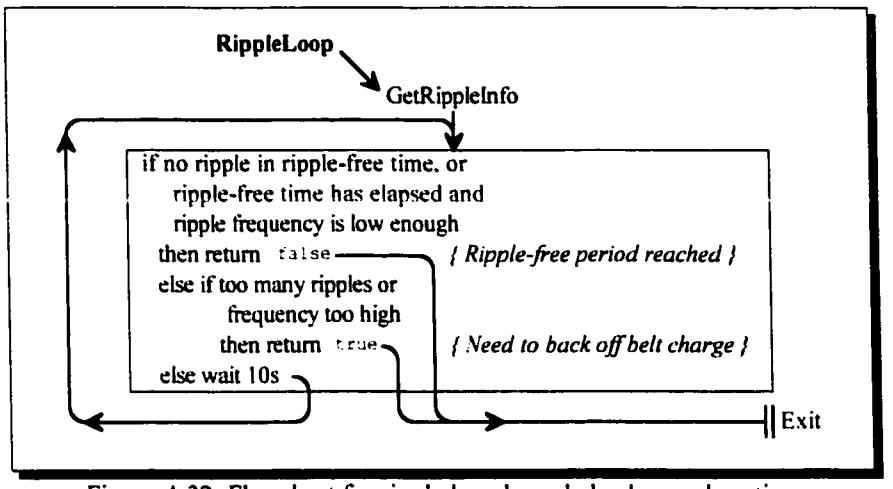


Figure 4-32. Flowchart for ripple loop knowledge base subroutine.

The ripple loop subroutine performs no control actions, but merely loops as long the terminal voltage remains unsteady, as determined by the *ripple signal* (first-order derivative of terminal voltage with respect to time). If no significant ripple is detected for 10s, or the ripple frequency is low (<0.01Hz), then `false` is returned, and the subroutine terminates. Otherwise, if too many ripples have been encountered (>200), or if the ripples are occurring too frequently (>0.9Hz), then `true` is returned, and the subroutine terminates. Otherwise, the ripple loop waits 10s and then reiterates.

The net result of the ripple loop is that brief periods of voltage instability are tolerated, but longer or more intense periods of instability are flagged for compensatory control action.

4.3.4.2 Voltage-Set-point Pilot Knowledge Base

The *voltage set-point pilot* (Figure 4-33) is a subsidiary knowledge base used for acquiring terminal voltage set-point. This knowledge base is called initially from the start-up knowledge base to set the terminal voltage, and subsequently during beam maintenance whenever the terminal voltage drifts significantly from its set-point.

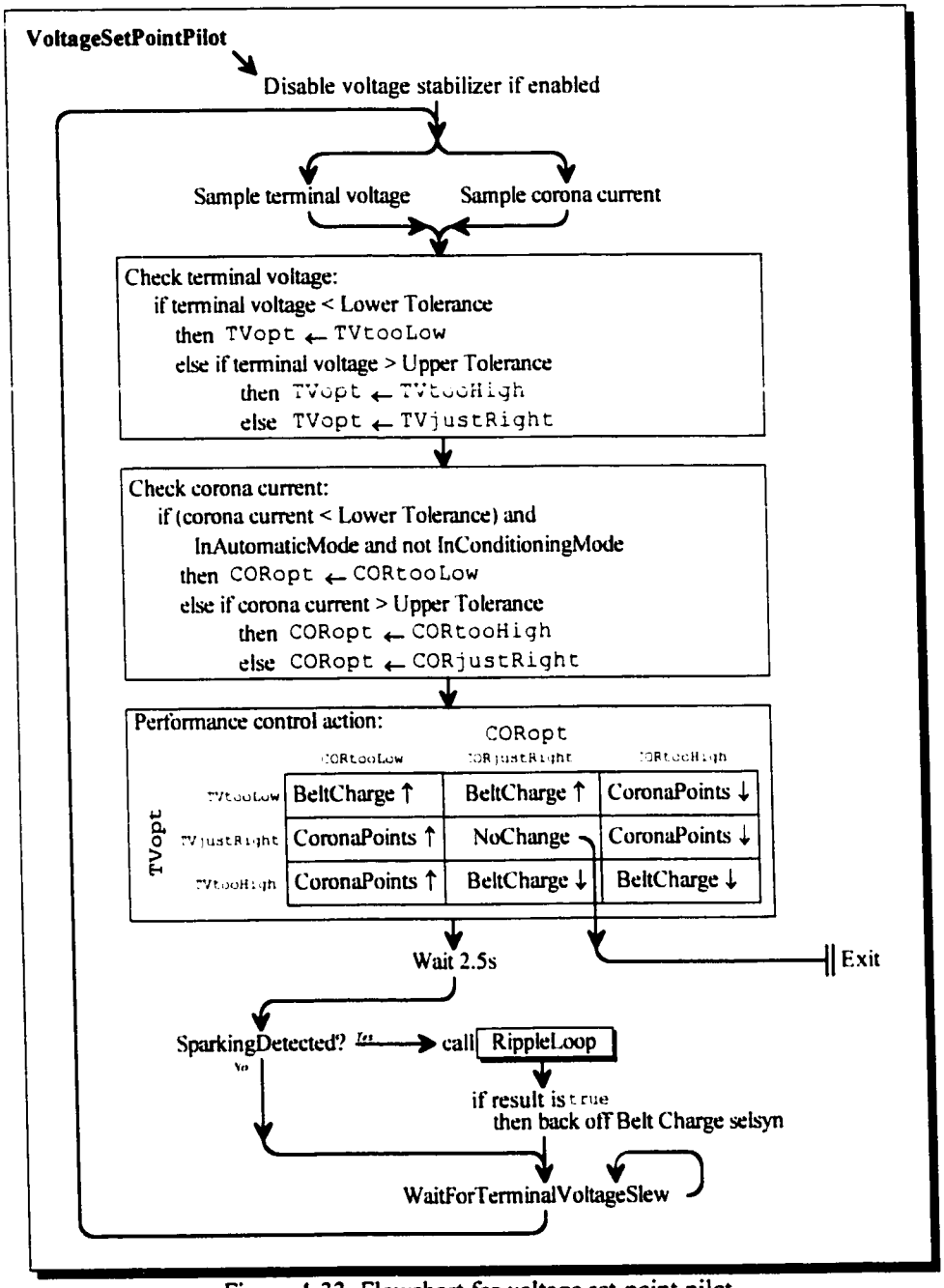


Figure 4-33. Flowchart for voltage set-point pilot.

The voltage set-point pilot samples the terminal voltage and corona current, and if the terminal voltage is out of tolerance, it is gradually brought to its set-point, either by adjusting the belt charge selsyn (which alters terminal voltage directly) or repositioning the corona points (thereby affecting corona discharge current, which affects terminal voltage indirectly). The knowledge base determines which form of adjustment is most suitable, depending on the values of the terminal voltage and corona current; if no control action is warranted, the subroutine terminates. Otherwise, after the appropriate control adjustment

has been performed, the knowledge base waits 2.5s for the terminal voltage to slew, and then checks to see if any sparking has occurred. If sparking has been detected, the ripple loop knowledge base subroutine is called; if this subroutine returns `true` then the belt charge selsyn is decreased slightly. The voltage set-point pilot next waits for the rate of change of the terminal voltage to reach near-zero (indicating that the terminal voltage has slewed completely), and then reiterates.

The net result of the voltage set-point pilot is that, upon termination, the terminal voltage will equal the specified set-point to within the specified margin of error. As a corollary, since the ripple loop subroutine is executed whenever sparking is detected, the voltage set-point pilot terminates with the terminal voltage stable.[47]

4.3.5 Knowledge Base for Voltage Conditioning

Voltage conditioning is performed to 'condition' the accelerator gradually to increasingly higher terminal voltages. This is typically necessary when operating at terminal voltages near the accelerator's maximum rating, after the machine's tank has been opened for maintenance, or whenever the accelerator exhibits periods of voltage instability. When performed manually, conditioning can consume as much as 24 or 48 hours of continuous accelerator operation. Consequently, automation of this aspect of accelerator operation is of great benefit to accelerator operators.

As shown in Figure 4-34, conditioning involves a general procedure for gradually increasing the belt charge selsyn to increase the terminal voltage whenever the terminal voltage is stable. A compensation strategy is also employed either to wait out the periods of instability which accompany increases in terminal voltage, or ultimately to reduce the belt charge if the voltage becomes unacceptably unstable. There are two variations of voltage conditioning: *warm* and *cold*. Cold conditioning is performed when the accelerator has been started from a 'cold' state, such as after a weekend of inactivity, or after maintenance. During cold conditioning, the machine is 'over-conditioned' to a terminal voltage significantly higher than the desired operating voltage so as to provide a region of stability around the desired operating voltage because stability at any given voltage implies

[47] This state of stability is, of course, not guaranteed to persist for any length of time after the voltage set-point pilot terminates.

better stability at lower voltages. (For example, if the accelerator is to be run at 4.0MV. cold conditioning would bring the voltage up to perhaps 4.2MV to provide a margin of stability.) Warm conditioning, in contrast, is performed during an ongoing accelerator run when the machine encounters a period of voltage instability; during warm conditioning there is no 'over-conditioning' to a higher terminal voltage. In either case, the conditioning operation proceeds until aborted by the operator, or until the target terminal voltage is reached and the voltage is stable for a significant period of time.

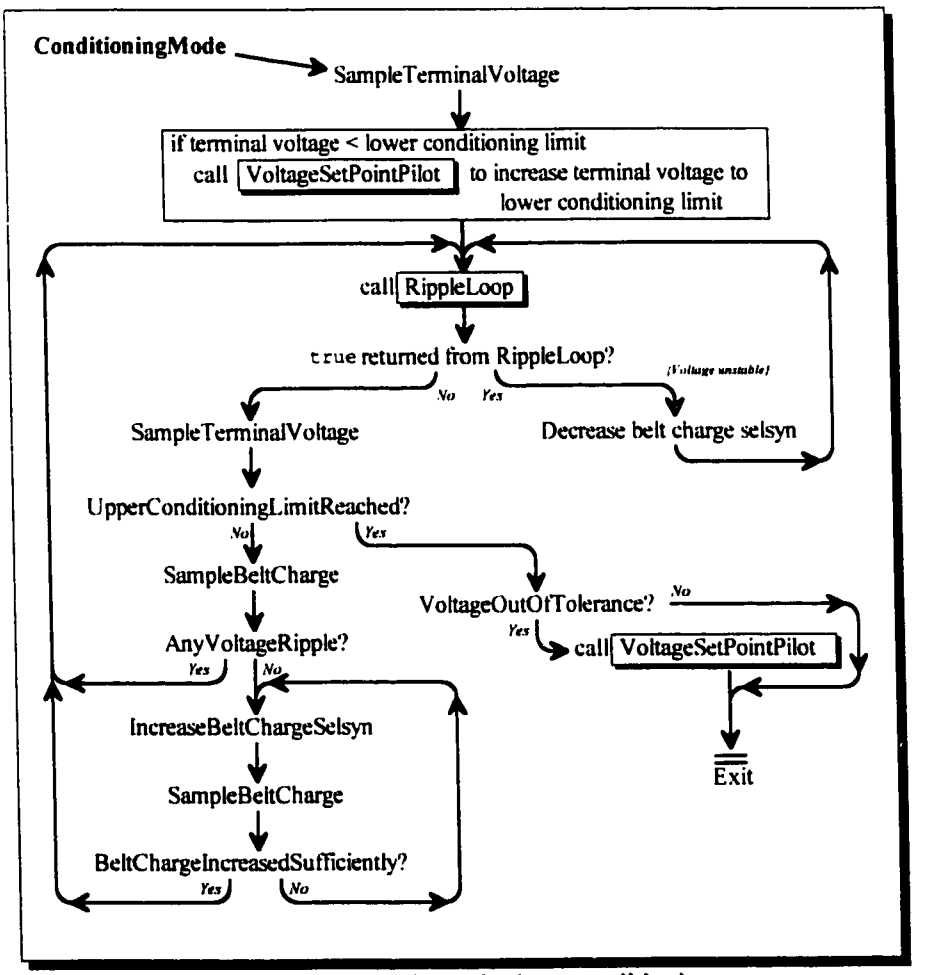


Figure 4-34. Flowchart of voltage conditioning.

4.3.6 Knowledge Base for Beam Maintenance

The *beam maintenance* phase of accelerator operation involves one major activity and several minor activities. The principal duty of the beam maintenance knowledge base is to maintain the particle beam on target within specified tolerance margins (beam energy and current). The minor activities include monitoring and possibly adjusting the corona

current, vacuum level, sample power, and beam current. The beam maintenance knowledge base consists of a set of parallel inferencing threads, one for each task to be performed. Each task can be separately configured and enabled (or disabled) by the operator (cf. Figure 4-11). Inferencing of this knowledge base is initiated immediately after the start-up knowledge base terminates successfully, or whenever the operator switches the accelerator into cruise control (automatic) mode.

Beam maintenance inferencing continues until one of the following occurs: ① The operator switches to manual control; ② The operator initiates automated shut-down; or ③ An unrecoverable operating fault (such as loss of beam on the first Faraday cup) is detected by the expert system. In the case of an unrecoverable operating fault, the expert system attempts to place the accelerator into a 'safe mode' (by, for example, switching off the belt charge to kill the beam) and calls for operator intervention, or performs automated accelerator shut-down if the accelerator is running unattended.

During beam maintenance, the particle beam may fluctuate in intensity and position due to source gas flow, fluctuations in the terminal voltage, and fluctuations in the field strength of the analyzing magnet. The most severe form of fluctuation is caused by a rapid drop in terminal voltage due to a discharge (spark) occurring inside the accelerator tank or accelerating column. Figure 4-35 captures the profile of a typical spark, illustrating how the precipitous drop in terminal voltage is usually followed by a rapid recovery to set-point. Sparks can occur at any time, but are usually more likely when operating at higher voltages, if the pressure of the SF_6 gas is too low, after the accelerator tank had been opened to atmosphere for maintenance, or if the accelerator has not been properly conditioned.

When a spark occurs while the voltage stabilizer is performing slit-based terminal voltage control, the voltage stabilizer usually loses control for several seconds, and sometimes may not be able to regain control even after the terminal voltage has recovered. If the voltage stabilizer loses control, the operator is obliged to switch the unit to Off, wait for the terminal voltage to recover, and then attempt to re-enable the voltage stabilizer. During this recovery period, the operator may need to re-acquire the terminal voltage

set-point manually, adjust the analyzing magnet set-point, or perhaps even adjust the gas and/or extraction selsyns to alter the stability of the particle beam, or to redirect the beam to target. Additionally, if the accelerator is exhibiting large instability in terminal voltage, and sparks repeatedly, the operator may be required to perform conditioning. In severe cases, the spark may even damage accelerator circuitry or components, requiring shut-down and maintenance.

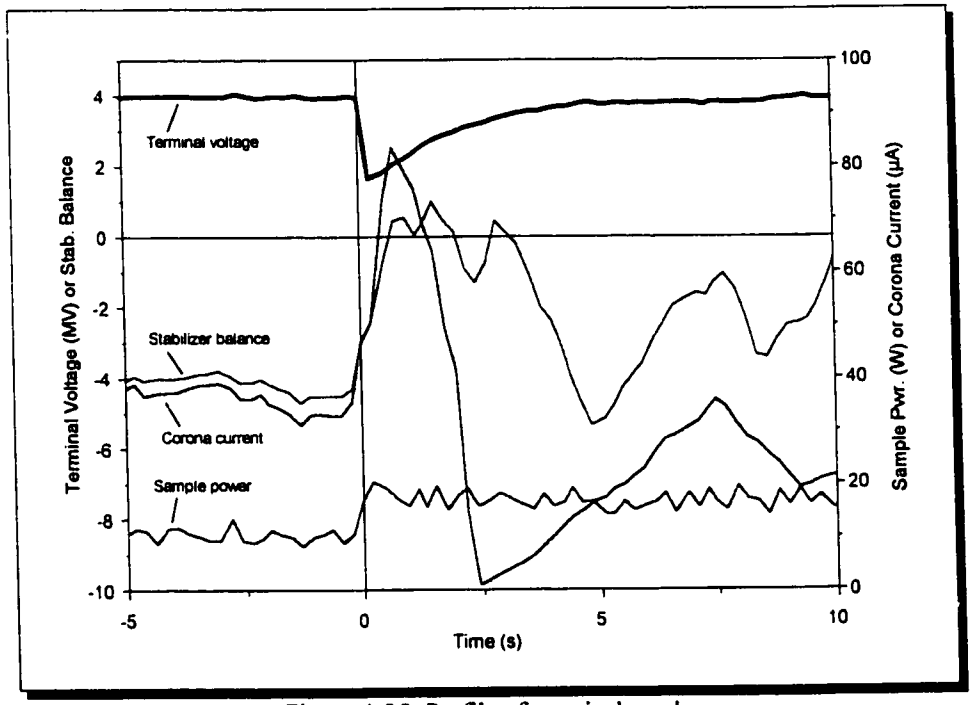


Figure 4-35. Profile of a typical spark.

4.3.6.1 'Auto-Pilot' Inferencing Thread

The *Auto-Pilot* is the main inferencing thread for beam maintenance, which can be divided into four major sections (colour-coded paths in Figure 4-36):

- *Preparation* (⟶): This phase prepares the accelerator for cruise control by: ① Ensuring the terminal voltage is stable and at set-point (possibly performing conditioning as well), ② Verifying that the particle beam is present at the first Faraday cup ('A' cup),[48] ③ Enabling the voltage stabilizer by switching it to Slit (or GVM, if the beam is not being bent), ④ Ensuring that the particle beam is striking the target (as implied by the sample power signal), and finally ⑤ Enabling the main thread's *helper threads* (described in the following sections).

[48] The first Faraday cup is located between the aperture and analyzing magnet, and is the first place where the beam current can be measured. If no beam is detected at this location, there is likely a fault in the accelerator (such as no source gas, or dead RF source), or the machine has not been properly configured to produce a particle beam. During automated operation under PACES, a loss of beam at the 'A' cup is considered a non-recoverable fault, and the program is forced to shutdown the accelerator and request manual intervention.

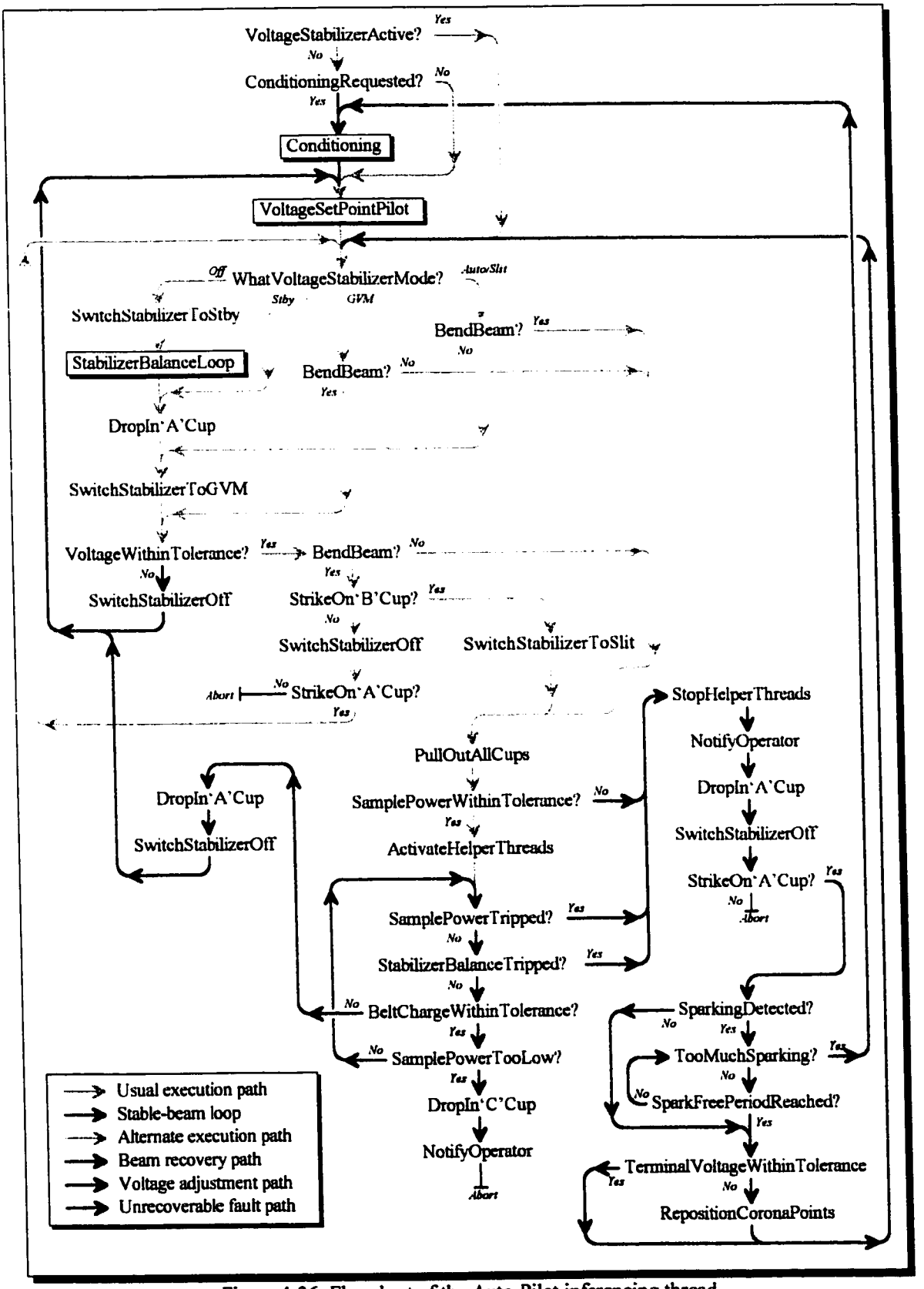


Figure 4-36. Flowchart of the Auto-Pilot inferencing thread.

- *Stable-beam loop* (⟶): This loop is entered after the *preparation* phase, and executes repeatedly as long as the beam is stable and on target. The sample power and stabilizer balance signals are continuously monitored. If the sample power signal jumps suddenly, or if the variance of the stabilizer balance signal drops to near zero, then the beam has been lost from target, and *recovery mode* is entered.
- *Recovery mode* (⟶): This stage initiates beam recovery by: ① Disabling the helper threads, ② Notifying the operator, ③ Disabling the voltage stabilizer, ④ Verifying that the particle beam is still present at the 'A' cup, ⑤ Waiting for any terminal voltage instability to subside (and possibly switching to the *voltage adjustment* phase), and finally ⑥ Looping back to the *preparation* phase to resume voltage stabilizer-based control.
- *Voltage adjustment* (⟶): Various paths in the main thread are used for adjusting the terminal voltage: ① Invoking the voltage set-point pilot if the terminal voltage if off set-point, ② Entering conditioning mode if too many sparks are occurring, ③ Performing fine terminal voltage adjustment during recovery mode by repositioning the corona points, and ④ Taking remedial action if the belt charge exceeds a preset limit.

Inferencing of the Auto-Pilot thread is sufficient to ensure that the beam remains on target during unattended operation, but by itself is unable to compensate for any 'drifting' of various accelerator parameters which may occur during a long accelerator run. Consequently, several *helper threads* are also employed, each responsible for monitoring a particular sub-system of the accelerator and making compensatory adjustments as needed. As mentioned previously, each helper thread is individually configurable by the operator and can be switched on/off as desired. The helper threads remain dormant until the Auto-Pilot enters its 'stable-beam loop', at which time the Auto-Pilot activates all helper threads that have been configured and enabled by the operator. The activated helper threads execute in parallel to the Auto-Pilot until the beam is lost and the Auto-Pilot enters 'recovery mode', at which time the helper threads are deactivated (and remain dormant until the beam is recovered).

4.3.6.2 Corona Current Optimization

During accelerator operation, electric charge conveyed to the high voltage terminal via the van de Graaff generator's belt accumulates to create the terminal voltage. This voltage gets reduced in three ways:[49] ① Lost as work performed to accelerate the beam's charged

[49] Of course, the terminal voltage always quickly becomes zero if the van de Graaff generator is stopped, or the belt charge power is cut.

particles, ② Lost through high voltage discharges (sparks), or ③ Removed through coronal discharge via the corona points.

Adjusting the belt charge selsyn increases the amount of charge being carried to the high voltage terminal per unit time. Varying the position of the corona points affects the amount of coronal discharge from the high voltage terminal per unit time. If the voltage stabilizer is inactive (and thus, fluctuations in terminal voltage are not compensated for by altering coronal discharge), any adjustments made to the belt charge selsyn or corona points position cause the terminal voltage to shift accordingly. Under purely manual control, the operator would typically use the belt charge selsyn to set the terminal voltage as desired, and position the corona points to attain a level of voltage stability with a moderate amount corona current.

When the voltage stabilizer is enabled, it accomplishes voltage stabilization by controlling the amount of coronal discharge (corona current) from the high voltage terminal: if the terminal voltage is below set-point, coronal discharge is reduced to increase terminal voltage; likewise, if the terminal voltage is above set-point, coronal discharge is increased to reduce the terminal voltage. If the baseline corona current is too low, there is insufficient margin to compensate for large drops in terminal voltage. Similarly, a large rise in terminal voltage is accommodated by bleeding off the excess as increased corona current, but a high corona current is undesirable. It is therefore desirable to maintain the corona current at a moderate level which is sufficient to afford compensation for voltage drops, but also low enough that compensation for voltage rises will not result in excessive corona current.

Figure 4-37 shows the corona current optimization thread, which is responsible for ensuring that the corona current remains within a prescribed tolerance interval. This thread executes in a loop, waiting for the corona current level to deviate from its tolerance interval.[50] When the corona current deviates from its tolerance interval, the thread waits several seconds to ensure the deviation is non-transient, and then checks for sparking. If sparking is detected, the thread waits several seconds before sampling the corona current again. If no recent sparking is detected, the thread performs a belt charge selsyn

[50] No control actions are performed if the voltage stabilizer is inactive.

adjustment to drive the corona current back into its tolerance interval and towards its optimum value.[51] Once optimum corona current is reached, the thread re-enters its waiting state.

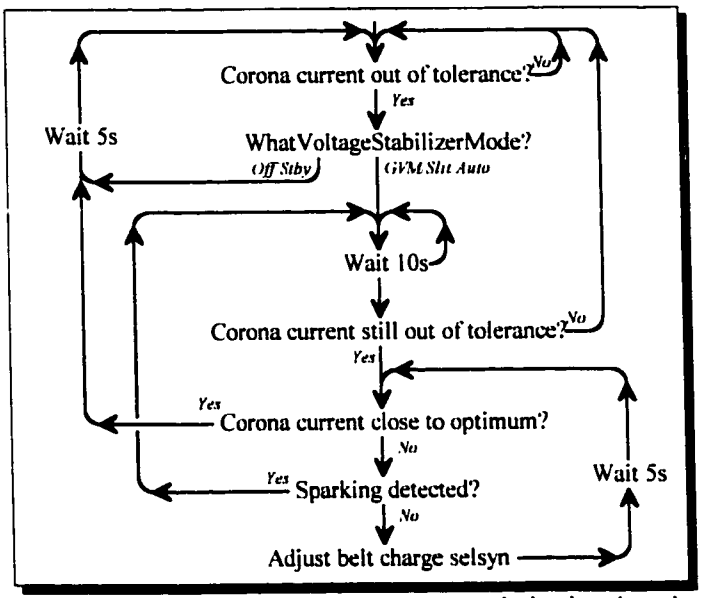


Figure 4-37. Flowchart of corona current optimization thread.

4.3.6.3 Beam Current Optimization

During an accelerator run, it is usually necessary to maintain the beam on target within some tolerance interval of beam current. The *beam current optimization* knowledge base thread is responsible for monitoring the target beam current to ensure that the particle beam is still striking the target and that the beam current is within some specific tolerance margin. Since the beam should normally strike the target unimpeded and uninterrupted, it is generally not possible to determine continuously whether the beam is actually striking the target,[52] nor measure the specific beam current on target. In some circumstances, the beam can be *sampled* periodically by dropping in a Faraday cup to measure the beam current;[53] under other circumstances, the beam current cannot be interrupted at all, and other means must be used to detect whether the beam is on target. In order to minimize the amount of time that the beam is lost from target during beam sampling, such sampling should be performed as infrequently as possible and last as briefly as possible.

---

[51] 'Optimum' in this case means 'optimum as prescribed by the operator'.

[52] If the sample power signal is available (i.e. the sample is being actively heated using dc current), then it is possible to detect a target strike indirectly, but not all experiments involve sample heating.

[53] Since dropping in a Faraday cup always stops the beam from travelling any further along the beam line, beam flow to target always gets interrupted by beam sampling.

Consequently, the operator is able to specify what type of *beam sampling* should be performed by the expert system:

- *None*: No beam sampling is performed. In this case, some other means (such as sample power monitoring) must be used to detect the presence of the beam on target.
- *Periodic*: The beam is sampled for a specified period of time with a specified frequency (e.g. sampled for 10 seconds every 10 minutes).
- *As needed*: The beam is sampled as needed by the expert system.

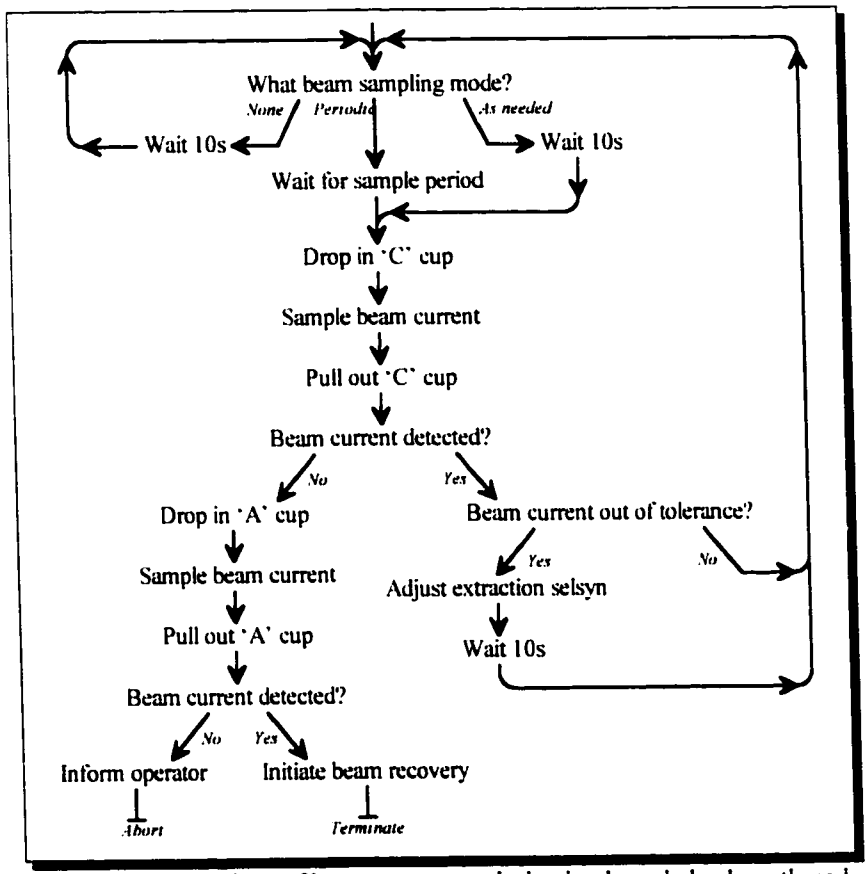


Figure 4-38. Flowchart of beam current optimization knowledge base thread.

The beam current optimization thread performs beam sampling as specified, and uses the measured beam current to make two decisions (as shown in Figure 4-38):

① If no beam is detected on the target Faraday cup ('C'), then the first Faraday cup ('A') is checked — if no beam is detected on 'A' cup, an operating fault is assumed, the operator is informed, and automated control is terminated.

② When a beam is detected on 'C' cup, the thread determines whether the beam current is within tolerance — if the beam current is out of tolerance, then the extraction selsyn is adjusted to bring the beam

current into tolerance. It is assumed that the beam is already properly focused for a specific beam energy (which has not changed), so no focusing is needed to optimize the beam current.

4.3.6.4 Sample Power Optimization

As outlined previously, the *sample power* signal is used to detect the presence of the beam on target indirectly. More precisely, the sample power signal can be used to determine the relative strength of the beam striking the target (q.v. § 2.3.2). The sample power optimization knowledge base thread (Figure 4-39) monitors the sample power signal to ensure that the sample on target is not being over-irradiated (which could damage the sample), or under-irradiated (which is insufficient for the experiment being performed). If the sample power signal is extremely low (indicating high incident beam current), the sample may be getting damaged; when this case is detected, safing action (dropping a Faraday cup and notifying the operator) is taken to protect the sample. Otherwise, if the sample power is out of tolerance, the extraction selsyn is adjusted to alter the beam current slightly. If the sample power signal does not respond quickly to extraction selsyn adjustment, the thread will initiate safing action (i.e. drop in the first Faraday cup and notify the operator) if too many consecutive extraction selsyn adjustments are made; this situation may occur if the beam is unexpectedly lost, or if some other operating fault completely disrupts the particle beam.

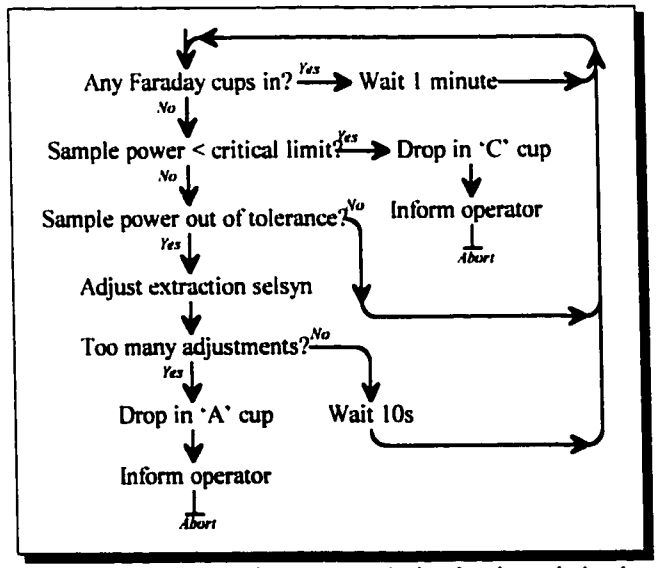


Figure 4-39. Flowchart of sample power optimization knowledge base thread.

4.3.6.5 Gas Optimization

It is desirable to maintain the accelerator system's vacuum level at an optimum level. The accelerator is constantly kept near vacuum by a series of pumps, but during operation, the vacuum level is reduced by the flow of source gas into the ionizing chamber and the generation of the beam plasma. If the source gas flow is too low (and the vacuum is high), there is insufficient gas to be ionized into a particle beam, and the beam is either non-existent, sputtering or unstable. If the gas flow is too high (and the vacuum is low), the quality and stability of the particle beam are affected by the presence of raw source gas in beam line. Therefore, the source gas flow rate must be well adjusted (using the gas selsyn) to optimize behaviour of the particle beam. The gas flow rate is controlled by a thermo-mechanical valve which has a long time constant (slow slew rate),[54] making closed-loop control difficult (q.v. § 5.1.3.3). Under manual operation, operators usually employ a strategy whereby the gas selsyn is set based on one of two conditions:

- If replicating the conditions of a past run, the gas selsyn is set to its previous setting.
- If not replicating a past run, the operator uses experience-based judgement to estimate an initial setting for the gas selsyn.

Thereafter, the gas flow rate is fine-tuned over the course of several minutes (or even hours), and possibly several times during the accelerator run, based on experiential knowledge.

The gas optimization knowledge base thread shown in Figure 4-40 is charged with the task of adjusting the source gas flow rate in an effort to maintain particle beam stability. The thread monitors the *vacuum* signal for deviation from a specified tolerance interval. When deviation occurs, a gas selsyn adjustment is performed which is proportional to the difference between the measured vacuum level and the optimum vacuum level.[55] A long waiting period (5 minutes) follows each gas selsyn adjustment to accommodate the slow slew rate of source gas flow.

---

[54] When the accelerator is started from a 'cold' state, the gas slew rate can be on the order of several minutes.

[55] 'Optimum' in this case means 'optimum as prescribed by the operator'.

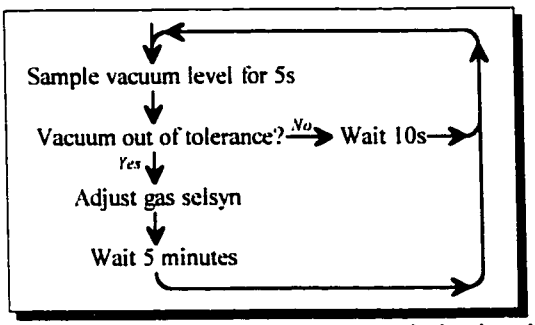


Figure 4-40. Flowchart of the source gas optimization thread.

◆ ◆ ◆

This section has described the design and implementation of the expert system-based control mechanisms used for automated accelerator operation. The next section explores the use of fuzzy logic-based control mechanisms to augment the capabilities of the expert system.

4.3.7 Fuzzy Logic-based Control Considerations

The parallel processing hierarchy discussed in Section 4.2.1 is intended to improve system response time to both the user and accelerator by dividing the overall control task between two or more processors. To accomplish this, the embedded controllers are charged with performing small subsets of the control problem. There are various methods for implementing control algorithms on embedded processors, including conventional control algorithms (such as PID control), and artificial intelligence-oriented techniques (such as neural networks). As evidenced in the section on knowledge-based systems in Chapter 3, fuzzy logic-based techniques have recently gained widespread popularity in application to a variety of control problems. Moreover, as explained in Chapter 3, fuzzy systems are conceptually more closely related to expert systems than neural networks because both expert systems and fuzzy systems use structured knowledge. The use of structured knowledge facilitates *system transparency* in that the 'knowledge' used for performing control actions is readily expressible in a well-defined manner that can be tested, modified and verified more easily than the unstructured knowledge stored in neural networks, for example. Consequently, both expert and fuzzy systems are desirable in applications which require accessibility to the underlying knowledge bases used for automation of complex systems.

As will be explained in the sequel, fuzzy logic-based controllers (FLCs) used in PACES are implemented as fuzzy associative memories (FAMs), ([Kos92]), which are stored in and executed by an SBC's RTK. The expert system is responsible for determining when a specific FLC should be activated, and how it should be configured. Once activated, however, the FLC operates autonomously until it completes its duties or is deactivated by the expert system. In this way, the expert system acts in a supervisory manner, relegating portions of the overall control problem to auxiliary processors, achieving the multi-processor topology described in Section 4.2.1.

Within the PACES application, fuzzy logic-based control techniques are used for two related types of control tasks. In the first (Type I), the expert system directs an FLC to acquire and then actively maintain a specified set-point. Once the FLC is activated, the expert system is relieved from the task of set-point maintenance until the set-point is changed (through user action) or an operating fault occurs in the accelerator. The process, illustrated in Figure 4-41, begins when the expert system, pursuing some chain of reasoning, decides to start an FLC for set-point acquisition and maintenance, and sends an RPC to the SBC (①). The RTK, which is executing its top level command loop (❶), initiates execution (inferencing) of the FLC's FAM (❷). Meanwhile, the expert system continues its current chain of reasoning (②). At some later time, the RTK may detect a breach of its pre-set operating limits (see Section 4.2.5.2) and signal the PC (❸), causing the expert system to switch its chain of reasoning to perform fault detection, diagnosis and recovery (③).

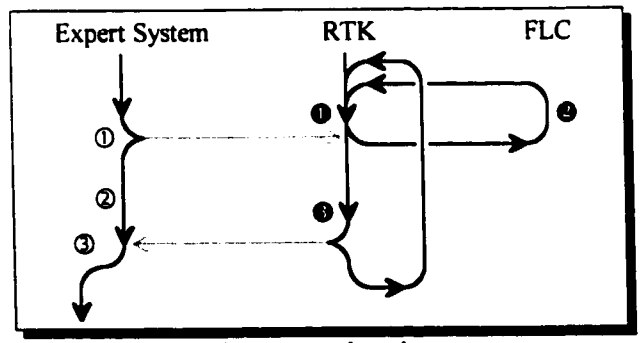


Figure 4-41. Type I control task: Interaction between expert system and fuzzy controller for set-point acquisition and maintenance.

A Type I FLC is used for establishing and then maintaining the belt charge selsyn setting required to generate a specific terminal voltage. It could also be used for acquiring and then maintaining the analyzing magnet current required to bend the particle beam so that particles of a specified energy are conveyed to the target.

In the second type of control task (Type II), the expert system activates an FLC periodically to perform optimization of a control parameter. When the FLC is activated, it operates continuously until its associated accelerator parameter has been optimized (that is, maximized or minimized). As shown in Figure 4-42, this process begins with the expert system sending an RPC to the RTK (①) to initiate the FLC (❶). The FLC then remains active and iterating (❷) until its control parameter is optimized (❸) at which time the RTK deactivates the FLC and signals the PC that the task has been completed (❹). Meanwhile, the expert system continues its chain of reasoning (②) until notification is received (③). This type of FLC is used for maximizing beam current by optimizing the extraction, focus and gas selsyn settings, and for optimizing gain control on the accelerator's voltage stabilizer subsystem.

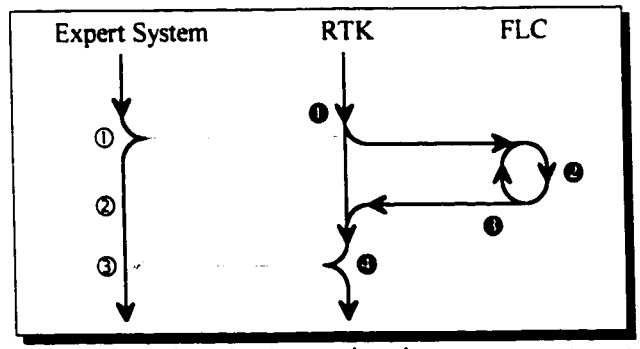


Figure 4-42. Type II control task: Interaction between expert system and fuzzy controller for control parameter optimization.

The FLCs are implemented as fuzzy associative memories (FAMs) which map input fuzzy state spaces to output fuzzy state spaces, ([Kos92]). Figure 4-43 illustrates the process of building a FLC. The FLC is first defined in a source code format (cf. Table 4-5), which specifies fuzzy membership functions, fuzzy variable types, fuzzy input/output variables, and FAM rules. A compiler translates this source code into a FAM which can be downloaded to the SBC when PACES is run. Finally, a library of fuzzy logic

inferencing subroutines running on the SBC is used to 'execute' the FLC's FAM to perform fuzzy inferencing.

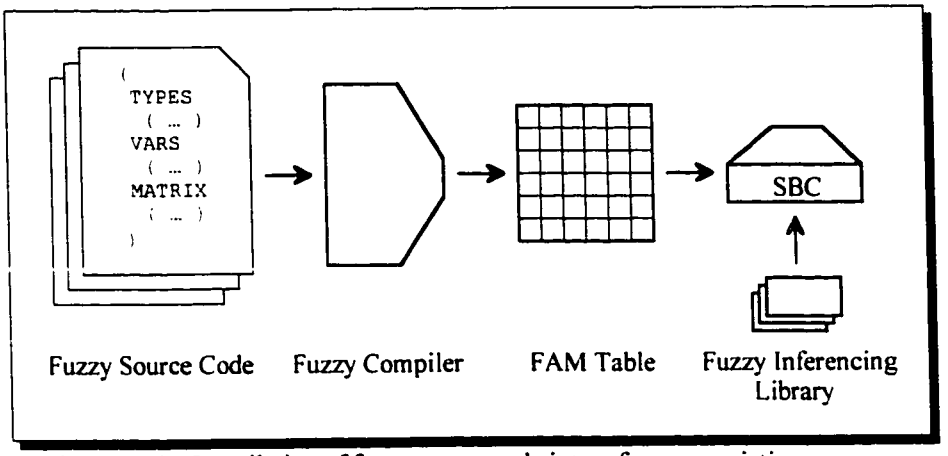


Figure 4-43. Compilation of fuzzy source code into a fuzzy associative memory.

4.3.8 Fuzzy Control of Terminal Voltage Set-point

This section describes the implementation of a fuzzy logic-based belt charge selsyn controller for terminal voltage set-point control, a Type I control task (cf. Figure 4-40). Three versions of this FLC were developed. The performance analysis of these FLC versions is presented in Chapter 5.

The belt charge selsyn FLC uses simple control rules to adjust the belt charge selsyn to acquire and then maintain a specific terminal voltage. This process is required during the initial stages of accelerator start-up (cf. Table 4-2), and then during the beam maintenance phase for coarse terminal voltage stabilization.[56] As indicated in Chapter 2, the terminal voltage usually drops during start-up when the particle beam begins to flow, because the particles carry charge away from the terminal as they are accelerated. This means that the operator has either to compensate by subsequently increasing the belt charge selsyn, or to provide for the voltage drop by initially setting the terminal voltage to a level higher than required. This problem is easy to remedy using the terminal voltage FLC, which acts to perform automatic compensation as needed.

An important aspect of terminal voltage control is the 'slew rate' (capacitive charging time) of the high voltage terminal. This is manifested as an average delay of approximately 3.5s between a belt charge selsyn adjustment and compliance by the terminal voltage.

[56] The accelerator's built-in voltage stabilizer subsystem is used for fine-scale terminal voltage stabilization.

Additionally, the accelerator used[57] has a 'dead zone' on its belt charge selsyn, and 11 full turns must be made initially before there is any commensurate charging of the high voltage terminal.

The first version of belt charge selsyn FLC developed had a single input and used the error between present and target terminal voltages to determine the number of steps to turn the belt charge selsyn (Figure 4-44). Table 4-5 shows the 'source code' that is compiled to produce the executable FLC: The `TYPES` section declares the membership functions for the fuzzy variable types `TerminalVoltageKV` and `BeltChargeTurn` (cf. Figures 4-45 and 4-46, respectively). The `VARS` section declares the FLC's two fuzzy variables, `deltaTV` and `deltaBC`. Finally, the `MATRIX` section defines the FAM that maps the input fuzzy variable (`deltaTV`) to the output fuzzy variable (`deltaBC`). Figure 4-47 plots the FAM's control surface, which maps error in terminal voltage to a belt charge selsyn adjustment. The fuzzy membership functions and FAM were derived from observations made of the control adjustments performed by expert operators during accelerator start-up.

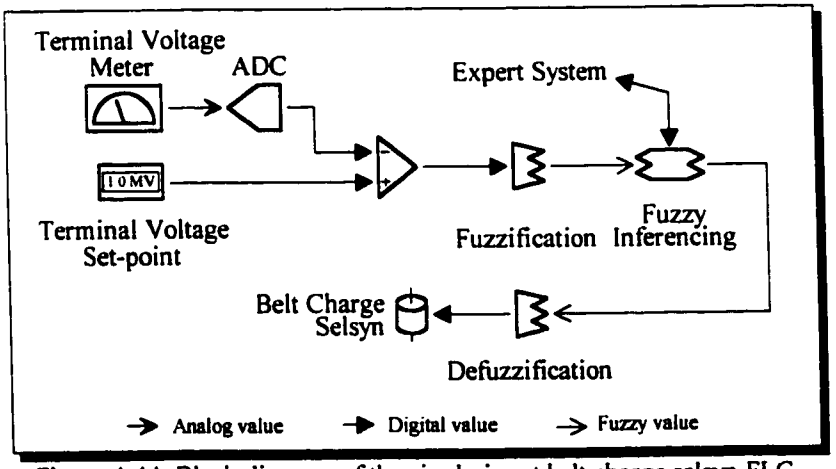


Figure 4-44. Block diagram of the single-input belt charge selsyn FLC.

[57] This FLC was installed and tested on the McMaster Accelerator Lab's KN-3000.

```
BeltCharge
 (TYPES
  (TerminalVoltageKV,-4.000,4.000,
   (WayTooLow,    Z,-4.00,-4.00,-2.00,-1.50),
   (TooLow,       Z,-2.00,-1.50,-0.55,-0.50),
   (Low,          Z,-0.55,-0.50,-0.15,-0.10),
   (LittleLow,    Z,-0.15,-0.10,-0.05,-0.06),
   (SlightlyLow,  Z,-0.06,-0.05,-0.025,-0.015),
   (OK,           T,-0.02, 0.00, 0.02),
   (SlightlyHigh,Z, 0.015, 0.025, 0.05, 0.06),
   (LittleHigh,   Z, 0.05, 0.06, 0.10, 0.15),
   (High,         Z, 0.10, 0.15, 0.50, 0.55),
   (TooHigh,      Z, 0.50, 0.55, 1.50, 2.00),
   (WayTooHigh,   Z, 1.50, 2.00, 4.00, 4.00)),
  (BeltChargeTurn,-100,100,
   (HugeCCW,   Z,-75,-75,-45,-40),
   (LargeCCW,  Z,-45,-40,-25,-20),
   (MediumCCW,Z,-25,-20,-15,-10),
   (SmallCCW,  Z,-15,-10, -5, -3),
   (SlightCCW,T,      -5, -3, -1),
   (NoTurn,    S,              0),
   (SlightCW,  T,      1,  3,  5),
   (SmallCW,   Z,  3,  5, 10, 15),
   (MediumCW,  Z, 10, 15, 20, 25),
   (LargeCW,   Z, 20, 25, 40, 45),
   (HugeCW,    Z, 40, 45, 75, 75))
 VARS
  (TerminalVoltageKV deltaTV),
  (BeltChargeTurn    deltaBC)
 MATRIX
  deltaBC = deltaTV *
     (
      WayTooLow     HugeCW,
      TooLow        LargeCW,
      Low           MediumCW,
      LittleLow     SmallCW,
      SlightlyLow   SlightCW,
      OK            NoTurn,
      SlightlyHigh  SlightCCW,
      LittleHigh    SmallCCW,
      High          MediumCCW,
      TooHigh       LargeCCW,
      WayTooHigh    HugeCCW
     ))
```

Table 4-5. Source code for single-input FLC.

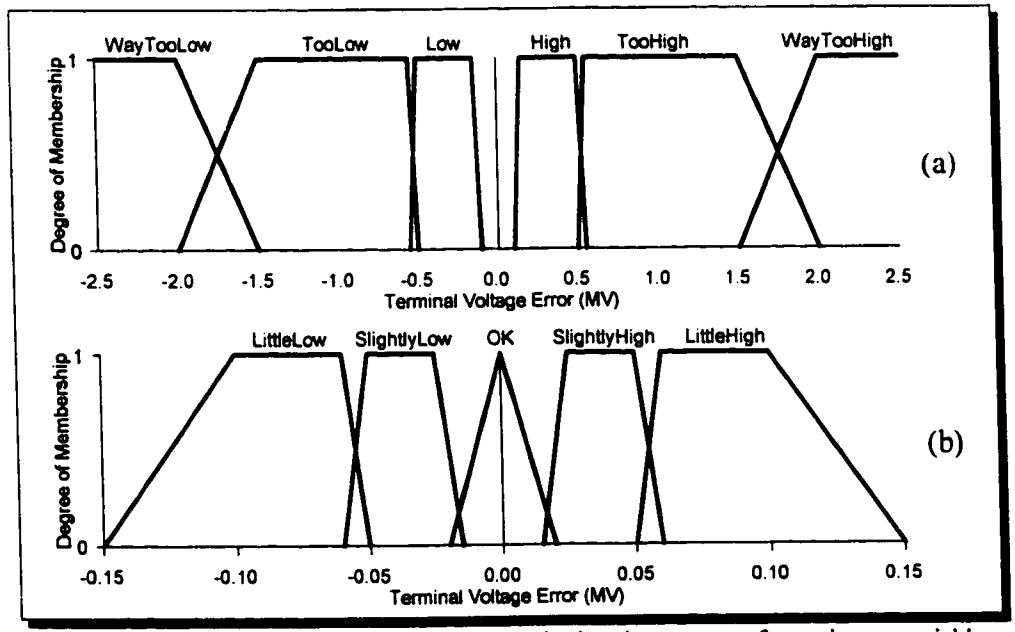


Figure 4-45. Membership functions for terminal voltage error fuzzy input variable. Part (b) is a magnified view of the membership functions located about the zero point in part (a).

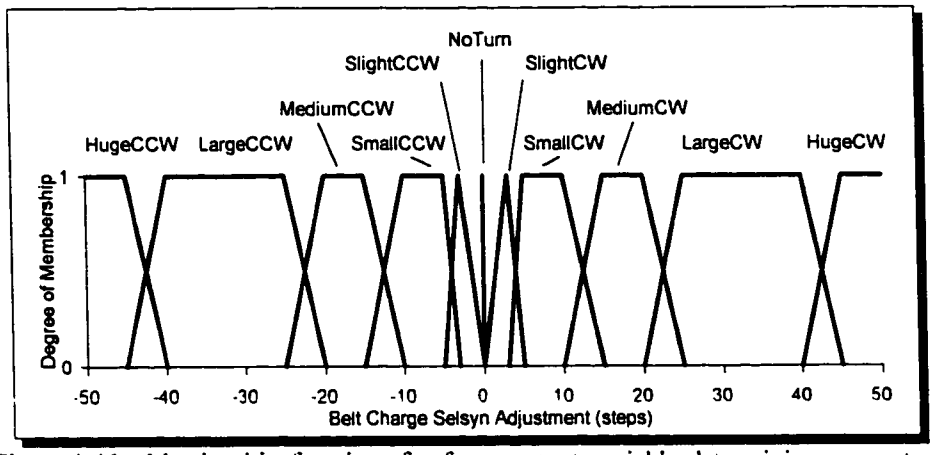


Figure 4-46. Membership functions for fuzzy output variable determining amount of belt charge selsyn adjustment.

This first version of the FLC used the average slew rate as a fixed loop delay of 3.5s between control iterations, and consequently may have spent more time than necessary waiting for terminal voltage compliance. To remedy this, the FLC was changed to 'execute' with a loop delay dependent on the derivative of the terminal voltage: the next iteration of the control loop was delayed until the terminal voltage derivative reached near-zero. This modified version resulted in improved performance, as presented in Chapter 5.

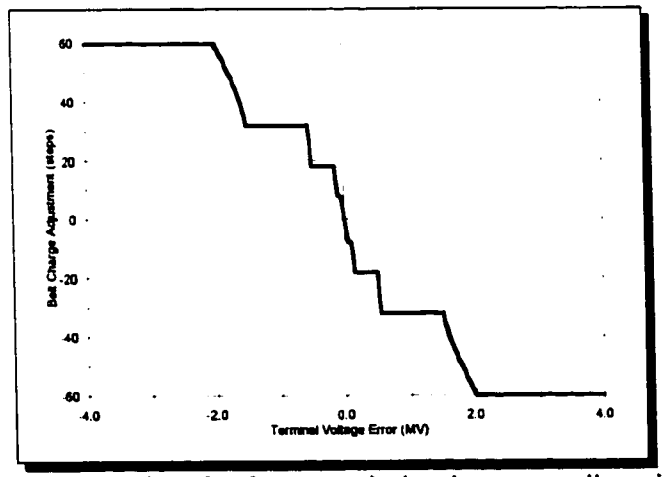


Figure 4-47. Control surface for fuzzy terminal voltage controller with one input (terminal voltage error).

A further modification to the FLC, depicted schematically in Figure 4-48, involved inclusion of a second input variable, namely the derivative of the terminal voltage. The membership functions for this second input variable are shown in Figure 4-49. Since a derivative-dependent delay was not needed, the control loop delay was fixed at 1s.

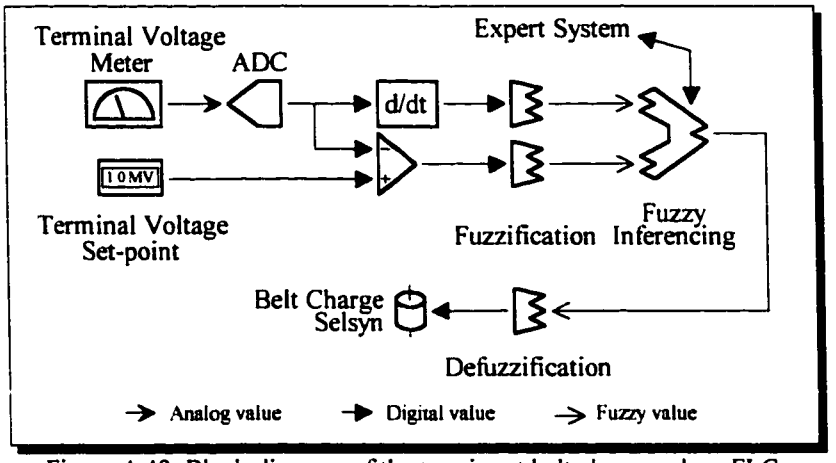


Figure 4-48. Block diagram of the two-input belt charge selsyn FLC.

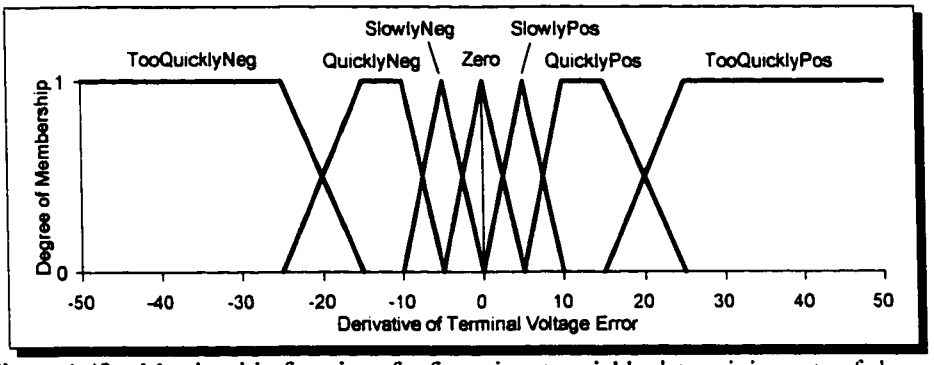


Figure 4-49. Membership functions for fuzzy input variable determining rate of change of terminal voltage (first order differential).

Figure 4-50 shows the control surface of the two-input FLC which determines belt charge selsyn adjustment as a function of both terminal voltage error and derivative of terminal voltage. The flat region along the left side of the control surface represents an area of inactivity, regardless of terminal voltage error, when the derivative of terminal voltage error is large and negative; such a precipitous drop in terminal voltage is likely indicative of a spark (high-voltage discharge within the accelerator tank or column) for which no belt charge selsyn adjustment should be made.[58]

[58] In this case, the expert system invokes high-level reasoning to recover from the spark, and the FLC remains inactive until recovery is complete.

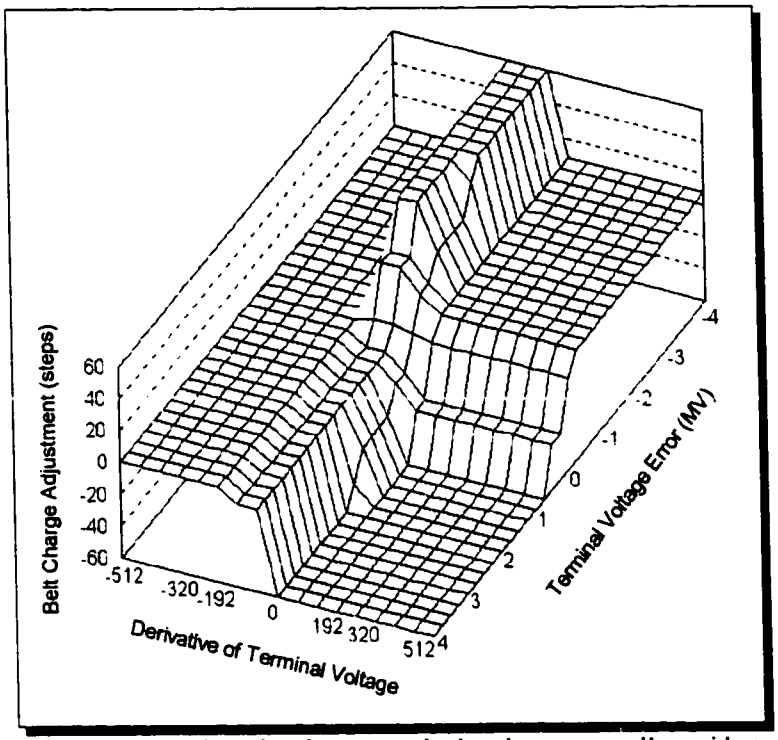


Figure 4-50. Control surface for fuzzy terminal voltage controller with two inputs (terminal voltage error and derivative of terminal voltage).

◆ ◆ ◆

This section has described the use of fuzzy logic-based controllers, and presented the design and implementation of a FLC for terminal voltage control. As will be shown in Chapter 5, this FLC performed its task successfully, indicating that fuzzy logic-based controllers are worthy of inclusion in the accelerator control system.

The next section investigates aspects of human factors that figure prominently in design and implementation of PACES.

4.4 Human-Factors Aspects

Chapter 3 discussed some of the broad human factors issues that are important in building computer systems that are designed to 'fit the user' in order that they demonstrate acceptability (utility, usability and likability). These characteristics are essential to PACES because the system's primary duty is to *assist* (rather than to replace) users in their operation of the accelerator. Since PACES is intended to be a computer-centered means of modernizing accelerator operations, it is imperative that the system is accepted (liked and used) by the operators. This requirement can be facilitated

by involving the operators in the development process, as has been done with PACES. Operator involvement serves to keep the operators interested in the project, and increases the likelihood that the system will be accepted when finally installed, since the operators have a sense of pride in what they have helped to develop for their own use. Operator involvement is, of course, also necessary to accomplish knowledge engineering and expertise transfer. In addition to direct operator involvement, other aspects of human factors were important considerations during PACES development.

The following sections consider some of the areas of PACES development that are concerned with concepts of human-computer interaction and the computer-human interface, including development of the PACES GUI, augmentation of operator abilities, and automation of operator expertise.

4.4.1 PACES User Interface

The user interface is PACES' link with the operator, the 'window' through which the operator interacts with the computer program and, in turn, the accelerator. As stated in Chapter 1, it is desirable that the system eventually evolves into a 'hermeneutical' (interpretational) state, ([Hol91]), in which the operator perceives the computer system as an integral part of the underlying physical process (accelerator). Although desirable, this state may be difficult to accomplish because the PACES GUI is situated in front of the real control panel, which is visible at all times. Will the operators look only at the computer screen, or will they find their eyes be continually drawn to the real control panel that they are familiar with and that is spread out right in front of their eyes? As will be explored in the sequel, this interesting question captures the essence of whether the technology insertion problem will be successful: *Will the operators voluntarily surrender the familiar, well-known physical control panel for the computer-based facsimile?* A partial answer to this question is: *They will, if the computer system enables them to do things they could not otherwise accomplish using the conventional control panel.*

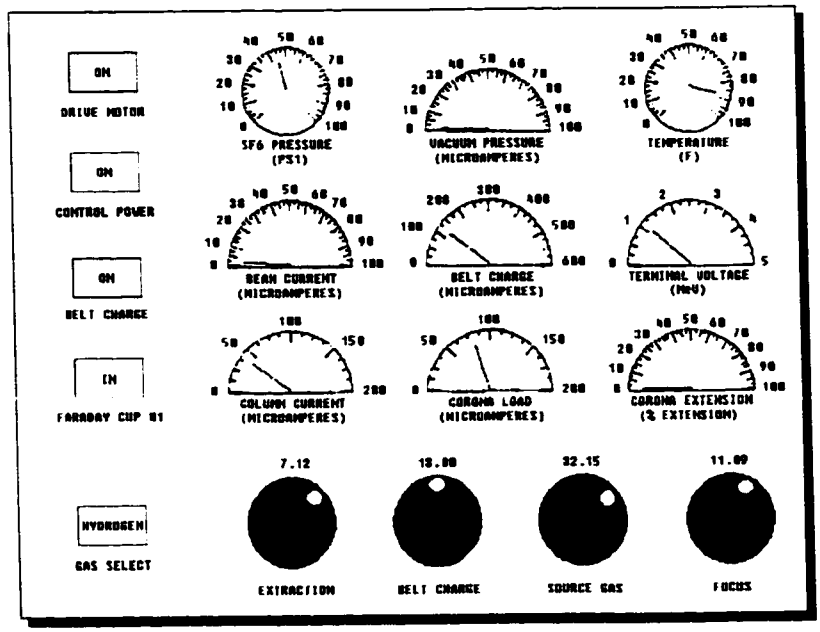


Figure 4-51. An early version of the PACES user interface. [DeM91]

In an effort to make the system desirable (and easy) to use, it was decided from the outset that the control system should use high-resolution computer graphics to produce a user interface that closely resembled the real control panel which the operators were well accustomed to using. Consequently, the facsimile control panel would require graphical versions of control panel instrumentation such as meters, selsyns and switches. Several early attempts at building such a graphical user interface (GUI) met with poor success, mainly because of the high degree of effort required to implement flexible, fast and easy-to-use GUIs. Figure 4-51 captures the main screen of one such attempt ([DeM91]) that was implemented from scratch in Turbo Pascal. Although these early attempts provided for reasonably good looking control panel facsimiles, it was found that too much effort was required to design properly, lay out and provide functionality for the many graphical objects. The main reason for this was that the GUI software had to be developed from scratch, requiring a large expenditure of effort. It was quickly realized that what was needed was an off-the-shelf GUI toolkit that would provide the needed flexibility and functionality without the overhead of developing the GUI from scratch.

After this initial failure to produce a capable GUI, it was decided that Microsoft Windows[59] offered a good platform for the PACES GUI.[60] Windows implements a powerful,

[59] Microsoft Corp., Redmond, WA.

[60] There is currently a large number of 'windows-based' GUI platforms available (such as OS/2, XWindows, GEM, and the Macintosh operating

object-oriented, graphical user interface environment suitable for rendering the facsimile control panel, complete with the required meters, switches and selsyns. Additionally, Windows has recently become such a popular PC-based operating system that it can be assumed that many PACES users will already be familiar with the standardized Windows GUI and even consider it their preferred medium of computer interface. This implies that less effort is required for users to learn how to interact with the PACES user interface, and that ultimate acceptance may be predisposed.

The Windows environment is predominantly a graphical, object-oriented *direct manipulation* style of human-computer interaction (cf. Section 3.3.1). The computer's mouse and keyboard are used to manipulate graphical objects that closely resemble real world counterparts in appearance and behaviour. This style of interaction is extremely powerful, and has recently become commonplace in a wide variety of computer environments. In many arenas, such as home or business computing, users frequently *demand* that the computer system offers this style of interaction. People consider direct manipulation (for some tasks) to be both visually and conceptually appealing, and find it easy to use, thereby promoting system usability and likability, leading to system acceptability.

Windows provides a multi-tasking 'operating system' for the co-ordination of user applications. It has a large library of general purpose subroutines, called the application program interface (API), which serves as a toolkit for building object-oriented graphical interface objects, managing system resources, and moderating inter-application communication. Initially, Windows applications could only be written in C, but more recently, a wide variety of programming languages has emerged for development of Windows applications, including Pascal, C++, BASIC, and FORTRAN. Figure 4-52 illustrates how Windows, DOS,[61] the application compiler and PACES (the application) are related.

---

system), and much of what is said in the sequel can be generally applied to any of these types of GUI platforms.

[61] *DOS* stands for disk operating system, a generic term, that has become the commonplace name for PC-based disk operating systems such as IBM's PC-DOS and Microsoft's the highly popular MS-DOS.

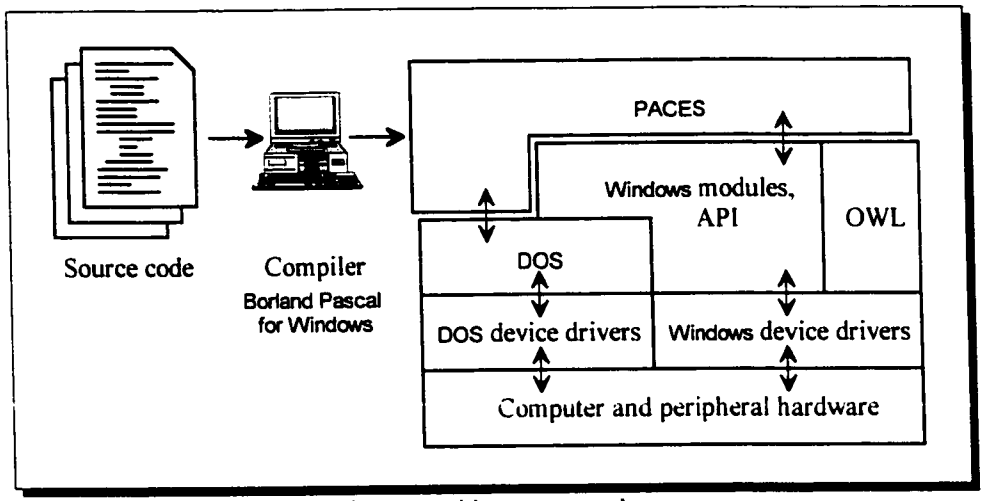


Figure 4-52. Interrelation of PACES with Windows and DOS. *Adapted from [Bor91], Fig. 1.3, p. 1*

The foremost feature of Windows applications is that they are *event-driven*. In contrast to a 'conventional' program, a Windows program handles all user input as *events*, which cause the program to alter its behaviour in response to the events. The Windows operating system responds to events by generating *messages* which are sent to the application program for processing. The program, therefore, is structured around a message processing engine which accepts messages from the Windows operating system and responds to them by activating different program subroutines. In this way, a Windows program is almost always 'waiting' for the user to generate events to which the program can respond. At the same time, however, since Windows is a multitasking environment, other parts of the program can be functioning in parallel with the main program component that waits for user input events.

The principal mode of input to the Windows system is the computer's mouse, although the keyboard plays a close secondary role. Most Windows applications are structured to accept mouse-based input as the main source of events, but also provide equivalent behaviour for keyboard input. The use of a mouse provides for a conceptually simple 'point and click' style of human-computer interaction; the user moves the mouse pointer around the screen, and presses the mouse's buttons to cause actions to occur. This commonplace style of interaction is well suited for the PACES application. In general, accelerator operators are accustomed to an 'eyes front, hands on' form of interaction with the accelerator: they face the control panel to observe its meters, and use their hands to

perform control tasks. Thus, it is sensible to design the PACES user interface so that the operators interact with the software in a similar manner.

PACES provides a GUI that closely resembles the actual accelerator control panel (cf. Figures 2-8 and 4-3). The control panel's meters, selsyns and switches are reproduced on the GUI, with effort taken to preserve as much of their appearance and behaviour as possible. The GUI meters have needles that are updated at regular intervals[62] to reflect the values displayed on the control panel meters. The mouse is used to manipulate the selsyns and switches: the user points the mouse cursor at a selsyn or switch and then presses the mouse's button to 'push' the switch or 'turn' the selsyn. Likewise, the various 'tools' available to the operator (cf. Section 4.1.4) are invoked in a similar manner by clicking on their associated iconic buttons.

The GUI resembles the real control panel in colour and approximate layout: The GUI's background is grey, meters have white faceplates, and selsyns have black knobs with silver cranks, just like the real control panel. The positioning of the meters, selsyns and switches also closely resembles the layout of the real control panel. Appearance is seen as a crucial aspect of system acceptance by operators, and it can be said that in this case *familiarity breeds acceptance*. One seasoned operator, seeing the PACES GUI for the first time, was pleased to note that, for example, the gas selsyn was positioned beside the extraction selsyn "as it should be", and that the scale on the vacuum system meter was "as it should be", ([Str92]). This operator was enthusiastic about the GUI from that moment onward, and took active involvement in the evolution of the GUI. Had the GUI been lacking in or completely devoid of resemblance to the real control panel, he would likely have been less positive about having to learn an unfamiliar computer system.

The GUI possesses what could be termed 'instrumentation consolidation' in that it concentrates instrumentation components that are, on the real control panel, spread across several large cabinets (cf. Figures 2-8 and 4-3).[63] This consolidation helps to focus the operator's attention and enables all major indicators[64] of accelerator state to be assessed in

---

[62] As mentioned previously, PACES receives telemetry data from the SBC every 200 ms, and the meters are updated at this rate.
[63] The instrumentation is spread out across these cabinets for technical and historical reasons.
[64] Some indicators are not present on the GUI, and not even sensed by the system, because they are not considered important enough or are, at present, too costly to sense digitally.

one field of view; the operator's gaze (and hands) no longer need to range across several large cabinets. An important issue to consider with respect to such instrumentation consolidation is the information density, or 'display loading', of the GUI. It is desirable to display as much information as possible, but this must be weighed against display (over)loading, which only serves to confuse the operator due to clutter and information overload. Some good guidelines for such human factors aspects of GUI design are discussed in [Gal97], [Gil89] and [Man97].

◆ ◆ ◆

The PACES GUI is designed to facilitate two aspects of human-computer interaction that are important to computer-assisted accelerator operation. In the following sections, the *augmentation of operator abilities* and the *automation of operator expertise* are investigated.

4.4.2 Augmentation of Operator Abilities

In relation to Section 2.2, the GUI analogues of control panel instrumentation components yield some benefits over the real control panel, but also pose some problems. The GUI meters provide more functionality than their real-world counterparts, but the GUI selsyns lose some of the useful characteristics of real selsyns. The GUI meters have the ability to display numeric values simultaneously with the analog values (needle positions), giving the operator a choice of analog or digital information (Figure 4-53). Generally, the analog form is used to gauge accelerator behaviour quickly in a qualitative manner, that is, to get an intuitive 'feel' about the accelerator. The digital information, on the other hand, frees the operator from having to interpolate the meter's scale, and provides exact, quantitative information when needed. Furthermore, the GUI beam current meter, whose dynamic range spans nine decades from nA to mA, has been given the ability to change its decade scale automatically as needed. This feature is seen by operators as a great improvement over the real-world beam current meter, whose decade scale must be switched manually whenever its signal goes off scale. Finally, the integration used to smooth needle ripple can be switched on or off individually for each GUI meter as the operator desires. Here again, there is improvement over the real-world meters because the

operator can choose between stable display (integrated signal) or 'jumpy' display (raw signal); the former is generally preferable, but there are times when it is more informative for the operator to see the true, non-integrated signal. Seasoned operators would usually prefer to see the non-integrated signal because they can glean more information from the raw signal than the integrated one; in contrast, novice operators would likely find the 'jumpiness' of the raw signals too distracting, and therefore prefer the integrated signals.

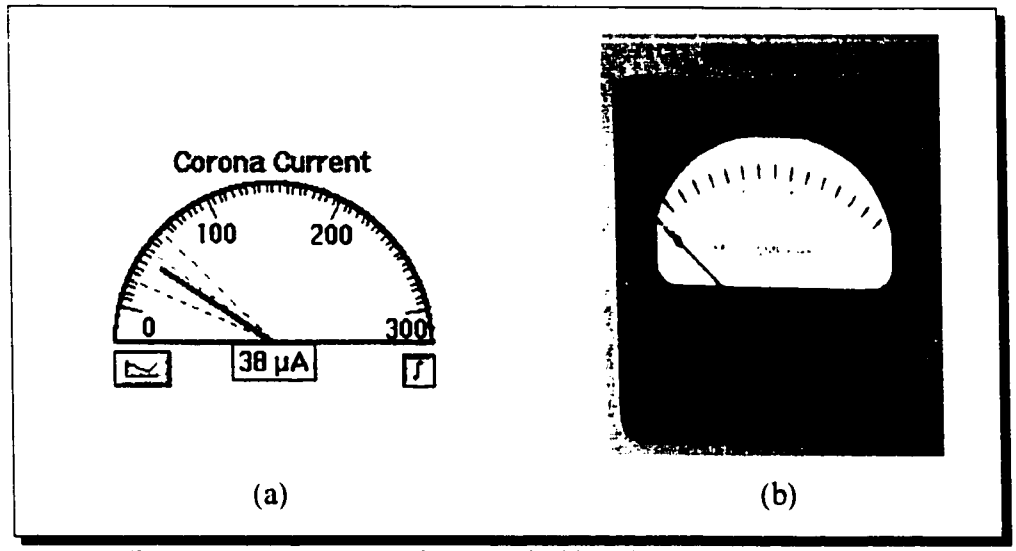


Figure 4-53. GUI meter (a) juxtaposed with real control panel meter (b).

The GUI selsyns have both added functionality and loss of functionality over the real selsyns (Figure 4-53). To their benefit, they require less hands-on control. The operator is able to perform 'fire and forget' operation using the *selsyn controller window* (cf. Figure 4-6), instructing a selsyn how much to turn, or to which absolute position to turn, and can leave the computer to keep the selsyn turning until its set-point has been reached. Such actions can be combined to adjust several or all selsyns automatically at the same time. This ability extends the user's grasp, where before a single operator could never manipulate more than two selsyns simultaneously. Each selsyn can also be switched to 'automatic', causing the computer to perform knowledge-based set-point maintenance while other selsyns are manipulated, thus providing a form of automatic compensation that was previously unavailable.[65] As a result, the operator can, for example, lock the belt charge selsyn to maintain a terminal voltage of 1.5MV automatically while manually

[65] One operator at McMaster termed this operation the 'third hand' capability, with which he was extremely pleased.

varying the extraction selsyn to adjust particle beam current (which directly affects terminal voltage).

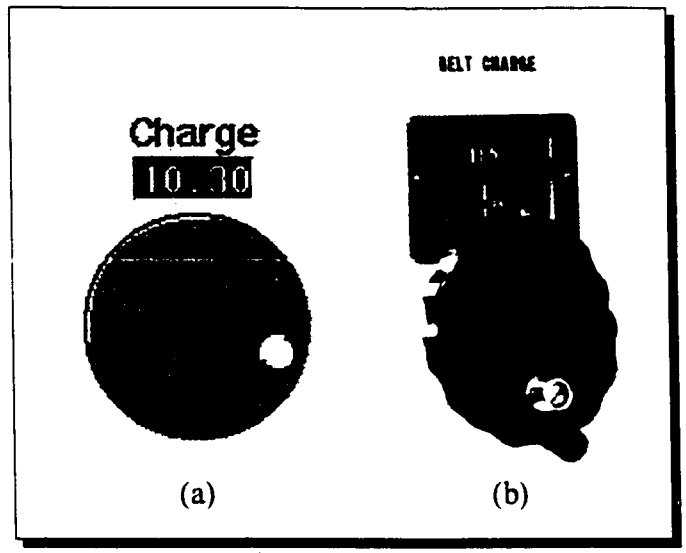


Figure 4-54. GUI selsyn (a) juxtaposed with real control panel selsyn (b).

On the negative side, the GUI selsyns have lost the torsional feedback that is so useful in their real world counterparts. Operators have learned to rely on the 'feel' of a selsyn to determine when it has reached zero or its upper limit, or when slippage is occurring in its tank-mounted partner. This loss of feedback has been found to impair operators because they have learned to rely heavily on the feedback. It would be possible to mimic such feedback with additional hardware, but this is deemed too costly and cumbersome to be worthwhile. A related problem deals with the stepper motors that are used to turn the control panel selsyns: Operators have reported that when the computer system is off and the selsyns are turned manually, there is a small but noticeable degree of resistance in the unenergized motors, causing a confusing and disliked change in the 'feel' of the selsyns. Evidently, the advocated 'non-invasiveness' of the piggy-back machine interface is not perfect.

A key facet of PACES design has been the operator-assisted evolution of the system. As development of PACES proceeded, field tests were performed at DREO and McMaster in which operators were asked to use and evaluate the system. Then, based on their complaints or suggestions, the system was altered and improved. This form of operator participation in development is considered valuable because it ensures that the operators

are kept interested and actively involved, and that the final version of the system will be an accepted and welcomed arrival. Four examples of this valuable interaction are presented in the sequel to illustrate how PACES has evolved to suit the perceived needs of the operators. It is noteworthy that such evolutionary development can proceed even after the system is installed, and the PACES object-oriented software is designed to accommodate such ongoing extension and expansion.

On one visit to DREO, an operator commented that one piece of hardware on the real control panel was particularly useful: a paper stripchart recorder that could be used to monitor target beam current varying over time. This capability was subsequently added to PACES in the form of a stripchart window for each meter (Figure 4-55), which could be opened (or closed) as desired to monitor the time-varying behaviour of an accelerator state variable. The stripchart windows also plot the first-order derivative, giving them capability beyond that of their real-world namesake. On the next visit to DREO, the operators were thrilled that they could now call up stripcharts for any meter, not just the beam current meter, and that they could observe rates of change as well.

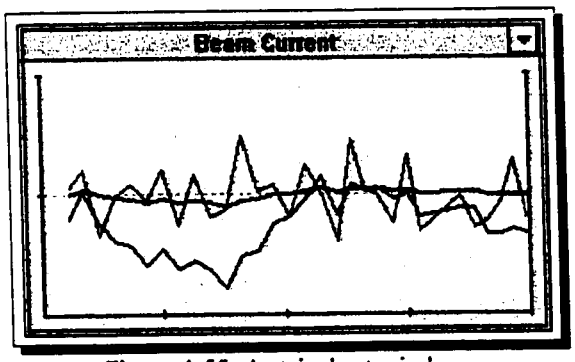


Figure 4-55. A stripchart window.

Another example of GUI evolution is the incorporation of a 'kiviat graph' (Figure 4-56). This type of multi-axis graph is used primarily in computer operating systems to indicate system balance and resource loads. Each axis measures a different variable, and lines link the points on neighbouring axes to form a polygon. The axes' scales are adjusted so that a 'normal' state is implied when the polygon most closely resembles a circle; system imbalances are therefore indicated by deviations from circular. This type of graph was added to PACES to provide an 'accelerator at a glance' indicator that would convey a collection of important accelerator state variables simultaneously to

imply machine stability. Operators have expressed interest in the concept of such a broad-spectrum indicator which is unparalleled on the real control panel.

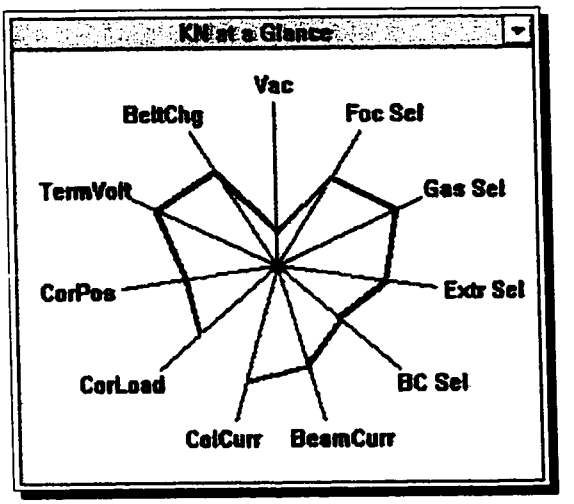


Figure 4-56. A kiviat graph of accelerator state.

During another visit to DREO, the site manager happened to mention the fact that at least two people were always needed during an accelerator run, one (the experimenter) performing experiments in the target area, and another (the operator) stationed at the control panel. The site manager commented that it would be of great advantage if the computer system could somehow be used to provide control panel access from the target area, making the accelerator site as a whole easier to use. After some thought, it was decided that the system could easily be extended by adding an additional PC in the target area which would communicate with the main PC via an RS-232 serial link. This optional remote console was recently given a test-run and demonstrated that it was possible to teleoperate the accelerator remotely just as effectively as from the control room. The remote operation facility has shown promise of reducing site staffing requirements and increasing availability of the machine since less people are required for operation. Furthermore, the serial link can be extended with the use of modems to provide long-distance accelerator operation, enabling off-site analysis and trouble-shooting of a malfunctioning accelerator.

One final example of GUI evolution involves the use of a sound card installed in the host PC to facilitate the use of high-quality sound effects as an extension of the user interface. PACES is able to emit a wide variety of audible indicators, such as bells and

sirens, which are used to gain the operator's attention. At AECL's Whiteshell Labs, PACES emits a digitized recording of a distinctive and familiar alarm when it detects that the beam has been lost from target; operators accustomed to the existing hardware-based alarm instantly recognize this software-based alarm, and know immediately that the beam has been lost. This, then, is another example of how PACES' mimicry of the real control panel facilitates operator acceptance and ease-of-use by experienced operators.

Plans exist to connect the audio output to the accelerator site's public address system so that operators are still in audible contact with the control system even when not physically present at the computer console. In this way, the computer can sound alarms and convey information that can be heard throughout the site, freeing operators from constantly having to remain at the computer console during accelerator operation.

◆ ◆ ◆

The augmentation of operator abilities forms one half of the objective of furnishing accelerator operators with a computer-centered system for assisting them in their work. The second half of the objective takes the form of automation of operator expertise, as is explored in the next section.

4.4.3 Automation of Operator Expertise

Figure 4-57 illustrates the accelerator user community in a general sense, consisting of operators and experimenters. Operators can be loosely categorized as *novices* or *experts*. Experimenters can be considered as *operators* (novice or expert) and *non-operators*.

The operations expertise incorporated into PACES consists of a large body of operating procedures, established step-by-step sequences for start-up, shut-down and beam maintenance operation. The procedures are generally perceived as mundane, and are the drudgery of accelerator operation. Accelerator users are seldom interested in start-up and shut-down, viewing them as necessary but uninspiring phases of accelerator operation. Operators are more interested in the intellectual aspects of operating demands for the day's experiments and the possibility of seeking novel means of achieving the requirements of the experiments, such as, for example, the challenge of fine-tuning the particle beam to get a mere trickle of particles of a specific energy. Experimenters, in contrast, are, by and

large, more interested in the physics underlying the experiments, and prefer to look beyond the humdrum of machine operation to the realm of scientific experimentation. Therefore, these established procedures, because of their algorithmic nature and their disappeal to humans, are ideal for automation by computer.

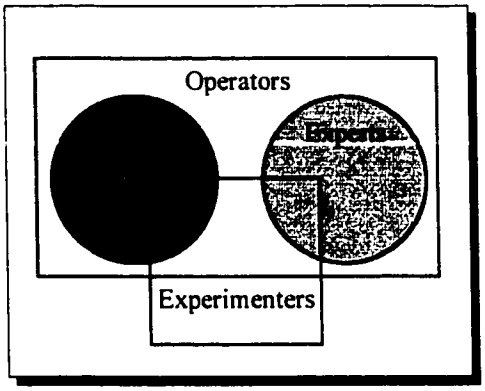


Figure 4-57. Operators and experimenters.

This form of automation affords benefit for novices and experts alike. Novices divested of these mundane duties are able to learn about accelerator operation in a controlled environment. Every day, the same procedures can be used to start-up the accelerator to a predictable state from which the operator-in-training is able to explore, on an ongoing basis, the finer details and nuances of accelerator operation. Eventually, as the novice operator's skills develop, the novice can rely less and less on automated operation if so desired. Additionally, the system's knowledge base can be employed for safeguarding the accelerator from errors and damage caused through lack of experience: the computer system can prevent novices, for example, from operating above 1 MV of accelerating potential, or above 10 μA of target beam current. The knowledge base can also be used as an advisor or tutor for operators-in-training, providing examples, guidance and tutorials of established operating procedures. Finally, the computer system can be employed as a training assessment monitor, logging operator control actions and machine data for later off-line analysis by an expert operator.

Whereas PACES facilitates training of novice operators, it also offers advantages for expert operators. Primarily, the experts are liberated from the drudgery of mundane accelerator operation so they can better spend their time preparing for the experiments at hand. For example, an operator could invoke automated accelerator start-up, begin

preparations in the target area while the computer is performing the start-up procedure. and return to the control panel when the particle beam has been stabilized and is ready to be conveyed to the target area. Moreover, the expert can make use of PACES' 'third hand' facility when making fine adjustments of accelerator state: one or more selsyns can be set to automatic so that the computer performs any compensation required as the operator makes control adjustments manually. This again is a form of liberation: the operator can concentrate on a specific control point of interest and rely on the computer system to compensate other control points as needed. In addition, PACES' data logging facility can be used for later off-line analysis, giving the expert a chance to experiment with the machine and later observe and analyze quantitative performance data.

The computer system also serves to bridge the gap between operator and experimenter. Traditionally, the experimenter works in the target area and the operator monitors the control panel some distance away. PACES' ability for remote operation enables the operator to control the machine from the target area, thereby permitting a strengthening of the experimenter-operator relationship. Operator and experimenter are able to take part in the experiment together, each gaining better understanding and appreciation of the other's perspective. This relationship can only serve to improve overall operations of the accelerator site due to improved personnel interaction.

◆ ◆ ◆

This section has explored the human factors aspects of PACES development, and documented examples of how the system has evolved under the assistance of accelerator operators into a software product that the operators find acceptable (useful, usable and likable).

The next section investigates the aspects of software engineering that were prominent in PACES' software development lifecycle.

4.5 Aspects of Software Engineering

Attention was drawn in Section 3.4 to the importance of properly applying formal software engineering methods to software development, and although there are many varying approaches and schools of thought, it is evident that a software development

project benefits from the employment of *some* form of software engineering principles. PACES is no exception to this claim, and software engineering methods have featured prominently in its design and implementation. In parallel to Section 3.4, the following sections will relate PACES' software engineering aspects on both the macro- and micro-scale levels.

4.5.1 The Macro-scale: PACES Development Lifecycle

PACES has been required to play a dual role during its development. In one respect, its purpose was to meet a contractual obligation to provide the Defence Research Establishment Ottawa with a working, computer-oriented accelerator control system; in another respect, it was seen as a foundation for academic investigation of the application of artificial intelligence and real-time systems concepts to computer-centered complex process modernization. The direct result of this duality is that many of the formalized macro-scale software engineering principles advocated in Section 3.4 were difficult to apply to PACES development. For example, PACES development was originally envisioned to involve the efforts of two people: a physicist would build PACES from the 'top down' by performing knowledge base development; simultaneously, an electrical/computer engineer would build PACES from the 'bottom up' by implementing the underlying accelerator interface, parallel processor system, and graphical user interface. This approach can be compared to the U.S. Department of Defense standard DoD STD 2167-A software lifecycle model (cf. Figure 3-27), which possesses parallel streams for development phases for both hardware and software. Similarly, the PACES approach can also be likened to the POLITE lifecycle model (cf. Figure 3-32), which possesses parallel streams for development of knowledge-based and 'conventional' program components. Yet, in another way, the PACES development lifecycle shares elements in common with the evolutionary rapid prototyping model (cf. Figure 3-30), in that rapid, evolutionary prototyping was used to build workable field versions of PACES quickly which could be tested and evaluated by accelerator operators. Indeed, even the spiral lifecycle model (cf. Figure 3-29) is mirrored in PACES development, from DREO's perspective, because as prototype versions became available for evaluation on specific

deadlines, the DREO personnel had the option of renewing, extending or cancelling the project.

Since PACES' development lifecycle appears to share elements in common with several different models, it is difficult to clarify the exact form of the PACES lifecycle model. Nevertheless, Figure 4-58 attempts to illustrate this lifecycle model as a useful amalgam of several other popular lifecycle models. For lack of a better name, it is called the 'parallel-threaded prototyping' (PTP) model. This model is a combination of various existing models that has proven effective in the development of PACES.

The PTP lifecycle model shown in Figure 4-58 begins with a *preliminary specification and design* phase in which the general requirements and functionality of the overall system are defined. During the next phase, the design and specification become more detailed, and branch into two parallel development threads: *hardware* and *software*. The software thread in turn branches into three threads: *machine interface*, *user interface*, and *knowledge-based system* (*KBS*). Finally, the *KBS* thread divides into the *engine* and *knowledge base* threads. As this division into parallel threads occurs, an increasing amount of detail is added to the overall system design as the design of the individual threads becomes more developed. During the next phase, *parallel development*, each thread follows a typical RUDE-like iterative cycle consisting of *design*, *build*, and *test/modify* phases, but there is intercommunication between the threads in each phase to promote eventual integration.

At some point during the *test/modify* phase, a decision is made that the overall system is ready for prototyping, and the next phase, *prototyping*, is entered. In this phase, the threads are integrated to form a prototype *version 'n'* of the system. This prototype is released ('fielded') for evaluation by the end-users,[66] who decide whether the version is acceptable. One of four outcomes can result from this evaluation:

① If all goes poorly, and the end-users are completely dissatisfied with the system, they can opt to *scrap* it altogether — this is usually an unlikely scenario, but still a possibility.

[66] This group includes not only users *per se*, but all personnel that are able to affect the evaluation of the prototype, such as site managers, administrators, licensing officers, commissioning staff, system maintainers, etc.

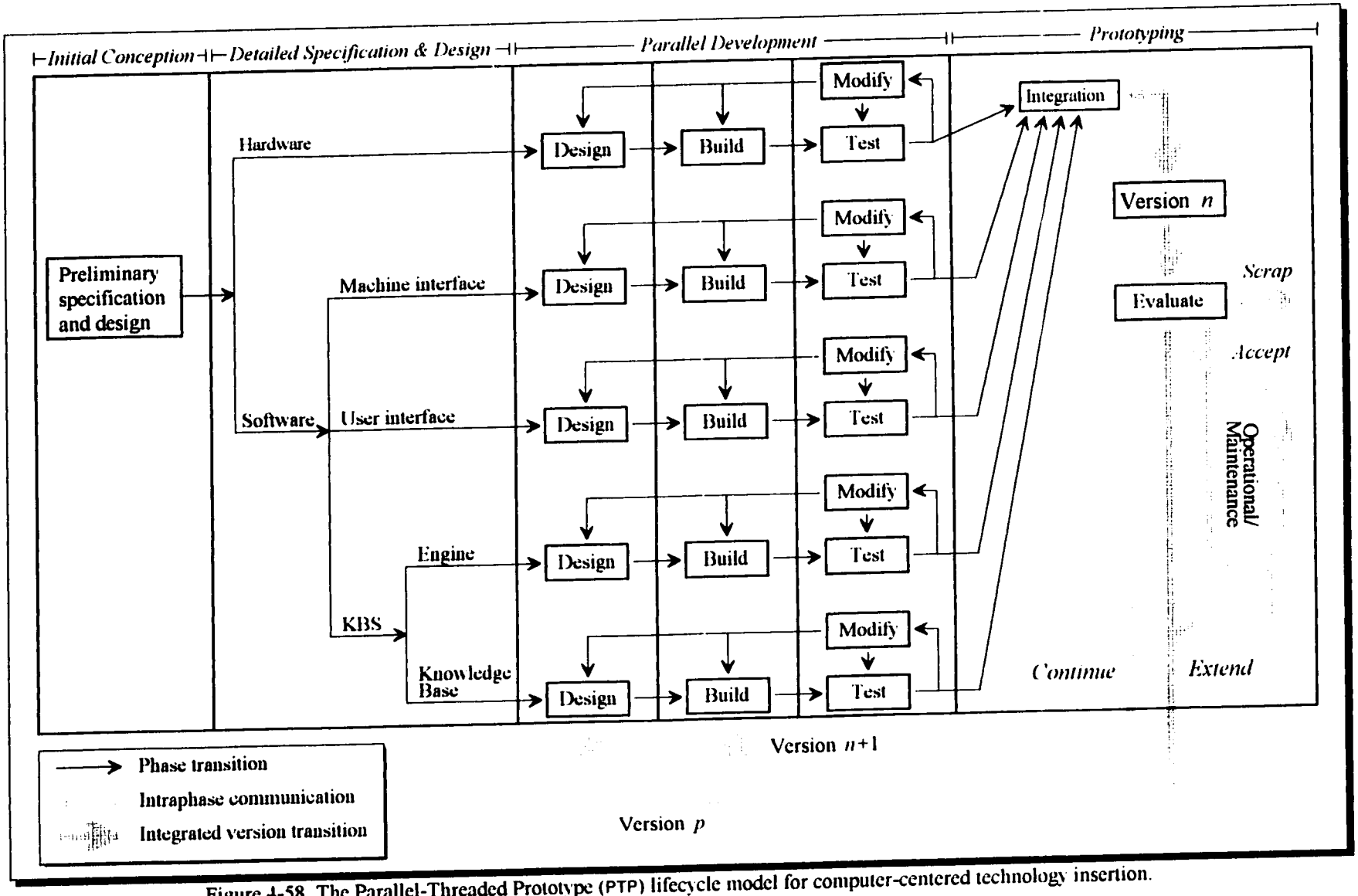


Figure 4-58. The Parallel-Threaded Prototype (PTP) lifecycle model for computer-centered technology insertion.

② Most likely, however, the evaluation will uncover bugs, deficiencies, insufficiencies or inefficiencies, and the end-users will decide to *continue* development towards the next prototype, *version* '*n+1*'.

③ If all goes well, the end-users can choose to *accept* the system and declare the project completed. The system then enters its *operational/maintenance* phase.

④ The end-users may find that the system is acceptable in its current version, but desire to *extend* it to add further features and capabilities. This outcome, which causes development to fall back to the *design* phase for *version* '*p*', can also be entered from the *operational/maintenance* phase if the end-users decide that the system should be extended.

It is obvious that this software development lifecycle model shares much in common with the models reviewed in Section 3.4, such as parallel development paths, RUDE-like iteration, and spiral prototyping. All of these features are considered valuable components of formalized software engineering methodologies, and each has found a place in the battery of principles that software developers can apply to building complex software systems. Nevertheless, as was stated in Section 3.3, each individual software development project has its own set of requirements and constraints, and, as such, any specific software development strategy may not be optimal. Indeed, software development teams evolve over time, learning which strategies to adopt, adapt and combine to yield an effective, hybrid strategy that performs to their satisfaction. In this sense, the PTP model, as applied to PACES development, has performed adequately, but is not necessarily equally applicable for all software development projects.

4.5.1.1 Development Cost

The installed hardware takes the form of a modern personal computer (PC), industry-standard embedded controller and ensemble of custom interface circuitry. The PC is the mainstay of the system, costing only approximately $3000, but providing a high level of flexible computing power. The embedded controller, costing less than $300, is designed specifically as a data acquisition and control engine. The custom interface circuitry, costing about $6000, is contrived specifically to facilitate non-invasive interface to the accelerator. Thus, on the order of $10,000 worth of hardware is required to form the basis for the control system. This figure is not only affordable but also of good value, considering that some individual instruments used on the accelerator (such as the NMR

magnetometer) cost an equivalent amount each. Of course, the highest cost is in software development, which has consumed approximately three work-years and some $60,000. Figure 4-59 illustrates this cost breakdown.

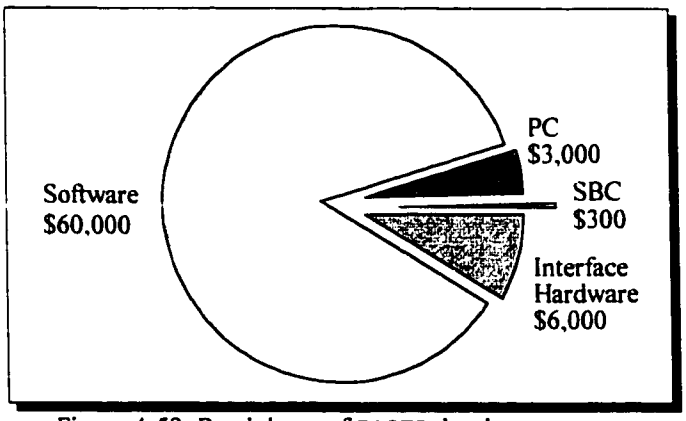


Figure 4-59. Breakdown of PACES development cost.

◆ ◆ ◆

This section has discussed the macro-scale facet of PACES software engineering, and introduced the hybridized PTP software development lifecycle model. The next sections focus on the micro-scale aspects of PACES software engineering, serving to illustrate how the RIMS programming techniques and object-oriented programming (OOP) have been applied to PACES development.

4.5.2 The Micro-scale: OOP and RIMS Applied to PACES

PACES is implemented in Borland Pascal for Windows, a programming environment that offers rich OOP capabilities and a modularization facility that promotes the RIMS techniques.

Modularity is an important quality of the PACES software. As the system has developed and evolved through several prototyping stages, the modularity of the software has ballooned considerably. PACES currently consists of some 35,000 lines of source code dispersed throughout 29 separate software modules. Of these, 15 are general-purpose software libraries, while the remaining 14 contain code specific to the PACES application. Figure 4-53 shows the hierarchy of modules that are linked to form the PACES program. At the level most removed from PACES, the Windows API is a non-object-oriented library which provides a gamut of subroutines for such things as performing GUI operations and

accessing system resources. Since the API is not object-oriented, BPW's ObjectWindows library is implemented 'on top of' the API to cast its subroutines as objects and methods. At the next level, *general-purpose units* contain subroutines and objects that perform operations not specific to PACES, such as database manipulation, serial link communications, and knowledge base inferencing. The *generic units* implement functionality common to all accelerators (such as GUI objects for selsyns, switches and meters, and SBC communication procedures), while the *site-specific units* implement functionality specific to individual accelerator sites (such as knowledge bases, GUI control panel objects, and electronic logbook manipulation). This is done to facilitate the development of a general-purpose particle accelerator control program that can be targeted for different accelerator sites in a straightforward manner by altering only the site-specific modules accordingly.

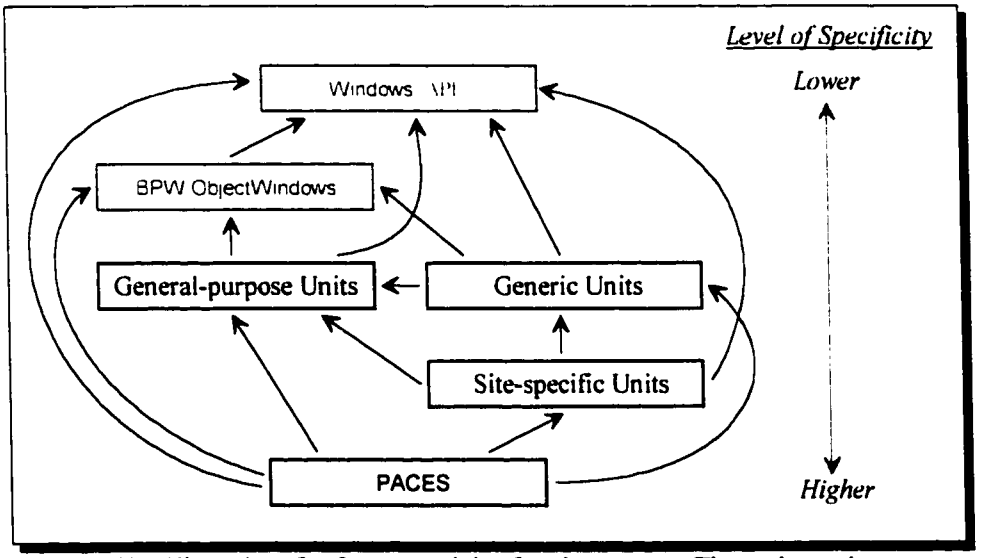


Figure 4-60. Hierarchy of software modules forming PACES. Those shown in gray are commercial products.

The modularity illustrated in Figure 4-60 enables use of the RIMS techniques of information hiding and code reusability. Different modules 'hide' information and provide objects, methods and subroutines for accessing the information, thereby imposing a degree of information integrity since access to information is tightly controlled. Additionally, the hierarchy lends itself to reuse of code because modules at higher levels of specificity can share the code of less-specific modules, and even define objects which inherit code from lower level modules. For example, the *site-specific units* reuse the code of the *generic*

units in that site-specific functionality is built upon (reuses) generic (non-specific) functionality.

Figure 4-61 delineates the inter-relationship between the PACES-specific modules. The *Main Program* module is responsible for loading the other modules and transferring control to the generic *User Interface* module. The generic *User Interface* module relies on the *KN Devices* module for rendering accelerator-specific GUI objects such as meters, selsyns and switches. The site-specific *User Interface* module implements control panel objects specific to different accelerator sites, such as the corona position controller which is present on the accelerator at McMaster but not at DREO. The *RTK* module is the SBC's real-time kernel, which is communicates with the *User Interface* module for data acquisition and control.

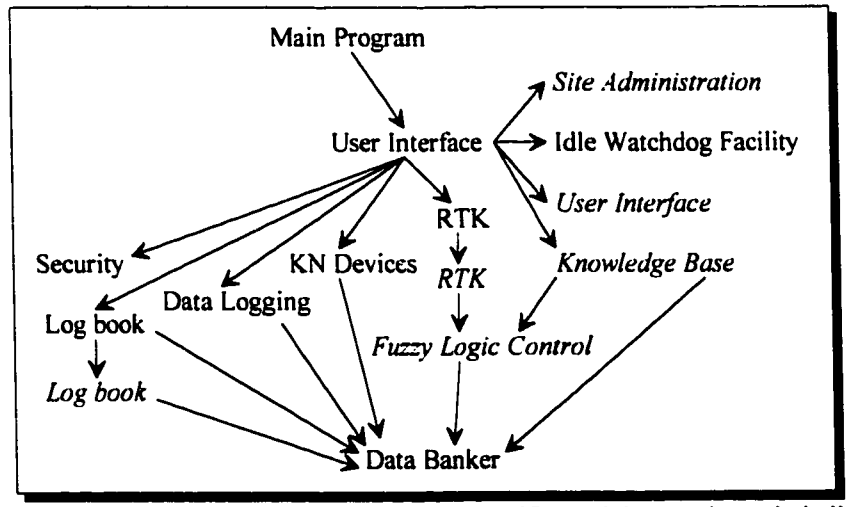


Figure 4-61. Module inter-relationships. Site-specific modules are shown in italics.

The remainder of the PACES modules are used either for automated accelerator operation, or for responding to various types of 'events' (or for supporting other modules in this task). Since all Windows applications are 'event-driven', they consist of a collection of sub-components that are executed depending on 'events' that occur. In this context, an *event* is either a user input action such as a keypress or mouse movement, or a system event such as a timer expiration event or serial communications event. Typically, the user will click a button on the GUI, causing an event which activates, for example, the KN's drive motor or initiates automated start-up. Much of the user-oriented event handling is performed in the generic and site-specific *User Interface* modules, but several other

modules also respond to user input actions. The *Data Banker* module is used to co-ordinate access of accelerator state variables, and all accesses to these variables are made through the *Data Banker* module's subroutines. The *Security* module is responsible for access authorization, and is utilized whenever an operator logs in or out of the system. The *Log Book* module is invoked whenever the user opens the electronic log book which is used for recording accelerator control settings for later duplication of accelerator state. The *Site Administration* module is concerned with accelerator site management, and is site-specific in its duties.

The rest of the modules are used for handling system events, such as timer expirations or communications events. These modules therefore operate in the background, without necessarily requiring user interaction. The *Data Logging* module is used to log telemetric accelerator data to disk for later off-line analysis. The *Idle Watchdog Facility* module monitors operator inactivity (of the keyboard and mouse), and initiates accelerator safing action if the operator has been inactive for a specific timespan (q.v. § 4.1.4.2). Finally, the *Knowledge Base* and *Fuzzy Logic Control* modules are used for decision making during automated start-up, shut-down and steady-state operation (q.v. § 4.3).

PACES is highly object-oriented, with objects and object hierarchies used to implement almost all aspects of the program. This makes the system flexible to revision and expansion, and facilitates porting of the system to different accelerator sites. For example, the GUI is completely object-oriented, and is composed of hierarchies of objects which implement the behaviour of the user interface. Figure 4-62 shows how the GUI is organized. The main object is the *control panel*, the GUI facsimile of the real control panel. This object is responsible for creating the subservient objects of the GUI which comprise the components of the user interface. The user interface objects include the meters, selsyns, switches and support tools present on the control panels at all accelerator sites (generic objects) and specialized tools which may differ from site to site (generic or site-specific objects). Since the site-specific control panel object is a descendant of the generic control panel object, it inherits its parent object's generic behaviour, and extends this behaviour with site-specific objects.

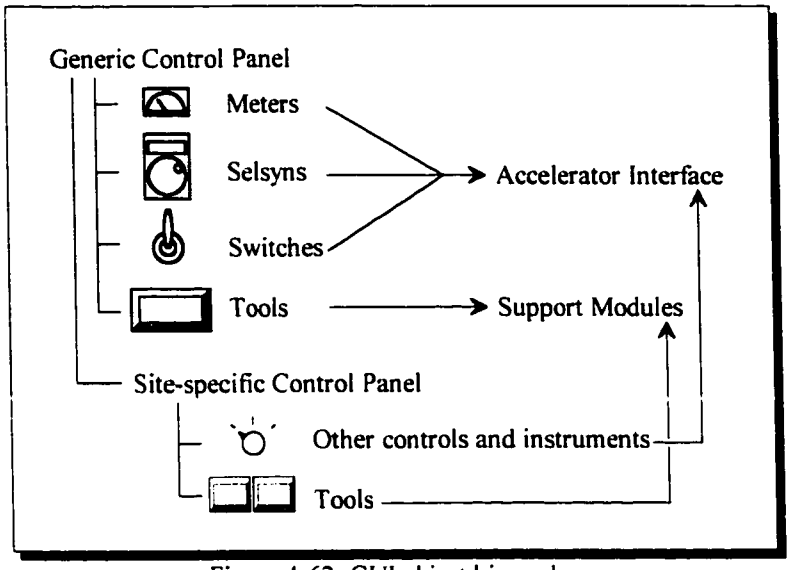


Figure 4-62. GUI object hierarchy.

The generic objects, and the modules they reside in, form a generic basis for the accelerator control system, and only a relatively smaller amount of site-specific objects and modules need to be added on top of this base to implement a site-specific system. Thus, it is evident that this arrangement makes for a highly modular, object-oriented system architecture which possesses the desirable RIMS properties, and leads to a control system which is both powerful (through genericity) and flexible (through site-specificity).

* * *

This chapter has described the design and implementation of PACES in considerable detail, elaborating on aspects of the system's overall development that are relevant to the theme of this thesis. The next chapter presents some analysis of the expert system's performance during automated accelerator operation.

Chapter 5

Autonomous Performance

This chapter details performance of PACES' knowledge base during automated accelerator operation. Time-series plots of accelerator operating parameters will be presented and discussed to illustrate how the expert system performs during accelerator start-up, conditioning and beam maintenance. The performance of the fuzzy logic-based terminal voltage set-point controller will also be presented and compared with the expert system's voltage set-point pilot.

The time-series plots which figure prominently in this chapter are generated off-line from telemetric data logged to disk during accelerator operation. Each plot shows how several accelerator parameters vary over time. By plotting selsyn position values (control outputs) simultaneously with analog meter values (control inputs), it is possible to demonstrate how the accelerator responds to various forms of automated control.

5.1 Expert System Performance

As described in Chapter 4, the expert system is responsible for automated accelerator start-up, beam maintenance, and shut-down. The start-up knowledge base is required to start the accelerator from an inoperative state and establish the particle beam within specified tolerance intervals for parameters such as terminal voltage and target beam current. The beam maintenance knowledge base is charged with maintaining the particle beam within specified tolerance intervals and engaging in beam recover whenever the beam is lost from target. Finally, the shut-down knowledge base performs accelerator deactivation. Due to its low level of complexity, the shut-down knowledge base will not be discussed further. Instead, what follows is a discussion of the performance of the start-up and beam maintenance knowledge bases.

5.1.1 *Automated Start-up*

Figure 5-1 shows some of what occurs during an automated accelerator run in which the expert system performs accelerator start-up (cf. Figure 4-31). There are two goals for the start-up procedure: The first is to reach a specified terminal voltage of 1.0MV.[66] Once the target terminal voltage is reached, the second goal of acquiring a strike (generating a particle beam that strikes the first Faraday cup) comes into effect. For this specific accelerator run, the gas and focus selsyns were set to positions used in a previous run, so only the belt charge and extraction selsyns were manipulated by the control system.

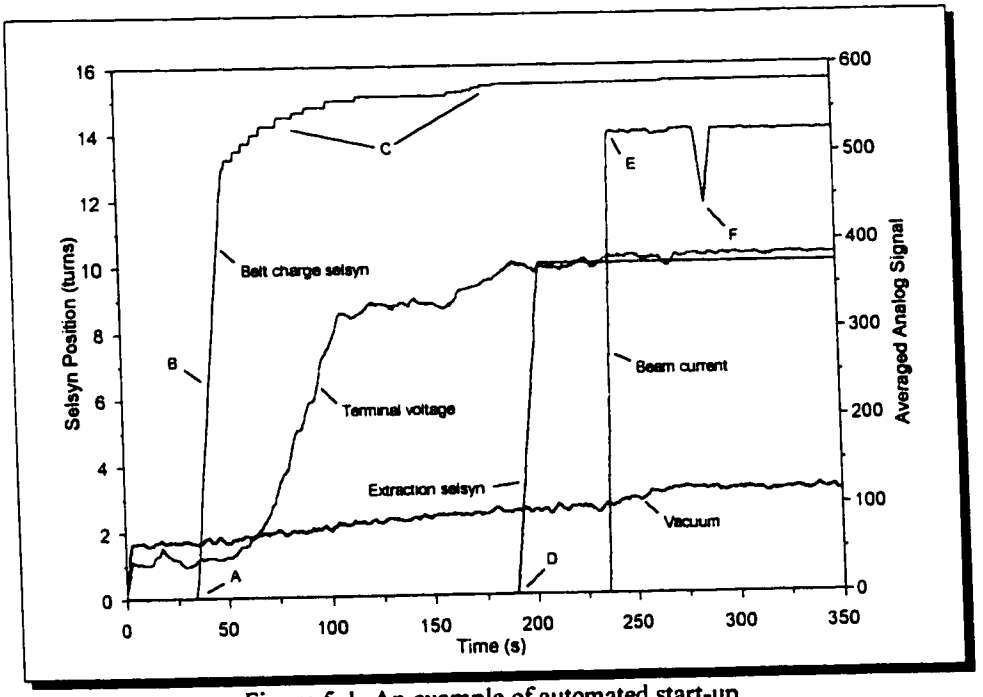


Figure 5-1. An example of automated start-up.

The control power is switched on at time 0s, and the terminal voltage assumes its uncharged baseline value of approximately 250kV. The start-up process then proceeds to verify several safety interlocks (e.g. cooling water turned on, tank SF_6 pressure within acceptable limits). Next, the van de Graaff drive motor is switched on, followed by the belt charge power supply. Automatic control begins at A, when the control system increases the belt charge selsyn to achieve the specified terminal voltage of 1MV. A fact stored in the knowledge base indicates that there is a 'dead zone' in the belt charge selsyn, and the first 11 full turns will have no effect on the terminal voltage. The expert system therefore quickly turns the selsyn through 11 turns (B), and then begins making small stepwise

[66] This terminal voltage is low enough that conditioning is not needed for stable operation.

increases of the selsyn (C) to bring up the terminal voltage to its target level. The expert system then begins increasing the extraction selsyn (D) in an attempt to generate a strong particle beam. Beam intensity occurs at E, marked by a pronounced jump in beam current. At this point, the start-up sequence terminates successfully, and beam maintenance mode is entered.[67]

In the example shown, automated start-up took about six minutes, but generally the time required for start-up depends on many factors (such as voltage stability, quality of vacuum, previous control settings and condition of the ion source) and can vary from run to run.

5.1.2 Conditioning

As stated in Chapter 4, the expert system performs voltage conditioning to 'condition' the accelerator gradually to increasingly higher terminal voltages. Conditioning is typically necessary when operating at terminal voltages near the accelerator's maximum rating, after the machine's tank has been opened for maintenance, or whenever the accelerator exhibits periods of voltage instability.

'Warm' conditioning is performed during an ongoing accelerator run when the machine encounters a period of voltage instability while in 'cruise control' mode, or when starting up the accelerator after a relatively short period of inactivity (e.g. overnight, or after a brief shut-down to change samples in the target chamber).

As illustrated in Figure 5-2, during warm conditioning (cf. Figure 4-34) the belt charge selsyn is repeatedly adjusted in a step-wise manner (A), with the expert system pausing after selsyn adjustments to allow the accelerator time to 'condition' itself to the higher terminal voltage (B). As long as the terminal voltage appears stable (that is, there is no sparking or excessive voltage ripple), the expert system gradually increases the terminal voltage. Usually at some point, however, the terminal voltage becomes unstable or a spark occurs (C). The expert system detects this instability and backtracks by decreasing the belt charge selsyn to recover stability at a lower terminal voltage (D). More pronounced episodes of instability (F,H) require greater amounts of backtracking to recover stability

[67] The sharp drop in beam current occurring at F may indicate beam sputtering due to an initial, short-lived instability in the plasma or source gas flow.

(G,I). Once the instability subsides, the expert system resumes its stepwise belt charge selsyn increases (E,J). Eventually, when the terminal voltage is stable and within tolerance of its set-point, the conditioning operation terminates, and inferencing switches to the Auto-Pilot knowledge base.

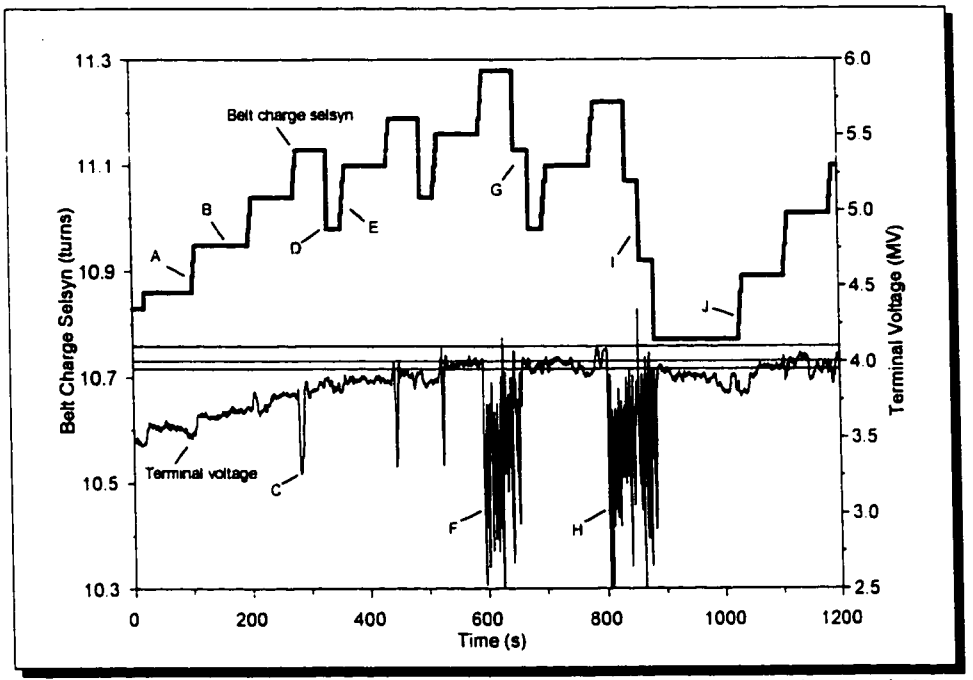


Figure 5-2. Portion of a 'warm' terminal voltage conditioning operation carried out during start-up after a brief period of shut-down.

'Cold' conditioning, in contrast, is performed when the accelerator has been started from a 'cold' state, such as after a weekend of inactivity, or after maintenance, and is therefore expected to exhibit a substantial initial period of voltage instability, especially at terminal voltages near the accelerator's upper limit. The difference between cold and warm conditioning is that in cold conditioning, the accelerator is deliberately over-conditioned to a higher terminal voltage than is needed for the run so as to provide a buffer zone of voltage stability; by conditioning the accelerator to stability at a higher voltage, stability at lower voltages is more reliable and longer lasting. Cold conditioning commences immediately after the accelerator has been started and the terminal voltage has been brought to a baseline level of approximately 3.0MV. Cold conditioning, like warm conditioning, involves repeatedly increasing the belt charge selsyn in a step-wise manner, with the expert system waiting out periods of voltage instability. When the specified upper limit for conditioning is reached and the voltage is stable, the conditioning operation

terminates successfully, and the voltage set-point pilot is executed to bring the terminal voltage down to the level required for the accelerator run. When the target terminal voltage is reached and is stable, the conditioning knowledge base terminates, and inferencing switches to the Auto-Pilot knowledge base.

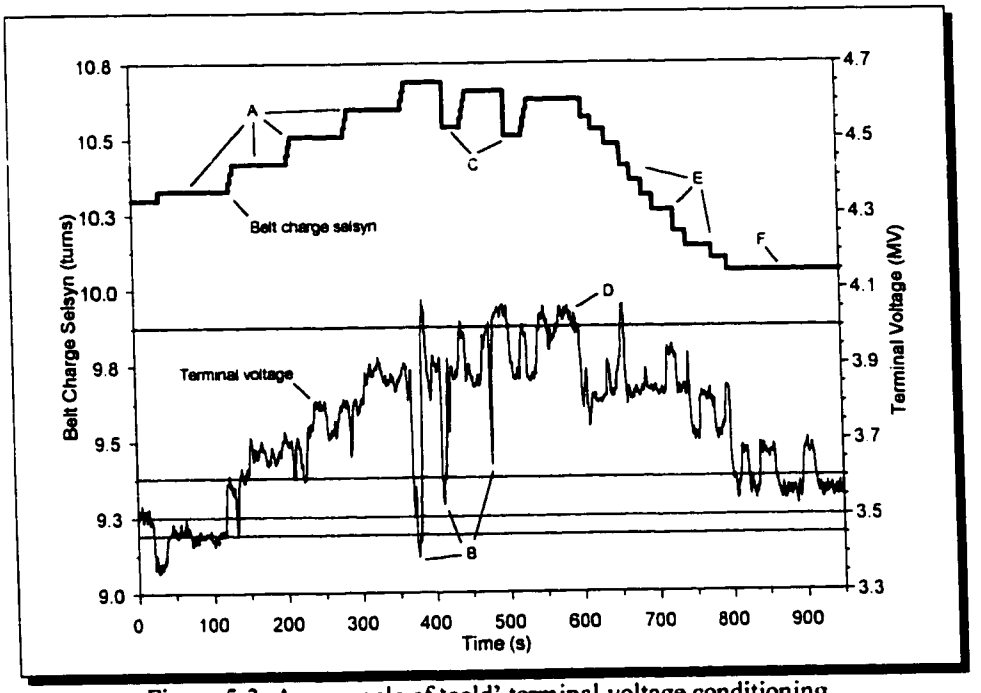


Figure 5-3. An example of 'cold' terminal voltage conditioning.

Figure 5-3 shows an example of cold conditioning for a target terminal voltage of 3.5MV. In the figure, the target terminal voltage is shown by the green line, the tolerance interval of [+100kV,-50kV] is indicated by the red lines, and the upper limit for conditioning is marked by the blue line. The belt charge selsyn is repeatedly increased to increase the terminal voltage through tiers of stability (A). When the voltage level becomes unstable or sparking occurs (B), the expert system backtracks to stability at a lower terminal voltage tier (C). When the upper limit for conditioning is reached and the voltage is stable (D), conditioning terminates successfully, and the voltage set-point pilot is executed to bring the terminal voltage down to the level required for the accelerator run (E). When the terminal voltage is brought to stability within tolerance of the required level (F), the conditioning knowledge base terminates, and the Auto-Pilot knowledge base is activated.

5.1.3 Beam Maintenance

When the accelerator is being operated in automatic mode — that is, when PACES is in control of the accelerator during the beam maintenance phase — the expert system is required to maintain the particle beam on target within specified tolerance intervals. As described in Chapter 4, the beam maintenance knowledge base comprises several threads which are inferenced in parallel: The Auto-Pilot is the principal knowledge base thread responsible for maintaining the particle beam on target, and several other 'helper threads' are charged individually with maintaining various accelerator parameters (such as corona current, source gas flow, and belt charge) within tolerance.

5.1.3.1 Auto-Pilot

The Auto-Pilot employs the voltage stabilizer hardware subsystem for maintaining the terminal voltage (and, therefore, particle energy) at set-point — as long as the voltage stabilizer is able to maintain the terminal voltage at set-point, the Auto-Pilot remains essentially idle. When the voltage stabilizer loses control of the terminal voltage (usually after a spark), the Auto-Pilot must be able to detect the loss of control and enact recovery in order to restore the particle beam to target with as little delay (lost beam time) as possible.

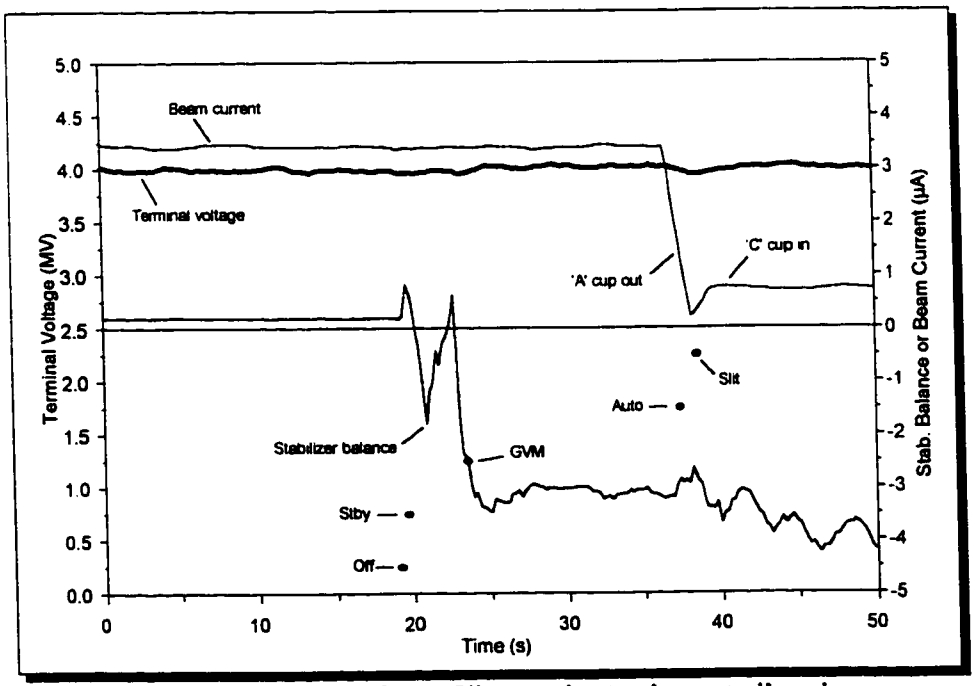


Figure 5-4. Example of Auto-Pilot entering 'cruise-control' mode.

Figure 5-4 shows what occurs as the Auto-Pilot is activated and takes control of the accelerator (cf. Figure 4-36). Initially, the terminal voltage has been brought to within tolerance of its set-point by the voltage set-point pilot (maybe after conditioning), and a particle beam is being detected on one of the Faraday cups (typically 'A' or 'B' cup[68]). The Auto-Pilot first performs several checks to ensure that the accelerator is ready for 'cruise control' mode (e.g. terminal voltage stable and within tolerance, beam current detected on Faraday cup, sample power within tolerance), and then switches the voltage stabilizer mode from Off to Stby and then to Slit[69], and enters its 'stable beam' loop.

Figure 5-4 shows that as the voltage stabilizer's mode is switched from Off to Slit, the stabilizer balance signal shifts from being steady and near-zero to varying and negative, indicating that the voltage stabilizer has assumed active control of the terminal voltage. This example also shows that 'C' cup has been left inserted into the beam line (for on-demand beam sampling), but it is more typical for the expert system to pull out all Faraday cups and monitor the beam indirectly using the sample power or stabilizer balance signals (as described below).

The voltage stabilizer is able to maintain the terminal voltage at set-point for long periods of time (sometimes hours) by compensating for minor fluctuations in terminal voltage. But, occasionally (and depending on the relative level of the terminal voltage), large terminal voltage fluctuations occur for which the voltage stabilizer is unable to compensate. This occurs because as the particle energy shifts too far off set-point, the analyzing magnet fails to bend the particle beam properly, and the beam fails to strike the energy slits, breaking the feedback path required for terminal voltage control (cf. Figure 2-7). The net effect of such a large voltage fluctuation is that the beam gets lost from target, and the voltage stabilizer is unable to recover the beam.

The beam maintenance knowledge base can detect beam loss in three different ways:

① *Sample power*: The sample power signal is monitored to detect indirectly whether the beam is on target. The sample power signal is low when the beam is irradiating the

[68] The KN-4000 at AECL's Whiteshell Labs has three Faraday cups: 'A' is upstream of the analyzing magnet; 'B' is downstream of the analyzing magnet, steerers and quadrupoles; and 'C' is located immediately upstream of the sample chamber.

[69] This assumes that the beam is to be bent through the analyzing magnet. If the magnet is not being used, the voltage stabilizer gets switched to GVM mode.

target, and high when the beam is not on target. The Auto-Pilot can monitor the sample power signal and infer a loss of beam from a jump in the sample power signal. This is the best method for detecting beam loss (because the beam does not need to be disturbed to detect its presence), but is not always available depending on the features of the target chamber which vary from experiment to experiment.

② *Stabilizer balance*: When the voltage stabilizer is controlling the terminal voltage, the stabilizer balance signal shows a relatively high amount of variance; when stability is lost, the stabilizer balance signal has low variance. It is therefore possible to infer loss of beam indirectly by detecting a significant change in the variance of the sample power signal. This effect is captured in Figure 5-4 above, which shows how the stabilizer balance signal changes from being low-variance and near-zero to high-variance and negative when the voltage stabilizer acquires active control after being switched to GVM mode.

③ *Beam current sampling*: The expert system can 'sample' the beam by inserting the 'C' cup into the beam line and measuring the beam current for a short time. The beam's presence on target is directly ascertainable using this method, but the beam is disrupted from target, which may not be tolerable in certain experiments.

Whatever method is used, the expert system enters its 'beam recovery' mode when it determines that the beam has been lost from target. Figure 5-5 shows an instance of automatic beam recovery in which the sample power is monitored to detect beam loss. Initially (A) the terminal voltage is stable at approximately 4MV, and the sample power (B) is low (approximately 10W). A spark occurs at time 0s (C), which causes the voltage stabilizer to lose control, and the beam is lost from target. As a result, the sample power quickly jumps to a high level (D), indicating that the beam has indeed been lost from target. A short time later, the expert system initiates recovery mode (cf. § 4.3.6) by switching the voltage stabilizer to Off (E). The terminal voltage shortly recovers to its set-point (F), and the expert system re-activates the voltage stabilizer, gradually switching it from Off to Slit (G), all the while checking that the voltage is stable. Once the voltage stabilizer regains control, the expert system pulls out the 'C' cup so that the beam strikes the target, and the sample power drops back to its previous level (H), indicating that the

beam is on target. At this point, beam recovery has succeeded, and the expert system switches back to 'stable beam' mode. In this example, the beam was lost from the target for 61.4 seconds.

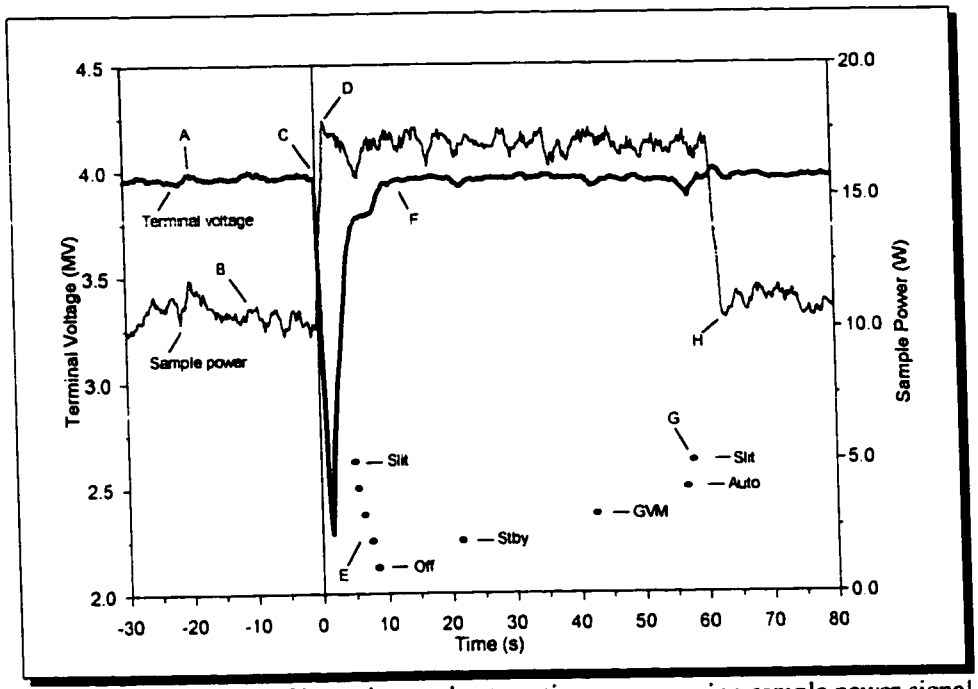


Figure 5-5. Example of beam loss and automatic recovery, using sample power signal to detect loss of beam.

In comparison, Figure 5-6 shows an example of beam recovery in which beam loss was detected from a change in the variance of the stabilizer balance signal. Initially, approximately 25s prior to beam loss, the terminal voltage enters a period of instability (A) punctuated by three sparks. This instability proves too much for the voltage stabilizer to handle, and the beam gets lost at 0s (B). The expert system determines that the variance of the stabilizer balance signal has changed significantly and concludes that the stabilizer has lost control. The stabilizer is immediately switched to Off and inferencing shifts into recovery mode. Although the terminal voltage continues to fluctuate (unlike the previous example), the expert system is still able to re-activate the voltage stabilizer (C), the terminal voltage settles back to its set-point under control (D) with the beam on target, and the expert system switches back to 'stable beam' mode. In this example, the beam was lost from target for 32.1 seconds.

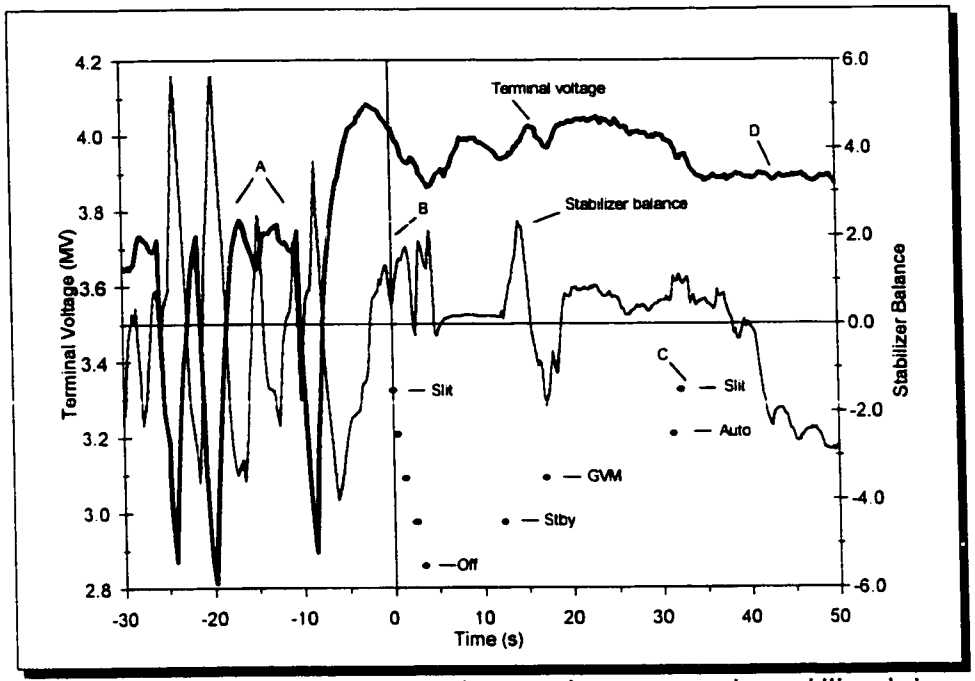


Figure 5-6. Example of beam loss and automatic recovery, using stabilizer balance signal to detect loss of beam.

A comparison of the recovery times for six instances of beam loss is presented in Figure 5-7. Recovery time during these instances of beam loss averages 54.3s.[70] This indicates that, usually, the expert system is able to recover the beam within one minute of beam loss, but this time can be affected by such factors as terminal voltage stability, quality of system vacuum, and the vitality of the accelerator's ion source.

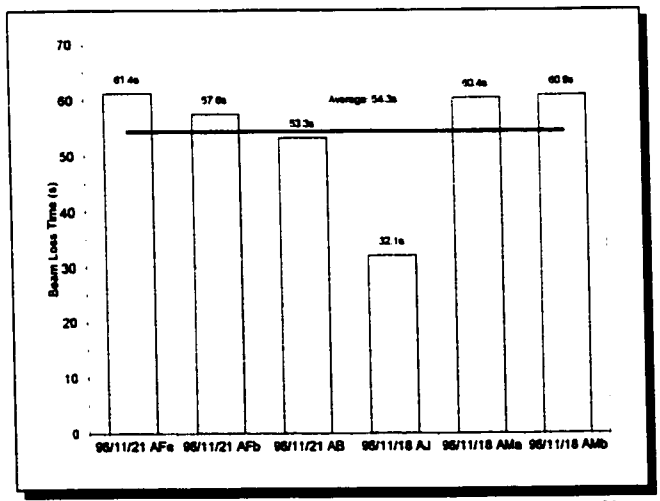


Figure 5-7. Comparison of recovery times for six instances of beam loss.

[70] If the beam loss time for instance #4 is discounted as an unusually rapid recovery, the average beam loss time is 58.7s.

5.1.3.2 Corona Optimization

As described in Chapter 4 (§ 4.3.6.2), one of the beam maintenance knowledge base's helper threads is responsible for ensuring that the corona current remains within its prescribed tolerance interval. Figure 5-8 illustrates the corona current optimization thread performing an automatic adjustment to maintain the corona current within the specified tolerance interval: The 'optimum'[71] corona current of 50μA is indicated by the green line, and the lower and upper limits (of 30μA and 80μA, respectively) are indicated by the red lines; the corona current signal is shown in blue. Prior to approximately 0s, the corona current is oscillating within the tolerance interval (A), but breaches the upper tolerance limit at 0s (B), triggering the corona current optimization thread to perform a control adjustment of the belt charge selsyn (C). After several decreases of the belt charge selsyn, the corona current drops back into its tolerance interval and approaches the optimum value (D), and the corona current optimization thread ceases control adjustment. In this example, approximately 30 seconds were required to adjust the corona current.

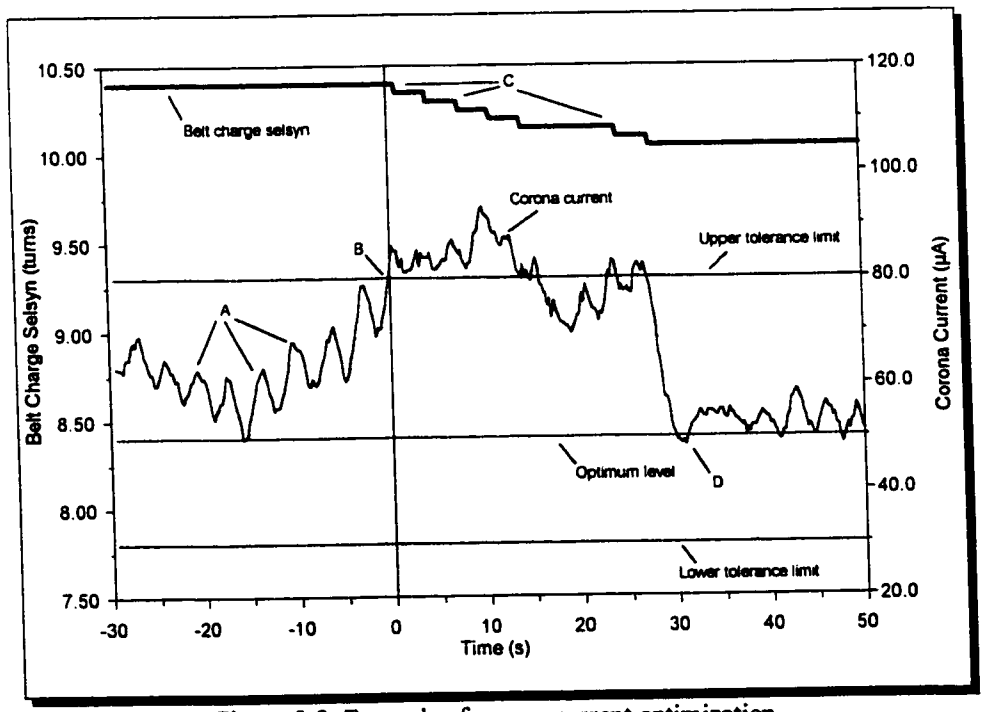


Figure 5-8. Example of corona current optimization.

5.1.3.3 Gas Optimization

The gas optimization helper thread is used by the beam maintenance knowledge base for adjusting the source gas flow rate in an effort to maintain particle beam stability by

[71] 'Optimum' in this case means 'optimum as prescribed by the operator'.

maintaining the accelerator's vacuum at an 'optimum'[72] level. It was mentioned in Chapter 4 (§ 4.3.6.5) that the gas optimization problem is complicated by the slow slew rate of source gas flow in response to gas selsyn adjustments. Figure 5-9 presents an example of slow gas slew rate, showing how the vacuum level (an indirect gauge of source gas flow) changes in response to gas selsyn adjustments. As shown in Figure 5-9, a two-turn increase of the gas selsyn (A) results in a slow, gradual rise of the vacuum level (B) which lasts approximately 150s (2.5 minutes). When the gas selsyn is decreased by 3.15 turns (C), the vacuum level does not show any significant response for approximately 38s (D). It is obvious that the gas slew rate is quite slow in comparison to the terminal voltage slew rate (which is about 3s); this slow slew rate consequently makes heuristic-based control slow and clumsy.

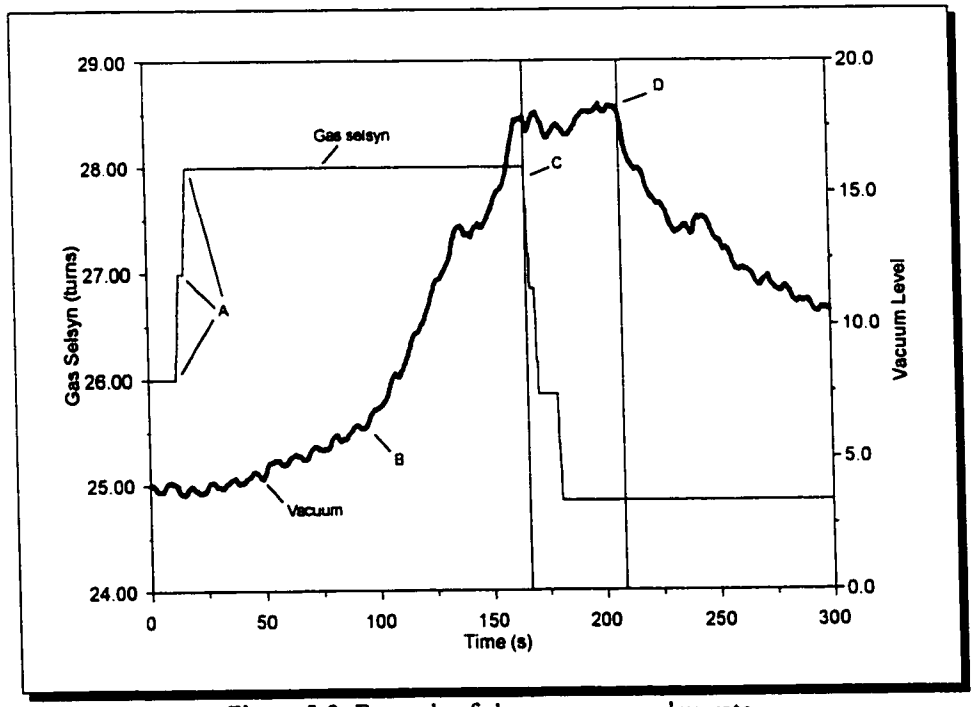


Figure 5-9. Example of slow source gas slew rate.

The gas optimization helper thread presented in Chapter 4 (§ 4.3.6.5) incorporates a long time lag (5 minutes) in order to accommodate the slow gas slew rate, and performs gas selsyn adjustments when the vacuum level deviates from the specified tolerance interval. Figure 5-10 plots an instance of gas optimization: The vacuum level is hovering about the lower tolerance limit, and slips below this lower bound long enough to be

[72] 'Optimum' in this case means 'optimum as prescribed by the operator'.

detected by the knowledge base (A). The gas optimization thread gradually increases the gas selsyn to raise the vacuum level towards 'optimum'[73] (B), and waits for 5 minutes after gas selsyn adjustment (C) to allow the gas flow rate to respond. During this time, the vacuum level slowly drifts up towards optimum (D) and then settles near optimum (E). This response satisfies the termination criterion for the gas optimization thread, and gas selsyn adjustment is ceased until the next time the vacuum level drifts out of tolerance. In this example, approximately 12 minutes were required to make the gas flow adjustment.

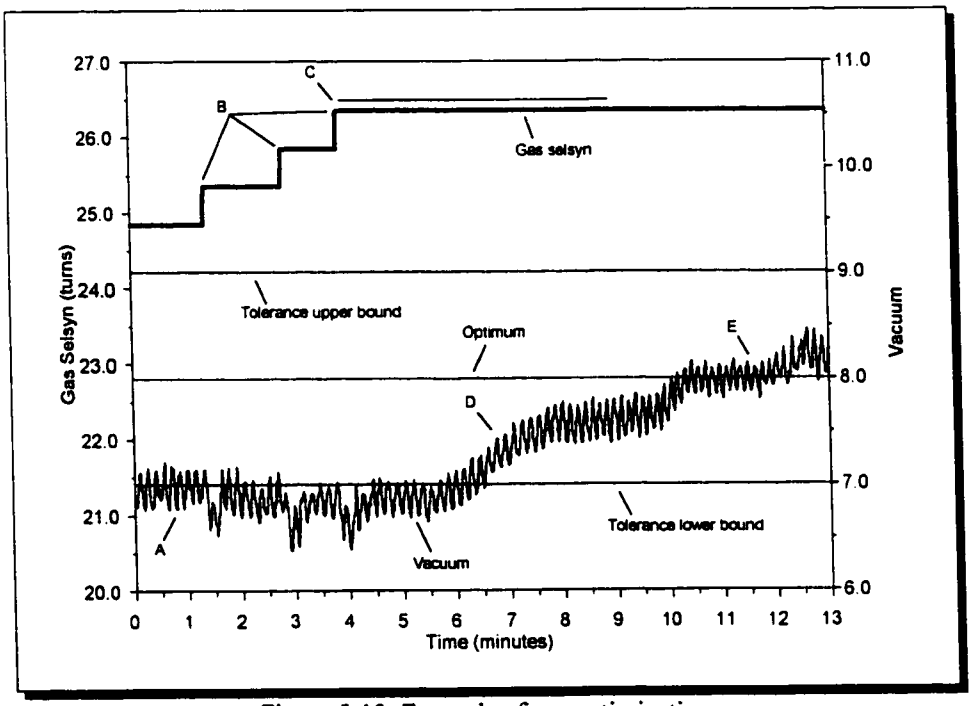


Figure 5-10. Example of gas optimization.

5.1.4 Fuzzy Control of Terminal Voltage Set-point

This section presents the results of an evaluation of the application of fuzzy logic-based controllers to terminal voltage control. As explained in Chapter 4 (§ 4.3.8), three versions of the belt charge FLC were evaluated and compared with both manual and expert system-based belt charge control. The test case (goal) for evaluating the FLC was a target terminal voltage of 1.0MV during start-up of the KN-3000 at McMaster. A tolerance window of [+100kV,-50kV][74] was specified, so any stable terminal voltage within the range 0.95MV to 1.1MV was sufficient to satisfy the goal (set-point). In the

[73] 'Optimum' in this case means 'optimum as prescribed by the operator'.

[74] This coarse error margin is acceptable because the accelerator's built-in voltage stabilizer is responsible for finer control of terminal voltage set-point.

following figures, the target terminal voltage is indicated by a green line, and the tolerance window by two red lines. Note, however, that no tolerance window was specified for the manual set-point acquisition trial.

As a basis for comparison, both a human operator and the existing expert system were used to start-up the accelerator to its target voltage. Figure 5-11 presents the data logged during a manual start-up. The operator's 'control loop' starts at 172s, and the goal state is reached at 300s, for an elapsed time of 128s. No less than 20 belt charge selsyn adjustments were performed to reach set-point.[75] This manual start-up was performed by a novice operator, and can be considered an upper bound on manual set-point acquisition.

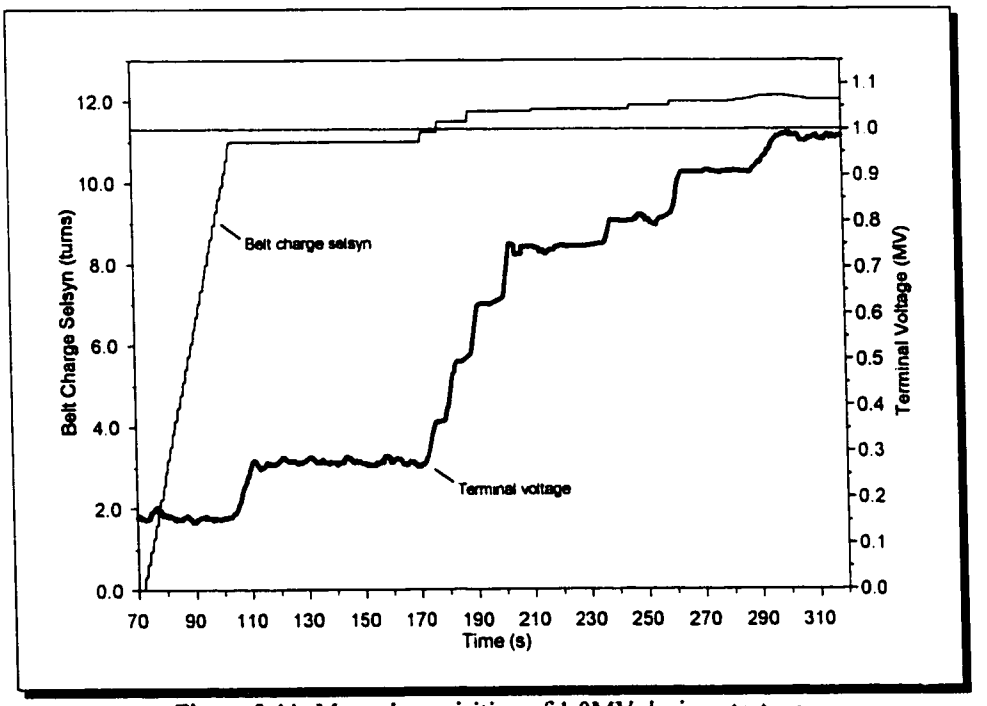


Figure 5-11. Manual acquisition of 1.0MV during start-up.

Figure 5-12 shows the expert system's voltage set-point pilot (q.v. § 4.3.4.2) being used to adjust the belt charge selsyn until the target terminal voltage is reached. In total, 6 belt charge selsyn adjustments were made. The voltage set-point pilot took approximately 33s to reach its goal state. There was no overshoot during terminal voltage set-point acquisition.

[75] The nine vernier decreases of the belt charge selsyn performed after 300s could be considered overshoot compensation except that no overshoot in terminal voltage is observable.

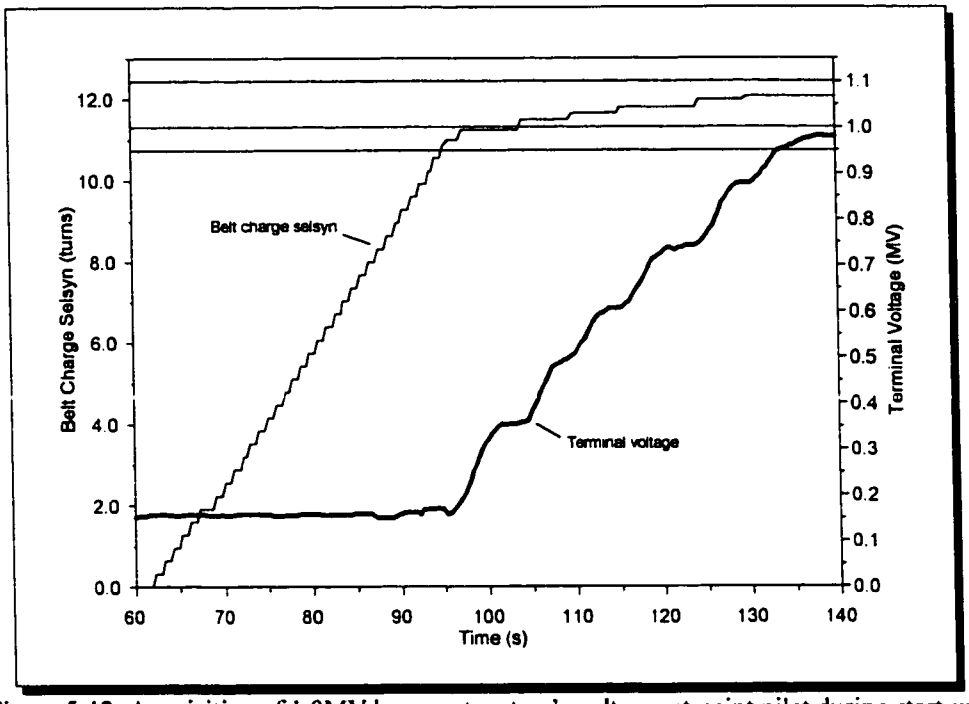


Figure 5-12. Acquisition of 1.0MV by expert system's voltage set-point pilot during start-up.

The first version of the belt charge FLC was 'executed' in a control loop with a fixed delay of 3.5s to accommodate the terminal voltage slew rate. As illustrated in Figure 5-13, this FLC required 28s to reach set-point, and performed 6 belt charge selsyn adjustments. No overshoot was observed.

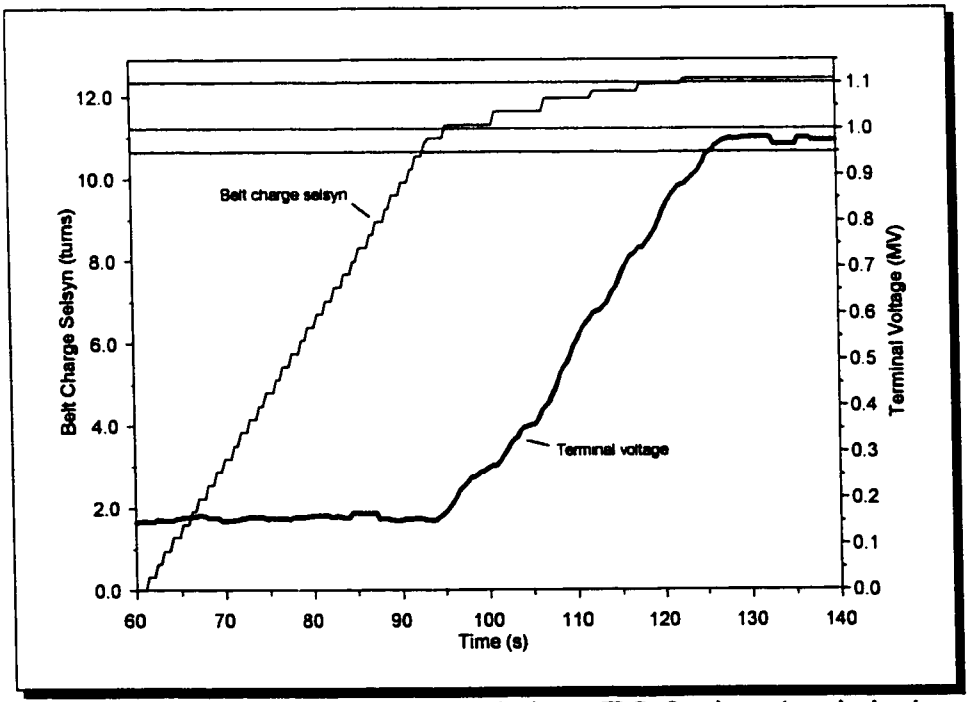


Figure 5-13. Performance of first version belt charge FLC: One input (terminal voltage error) and 3.5s delay between control iterations.

The second version FLC employed a control loop delay dependent on the derivative of the terminal voltage, which resulted in improved performance. The modified FLC (Figure 5-14) took 23s to reach the required terminal voltage, performed 5 selsyn adjustments, and did not exhibit any overshoot.

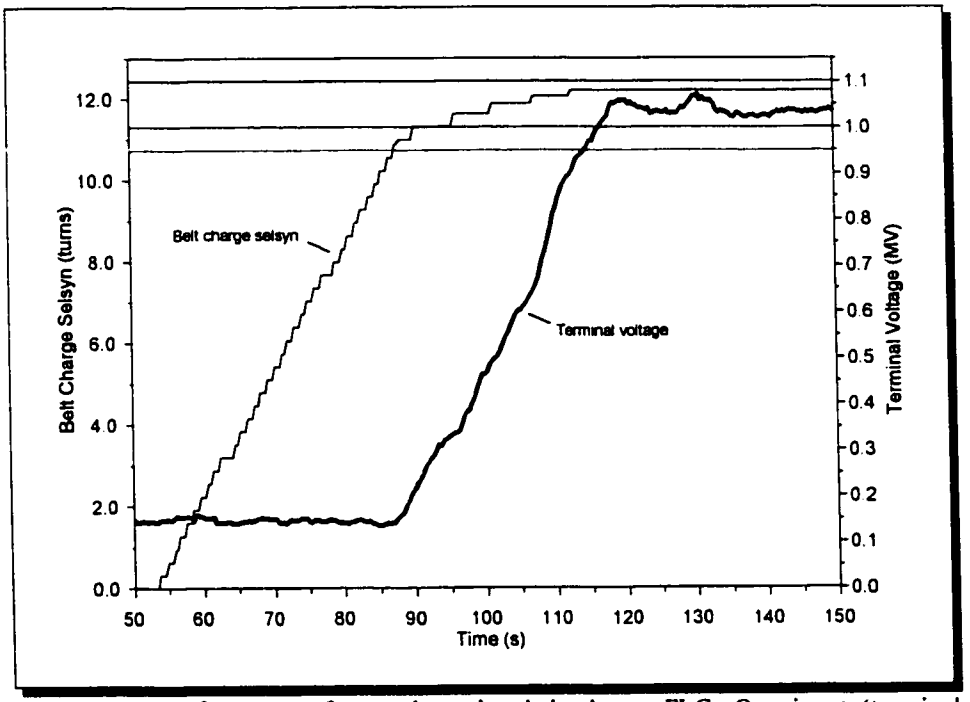


Figure 5-14. Performance of second version belt charge FLC: One input (terminal voltage error) and derivative-dependent delay between control iterations.

The third version FLC involved inclusion of a second input variable, the derivative of the terminal voltage. As shown in Figure 5-15, this version spent 26s reaching the set-point, executed 10 selsyn adjustments, and caused an overshoot of 0.11MV; four selsyn adjustments were applied for overshoot compensation. This performance suggests that the two-input FLC is potentially the fastest at reaching the set-point, but that fine-tuning of its FAM is needed to eliminate overshoot.

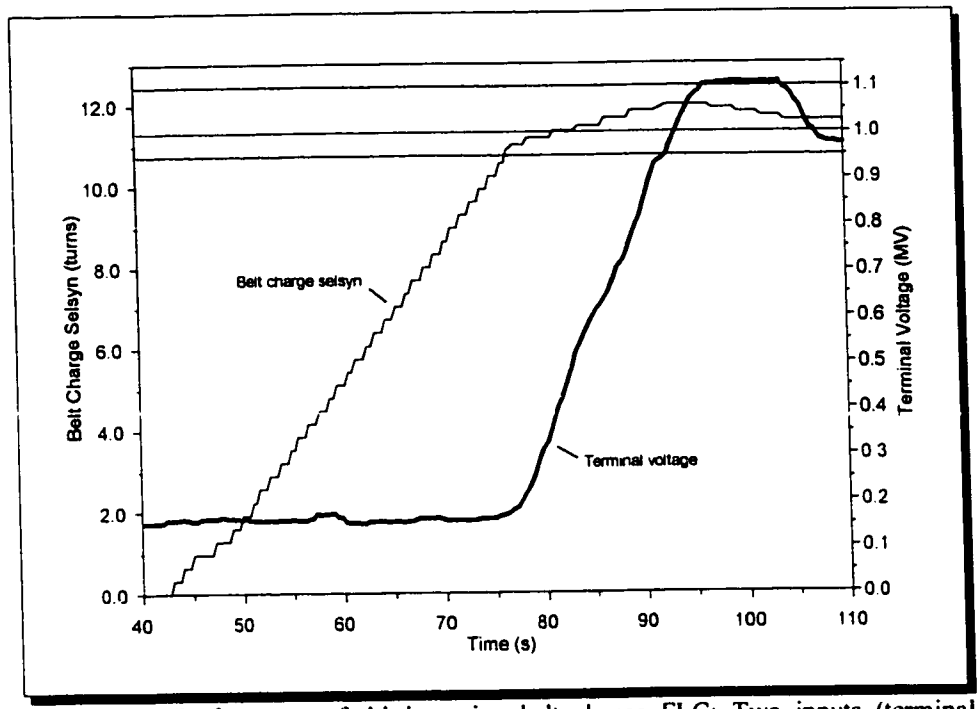


Figure 5-15. Performance of third version belt charge FLC: Two inputs (terminal voltage error and derivative of terminal voltage error) and 1s delay between control iterations.

Table 5-1 summarizes the results of this study. The constant-delay single-input FLC (version one) required less time to reach the target voltage than the expert system's voltage set-point pilot. The variable-delay FLC (version two) also required less time, and executed fewer control actions than the expert system. The two-input FLC (version three) was second fastest to reach the target voltage, but caused an overshoot of 10% and expended 4 selsyn adjustments to compensate.

Controller	Time (s)	Control Actions	Overshoot
Manual	128	20	Possible
Expert system voltage set-point pilot (variable delay)	33	6	No
Constant-delay single-input FLC (version 1)	28	6	No
Variable-delay single-input FLC (version 2)	23	5	No
Constant-delay two-input FLC (version 3)	26	10	Yes

Table 5-1. Summary of results comparing belt charge FLCs with manual and expert system-based terminal voltage control.

* * *

This chapter has described the performance of PACES' artificial intelligence-based automation mechanisms. The results presented show that the expert system and fuzzy

logic-based controllers are capable of performing autonomous accelerator start-up and beam maintenance operations with accuracy and response times comparable to manual operation.

The following chapter discusses considerations involved in generalization of the computer-centered technology insertion (CCTI) approach which was used in the PACES project. The PTP development lifecycle model will figure prominently in this discussion, serving as a foundation and backdrop for the many diverse aspects of the CCTI operation which must be properly considered and developed in detail in order to produce a computer system which both successfully performs the tasks required of it and is ultimately accepted by the people who use it.

Chapter 6

Discussion

The concept of technology insertion introduced in Chapter 1 is concerned with modernizing and upgrading existing human-machine systems through the addition of computer-centered hardware and software in an effort to increase the useful lifespan, efficiency and capability of these systems. Because humans and machines are radically different entities, many of the things that humans are good at are usually things that machines are not good at, and vice versa. A good illustrative comparison of the differences between humans and machines is supplied by Siddall, ([Sid94], § 1, p. 6):

> "The human has intelligence, foresight, creative ability, diagnostic ability and extreme versatility in all respects. On the negative side, the human can get bored, distracted, confused or frightened, and needs to eat and to spend time in the cafeteria and washroom, and is commonly burdened with incidental chores such as writing reports and talking on the telephone. On one hand the human can make 'intelligent' errors in routine situations: on the other hand he can work tenaciously to resolve a complex and unforeseen mishap. The machine is tireless and entirely unemotional and can bring kilowatts of effort to bear in milliseconds after months of total inaction. On the other hand, in human terms it is just plain dumb: it has no idea what ought to be done, only what it has been told to do, right or wrong."

Given this large dissimilarity between humans and machines, it is sensible to ensure that each party within a human-machine system is charged with the tasks for which it is best equipped. Prior to the computer age, humans were required to perform many tasks that were better suited for computers to perform had they existed. In modern times, computers (and computer-controlled machines) replace humans in areas where computers are better suited to the tasks at hand, thereby freeing humans to concentrate on the tasks for which *they* are best suited.

Whereas modern human-machine systems are likely to include some form of computerization, many older systems still in operation today possess no computerization

to speak of, and rely heavily (or entirely) on human operators to perform tasks that would be better accomplished by computers or computer-directed machines. It is, therefore, important to ensure during computerization of such systems that tasks are assigned appropriately to the human(s) and computer(s) to ensure that each party is not assigned tasks for which it is not suited.

The preceding chapters have presented the Particle Accelerator Control Expert System as a case study of how computer-centered technology insertion (CCTI) can be applied to a specific human-machine system, the KN-3000 particle accelerator, and it is possible to extrapolate from this case study towards a generalized methodology for CCTI. The term *technology insertion* is used in this context to describe the modernization of the accelerator (or similar human-tended machine) by installing specialized computer-centered hardware and software to increase the usability and capability of the machine. In this case, *increased usability* means that less-skilled operators can operate the accelerator as experts, while *increased capability* means that the accelerator can be used in new ways that were previously impossible or intractable.

The insertion of such a computer system into an existing human-machine system is bound to disrupt proper operation of the machine, mainly due to human factors-related issues arising from the disturbance of operating procedure. There is, therefore, an acute need that the computer system is designed from the outset to minimize disruption of the existing human-machine relationship. In practice, there will always be *some* degree of disruption, and the solution is to offset this disruption by furnishing the operators with computer-oriented capabilities that did not previously exist as part of the complex system. That is, the operators will be more accommodating to a small amount of disruption if they perceive a *net gain* in their ability to operate the machine more effectively and efficiently, and in ways not previously possible. This, in turn, requires that the operators find the computer system acceptable (useful, usable and likable), and it is well understood (cf. § 3.3.2) that these qualities are more easily attained if the computer system is well designed to be considerate of human factors concepts.

At the same time, a significant factor in assessing the computer system's acceptability is its ability to function *properly* (that is, function from a real-time systems point of view in a timely, reliable and error-free manner). Users may indeed find a computer system easy to use, and likable to use, but ultimately reject the system due to its lack of reliability. Thus, it is necessary to incorporate real-time systems concepts into the development of the computer system in order to provide such important qualities as efficiency, timeliness, predictability, reliability, fault tolerance and safety (cf. § 3.1). This is especially important if the computer system can affect expensive equipment and property, or jeopardize human lives. Echoing this concern, Stankovic ([Sta88]) writes that:

> "Many real-time systems of tomorrow will be large and complex and will function in distributed and dynamic environments. They will include expert system components that will involve complex timing constraints encompassing different granules of time. Moreover, economic, human and ecological catastrophes will result if these timing constraints are not met. Meeting these challenges imposed by these characteristics very much depends on a focused and co-ordinated effort in all aspects of system development."

The proper incorporation of human factors and real-time systems concepts into the technology insertion exercise leads to development of acceptable interfaces between the computer system and the user, and the computer system and the machine, respectively. But, as indicated in Chapter 1, the insertion of a computer system into an existing human-machine interaction can create a 'technology insertion gap' between human and machine which is likely to disrupt proper machine operation. The remedy to this is to 'bridge' this gap by providing the system with functionality to assist operators through computerization and automation. *Computerization*, in this context, means using a computer system to transform aspects of machine operation from being entirely manual to being computer-assisted or completely automated, thereby streamlining or simplifying otherwise tedious or mundane activities. For example, Sage ([Sag90], p. 5) observes that:

> "In most cases, a new tool or machine makes it possible to perform a familiar task in a somewhat new and different way, typically with enhanced efficiency and effectiveness. In a smaller number of cases, a new tool has made it possible to do something entirely new and different that could not have been done before."

In this sense, the computerization exercise should strive to improve the existing human-machine interaction by offering novel capabilities for the human-machine system,

novel methods for the human-machine interaction, and novel perspectives for the humans interacting with the machine. In a sense, the computerization exercise should act as a flexible, mutable encapsulation for the machine — a malleable human-machine interface which enables the machine to be fit to the user, instead of the user being fit to the machine.

In addition to computerization *per se*, CCTI can also implement *automation* to liberate the operators from duties that do not require direct human supervision but which have traditionally involved human supervision for lack of better means of accomplishing the tasks. Artificial intelligence reasoning techniques can be used to enhance automated operations in order to capture human operational expertise and problem solving abilities, thereby preserving some semblance of the original human-machine interaction during automation.

Evidently, a computer system intended to possess such a diverse collection of attributes, from several different disciplines, implies a high level of system complexity, which, in turn, presents a challenge for achieving sufficient reliability. The more complex the system, the more difficult it is to prove reliable in all situations. Consequently, it is prudent to strive towards minimal system complexity while still providing the intended characteristics of the system. This effort to minimize complexity simultaneously while maximizing functionality presents systems developers with a tall order, and it may be tempting for developers to forego it in the interests of yielding to budgetary pressures and time constraints. The key, however, is to invest the effort early in the development process to ensure that the proper balance between simplicity and functionality is achieved. This may be difficult, or even impossible, if an iterative or prototyping development strategy is followed — the system could easily become overly complex as functionality is increased with each new version or prototype. Moreover, the method advocated earlier of involving users in prototype evolution can lead to systems whose complexity mushrooms due to poor initial planning followed by cycles of iterative system extension. On the other hand, completely abandoning an iterative prototyping lifecycle in favour of a more linear, more compartmentalized development scheme can result in an end product that is stilted and lacking in utility (functionality), usability or likability.

The PTP system development lifecycle model presented in Section 4.5.1 (cf. Figure 4-58) is organized to benefit from both linear (non-iterative) and iterative development phases. It combines the foresight of linear, phased development with the resiliency of iterative development. As stated in Section 3.4, there are many different lifecycle models, and developers tend to choose and evolve models based on the project requirements at hand and on past experience. In practice, no single lifecycle model is a panacea for all projects.

The PTP model is a skeleton framework upon which a generalized CCTI methodology can be built. It is intended to be employed for technology insertion projects targeted for small-scale human-tended systems, with a workload suitable for involvement of a small number of people and can be accomplished under a relatively small budget and in a relatively short timeframe. The development team should include engineering skills in the following areas: electrical, computer, process and control, software and human factors engineering. Additionally, domain experts and machine operators should be available for in-depth consultation during development, product evaluation during the prototyping stage(s), and during testing and commissioning of the final system.

The first step in the technology insertion exercise begins when a management-level decision is made to investigate the possibility of computerizing an existing human-machine system in an effort to modernize operations. This decision may come about for any number of reasons, and the most important thing to determine is whether a computer-centered modernization approach is warranted. What does *modernization* entail in terms of the system in question? What is hoped to be accomplished or gained from modernizing the system? It is likely that the existing system is already 'mature' in the sense that it has been well-debugged, the humans are accustomed to using the machine, and the system as a whole has been operating as designed and expected for a long time. Will system modernization through the insertion of computer-centered technology yield a net improvement in system operation, or will it disrupt and complicate the system to the detriment of effective operation? Unfortunately, it seems commonplace these days for system managers to justify computerization by claiming that the mere existence and availability of a good, powerful tool (the computer) justifies its use in a given application.

People are, in a sense, saying: 'Computerizing it *has* to make it better!' This illogical justification has all too often resulted in computers being 'thrown into' existing systems (that were working perfectly well beforehand) and wreaking unnecessary havoc. Siddall's treatise ([Sid94]) on the debacle of the computerized shutdown system for the Point Darlinginton CANDU nuclear power plant is a prime testament to the problems that can arise through misdirected computerization of existing, well-functioning systems. This report chronicles and diagnoses several situations in which computer software systems have led to potentially disastrous and lethal outcomes, including the Therac-25 accidents (cf. Section 3.1.2), the Bruce-A nuclear power plant refuelling system accident, and the problems suffered by the automatic shutdown system at the Darlington nuclear power plant. If this report can be taken as a representative sample of how computerization can adversely affect the proper operation of complex systems in general, then it has much to warn about the uncareful, brute force insertion of computerization into existing systems.

Nevertheless, there are many legitimate reasons for deciding to modernize a system. It may be that the human-machine system has experienced loss of operations expertise due to personnel layoffs, operator retirement or other factors. In this scenario, the modernization may be seen as a way to diminish the loss of expertise, or preserve it in a usable form. It may be the case that management perceives the need to extend the useful lifespan of the existing system because there is no short-term prospect of getting a new, modern system to replace it. A computerization project might be considered, in this situation, as a means of improving operations to increase the usefulness and lengthen the lifetime of the system. Another reason may stem from a desire to simplify or streamline operations by employing computerization to improve efficiency and automate aspects of system operation.

One of the issues that needs to be resolved early in the design process is determining what form the computer-centered modernization should take. Should the computer system *replace* the humans entirely, or should it *assist* them in their duties? Should it be autonomous or human-supervised? It is assumed, since the systems in question are human-tended, that it is desirable (or necessary) to maintain human involvement in some form, whether in the capacity of direct system control, or in a supervisory role. It is

therefore necessary for the computer system to incorporate human factors engineering to promote system acceptability (that is, utility, usability and likability).

Moreover, if the computer system is planned to have some level of autonomy, then it may be worthwhile to consider inclusion of artificial intelligence paradigms for problem solving and self-directed reasoning. Inclusion of AI mechanisms is certain to complicate the system, but the benefits of such prospects as machine learning, adaptability and human-like decision making may make employment of AI worth the extra effort.

Additionally, should the computer system be 'on line' (physically connected to the system) or 'off line' (playing an advisory role in system operation)? If the system is intended to be 'on line', then it must be fault tolerant and reliable, and perhaps even failsafe in its operation to avoid calamity. In the case of an off-line computer system, dependability is an important issue since humans will expect the advisory computer system to give good advice at all times.

Finally, it must be decided what programming language(s) and/or programming environment(s) will be used during development. This decision is influenced to a great extent by the degree to which third-party software modules can be acquired and utilized for the computer system. It may be that much of the software for the CCTI exercise already exists as separate libraries which need only be cobbled together; or, at the other extreme, it may be that the entirety of the software system must be developed from scratch.

All of these considerations must be weighed during the *initial conception* stage of the CCTI exercise; once the computer system's requirements have been delineated, the project can move into the *detailed specification and design* phase of the PTP lifecycle model. During this phase, several parallel threads of planning are carried out, beginning with two parallel threads for the hardware and software. The hardware used to interface the computer system to the machine (and possibly also the human) is designed. Various key issues must be resolved at this point: What type of computer should be used? Should there be an element of multiprocessing? If so, what type of topology should the multiprocessor employ? Should the processors be homogenous (all the same type) or heterogeneous

(different types)? In PACES, an IBM-style 80486 personal computer was used as the main computer (interfacing with the operator), and presided over one or more 8051 single-board computers which were used for controlling the machine interface. In this sense, PACES implemented a heterogeneous 'star-shaped' multiprocessor topology. Nevertheless, it may be that some other multiprocessor topology is more appropriate for the CCTI exercise in question. Some common topologies are shown in Figure 6-1.

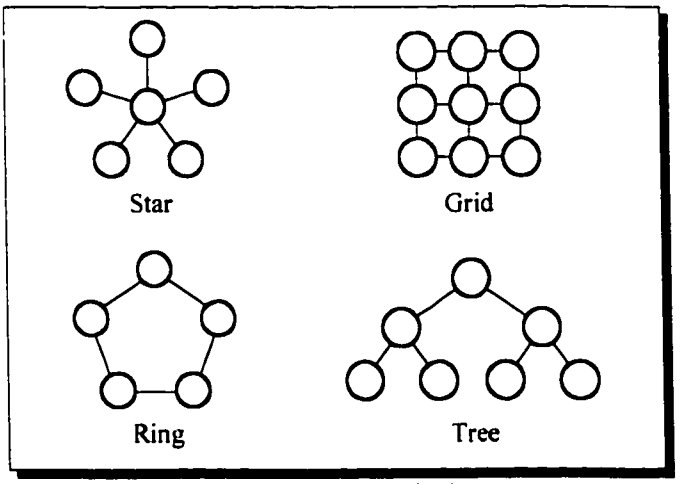


Figure 6-1. Various multiprocessor topologies. *Adapted from [Hwa84], p. 335*

Since the CCTI exercise involves interfacing with the machine to be controlled, how should the machine interface (MI) be realized? Should off-the-shelf components be used? Should custom DACS circuitry be built? How should the MI components be connected to the computer(s)? PACES made use of some existing, off-the-shelf MI components (e.g. terminal voltage stabilizer, single-board computer, NMR magnetometer), and added some customized DACS circuitry for digitizing control panel meters, turning selsyns and actuating control panel switches. Other CCTI operations may involve a large reliance on existing sub-systems, or require the entire MI to be designed and built from scratch. Again, the exact nature of the MI depends largely on the target machine, the existence of DACS hardware already built into the machine, and the availability of third-party DACS hardware which could be used in lieu of developing MI components from scratch. Ultimately, it is recommended that the qualities of the MI described in Section 4.2.2 (p. 106) are pursued: *Frugality* requires that the MI be low-cost and make use of existing MI components whenever possible. *Modularity* and *flexibility* ensure that the MI will be extensible to meet the changing needs of both the machine and the computer system.

Utility calls for the MI to be as autonomous as possible by 'off-loading' DACS tasks from software to hardware where appropriate so as to reduce software complexity and encourage the use of semi-autonomous hardware sub-systems. Finally, *passivity* is recommended in order to minimize the time that the machine is inoperable during CCTI development and to minimize disruption of normal machine operation during times when the computer system is not operating (due to such things as debugging, maintenance and revision). Together, these five qualities of the MI, when applied properly, will yield a machine interface which is relatively inexpensive yet flexible, simple to control via software, and minimally disruptive to normal machine operation.

The second thread of planning which should be carried out in parallel to the hardware thread concerns design and specification of the software which will run on the computer(s). There are two (possibly three) sub-threads of the software planning, one for the machine interface, one for the user interface (UI), and possibly one for the artificial intelligence (if any). The machine interface software is responsible for communicating with the machine interface hardware described above, and should incorporate appropriate features of real-time systems as described in Section 3.1. Issues which should be addressed for the machine interface software include such things as: How should the DACS workload be distributed between processors? What form of interprocessor communication should be used? Which aspects of the DACS are time-critical, resource-critical, fault-critical? What form and level of fault-tolerance should be employed?

The UI software should be planned out in a manner similar to that for the MI software: First and foremost, how should the user(s) interact with the computer system? The most common contemporary form of UI is the so-called 'windows'-based, direct manipulation GUI predominated by a high-resolution colour graphics monitor, mouse and keyboard, but being most common does not necessarily include being most appropriate. It may be that some other form of UI is better suited to the machine at hand. As detailed in Section 3.3, design of the UI must consider a wide range of details, including such things as the task(s) to be performed, the skill-level(s) of the users, and the users' attitudes towards computerization of their work (and towards computers in general). Additionally, it must be decided whether the user(s) should assume an active role in the UI's development, and

how this will affect the development process and the finished product. Ultimately, the most important detail is that of user acceptance of the computer system: the user(s) must find the computer system likable, usable and possessing of utility. Without user acceptance of the computer system, the CCTI operation is doomed to failure.

The final thread of parallel specification and design involves the knowledge based-system, if such is to be included in the computer system. Artificial reasoning should be considered for such tasks as: autonomous machine control; fault detection, diagnosis and recovery; operator training; and advising operators during machine operation. It was mentioned in Section 3.2 that artificial reasoning paradigms can be generally divided into three categories: expert systems, fuzzy systems and neural systems. Although only expert and fuzzy systems were applied in the PACES project, it may be that other CCTI exercises may be better served by neural systems as well as (or instead of) expert and fuzzy systems. In general, expert and/or fuzzy systems are applicable whenever there exists a good body of domain knowledge which can be quantified, collated and assembled (by a knowledge engineer) into a knowledge base. If no such domain knowledge is readily attainable, then it may be more appropriate to use neural systems, which are amenable to self-training without any need for existing domain knowledge. As for expert/fuzzy systems, the planning of the knowledge base (the knowledge engineering) should take place in parallel with the planning of the inference engine so as to ensure that the computer system possesses the proper set of inferencing tools for manipulating the knowledge base to accomplish the duties of the artificial reasoning sub-system. Some of the details worthy of consideration during this thread of planning include: What structure should the knowledge base assume? How extensible does the knowledge base need to be? Should a commercial expert system shell be used, or should one be developed from scratch? Should the inference engine be forward chaining, backward chaining, or both? How quickly do decisions need to be made? How accurately do decisions need to be made? What mechanisms should be employed to make the decision making reliable? What portion of the artificial reasoning should be performed by an expert system rule base, and what portion by fuzzy logic system FAMs? And finally, if the computer system is multiprocessor, how should the knowledge base be distributed among the processors?

Once the *detailed specification and design* phase has been completed, the CCTI project moves into the *parallel development* phase, during which time the hardware for the MI and the software for the MI, UI and KBS are further designed, coded, tested and modified in an iterative cycle. During this time, it may be that the machine's operators are taking an active role in the development, evaluating aspects of the UI, furnishing information and advice for the knowledge engineering, or providing technical assistance during the construction of the hardware MI. Much of the work performed during this phase needs to be carried out *on-site*, with *ad libitum* access to the machine and its personnel (operators, technicians, managers, etc.), but some parts of the development might be amenable to *off-site* development.

The *parallel development* phase is followed closely by the *prototyping* phase, in which the various parts of the computer system are integrated, connected to the machine, and presented to the end-user(s) for evaluation. As described in Section 4.5.1 (p. 173), this evaluation has one of four outcomes: At worst, the end-user(s) may decide to *scrap* the system; instead, it is more likely that they will either request that development *continue* or that the system be *extended*, and the parallel development/prototyping cycle continues; or, at best, the end-users may decide to *accept* the system, and the system enters its *operational/maintenance* phase.

* * *

This chapter has presented a discussion in general terms of how the CCTI exercise exemplified by the PACES project can be extrapolated into a framework for a CCTI methodology for modernizing and upgrading human-machine systems.

The next, and final chapter offers a summary of the work presented in this thesis, draws some conclusions about the use of CCTI for modernization of human-machine systems, and outlines some areas worthy of further study.

Chapter 7

Conclusion

This thesis has explored the subject of computer-centered technology insertion (CCTI) for modernization of small-scale, human-tended systems. The Particle Accelerator Control Expert System has been presented as an example of such a technology insertion operation in which a hybrid computer system has been developed to become a modernization component intended to provide performance enhancement and operations support for a KN-3000 particle accelerator facility.

Chapter 1 introduced the technology insertion problem by claiming that computer-centered modernization has great potential benefits, but must be performed sufficiently well that the resulting system functions *properly* and *reliably*, and is *accepted* by the users.

Chapter 2 described the KN-3000 particle accelerator, outlining it basic operating principles, and explaining how the loss of operations expertise at Defence Research Establishment Ottawa, coupled with budgetary constraints and the necessity of maintaining accelerator operations at their present level, establishes a need for modernizing this system through the insertion of computer-centered technology.

An extensive literature survey was presented in Chapter 3 to show how four diverse areas of electrical and computer engineering have bearing on the technology insertion exercise: Real-time systems principles are required to interface the computer system with the machine. Human factors concepts are required to ensure that the computer system is accepted by the users. Artificial intelligence paradigms can be applied to provide the computer system with autonomy. Lastly, software engineering methods are essential for amalgamating these diverse disciplines into a functioning computer system.

The Particle Accelerator Control Expert System (PACES) was investigated in Chapters 4 and 5 as a case study of a technology insertion operation. Real-time systems principles were applied to build a hierarchical, multiprocessor environment capable of timely response to both the accelerator and the user. The artificial intelligence paradigms of expert systems and fuzzy logic controllers were used to provide PACES with a degree of reasoning and problem solving ability sufficient to enable automated accelerator operation. Aspects of human-computer interaction and the computer-human interface were involved in making the software system useful, usable and likable by the accelerator operators. The software engineering concepts of the software development lifecycle, object-oriented programming, modularity, specification, reuse of code, and information hiding were harnessed to forge a working, hybridized computer system for performance enhancement and operations support.

Finally, Chapter 6 presented a discussion concerning computer-centered technology insertion in general, highlighting the pertinent issues which must be considered and addressed during the planning, development and maintenance phases of CCTI operations.

The issues of accelerator modernization and operator performance enhancement were paramount during design and implementation of PACES. Operator performance enhancement was achieved in several forms, including preservation of operating expertise, automation of operating procedures, augmentation of operator abilities, and streamlining of record-keeping requirements. Within the PACES environment, performance enhancement relies completely upon technology insertion: The installation of new, computer-oriented hardware and software is required for the performance enhancement features. It must be emphasized that this computer system, with its customized accelerator-interface hardware and flexible operator-interface software, serves as a low-cost and vital bridge between the ageing accelerator and a reduced, less-experienced operator pool.

The PACES project, as an example of CCTI in general, constitutes a body of original work in that it has successfully combined aspects of real-time systems, human-computer interaction and artificial intelligence to produce a semi-autonomous particle accelerator

control system and operator performance enhancement platform: Simple, relatively inexpensive electronic circuitry was used to connect a modern, hierarchical multiprocessor computer system with the ageing accelerator which was never intended or designed to be computer-controlled to any appreciable degree. The non-invasive nature of the computer-machine interface provided for a flexible and modular 'piggy-back' data acquisition and control system which facilitated computer-based accelerator control without disrupting normal, manual (non-computerized) accelerator operation.

Simultaneously, the customized PACES graphical user interface offered operators a novel perspective on accelerator operation, furnishing them with a software-based collection of tools to improve and augment their expertise with such features as instrumentation consolidation, combined-mode analog/digital meters, time-based stripcharts, semi-autonomous 'fire and forget' selsyn controllers, system-wide kiviat graphs of accelerator parameters, electronic record keeping and real-time accelerator data logging. Also, the highly desirable qualities of software acceptability (utility, likability and usability) were approached through the active involvement of the end-users (operators) in the design, implementation and testing phases of system development, thereby promoting the successful bridging of the so-called 'technology insertion gap'.

Additionally, the inclusion of artificial intelligence reasoning mechanisms imbued the control system with moderate autonomy and the ability to control the accelerator in ways modelled on and similar to conventional manual operating procedures, with comparable (and in some cases superior) response time and accuracy. The expert system-based Auto-Pilot enabled PACES to perform automated start-up, shut-down, voltage conditioning and beam maintenance operations (under operator supervision or autonomously), with the ability to generate and then maintain the particle beam on target within specified tolerance intervals, even through periods of terminal voltage instability and particle beam loss. Furthermore, the limited application of fuzzy logic-based control techniques to terminal voltage set-point acquisition and maintenance indicated that fuzzy logic-based controllers offer a simple and useful means of implementing various appropriate sub-components of the overall control task in replace of higher-overhead expert system-based decision-making mechanisms.

Finally, the rigours of software engineering and the power of object-oriented programming, under the encompassing umbrella of the Parallel Threaded Prototyping development lifecycle model (or other suitable lifecycle model), ensure that all the abovementioned aspects of the computerization exercise will meld together seamlessly and effectively to yield a monolithic, hybridized control and performance enhancement system.

Several conclusions can be realized from this work. Foremost, it is not only possible but readily accomplishable to modernize and upgrade a mature, human-tended machine via computer-centered technology insertion, even if the target machine was never intended to be modernized in such a way. Moreover, the application of computer-centered operations support and performance enhancement can improve the operator's ability to use the machine while at the same time expand the machine's capability to do its job. That is, the truly flexible and extensible nature of the computer software creates a pliable human-machine interface which can both fit the machine to the human and evolve to better levels of human-machine interaction — the computer intermediary serves as a mutable stage for the human-machine interaction, providing ways of interacting with the machine that are otherwise impossible. When this power of flexible interfacing is coupled with the power of artificial intelligence-based autonomy, human-computer teamwork is bolstered because each party can attend to the duties to which it is best suited, ultimately resulting in an improved, composite human-computer-machine interaction. The computer system, in this sense, can be likened to a *Swiss Army Knife*, an efficient, compact collection of useful tools to aid the operator at work — each tool, on its own, improves the operator's ability to function, and by combining several useful tools into one, an even greater improvement is gained. Thus, the exercise of computerizing a human-machine system by assembling a 'toolkit' of various tools selected from the realms of real time systems, human factors and artificial intelligence can offer significant advantages, and can improve the operation of the machine, preserve operational expertise, and possibly even extend its useful lifespan.

The work presented in this thesis suggests various avenues of further study, particularly in the areas of artificial intelligence and hierarchical multiprocessing for process control. How can the PACES knowledge base (and underlying inference engine) be extended to yield faster decision-making, more comprehensive fault detection and

diagnosis, and enhanced adaptability to gradually changing operating regimens and age-related shifts in accelerator characteristics? How much more of a role can fuzzy logic-based controllers play in accelerator control? Is it possible to develop adaptive fuzzy logic controllers to accommodate gradual changes in the accelerator's behaviour as it ages? If the hierarchical processing topology is expanded through the addition of other embedded controllers, how best to partition the work load between the host computer and embedded controllers? Should the knowledge base be distributed between processors, and if so, should such distribution be static or dynamic?

With regard to the computer-human interface, how can the PACES GUI be extended and improved? Are there more effective ways of increasing operator performance? Are there novel forms of human-machine interaction which could be incorporated into PACES to enhance accelerator operation further? What other aspects of accelerator operation can be relegated to automatic control in order to relieve the operators and free them to accomplish other tasks concurrently?

All of these questions are worthy of further investigation, and would serve to broaden the effectiveness of PACES as an example of computer-centered technology insertion.

* * *

There was a time, not so long ago, when the computer was no more than the cogs-and-gears dream of Charles Babbage, unable to take form for lack of adequate funding and mechanical engineering skills. And, there was a time, also not so long ago, when humans were obliged to perform all variety of arduous and drudgerous tasks, for want of more suitable agents to labour in their stead. And then, the dream took form, and the computer assumed the yoke, shouldered its harness and bent to its task, relieving its creators of their labour.

It seems increasingly likely that in the years ahead computerization will creep its way into every corner of the world and into every aspect of human life. Computer-centered technology insertion is a part of this ongoing computerization, a way of modernizing and computerizing small-scale human-tended machines and systems. This thesis has sought to

identify and explore the wide-ranging details and concerns of computer-centered technology insertion, and has suggested a hybridized methodology for accomplishing such computerization exercises. In doing so, it is hoped that similar technology insertion operations are met with all measure of success in the years to come.

References

[Bac84] Bachant, J., McDermott, J., "R1 Revisited: Four Years in the Trenches", *The AI Magazine*, vol. 5, no. 3, Fall, 1984.

[Bad91] Bader, J., Edwards, J., Harris-Jones, C., Hannaford, D., "Practical Engineering of Knowledge-based Systems", *Artificial Intelligence and Software Engineering*, Partridge, D., ed., Ablex Publishing Corp., Norwood, New Jersey, 1991, pp. 383-407.

[Bar87] Barnard, P.J., "Cognitive Resources and the Learning of Human-Computer Dialogs", *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, Carroll, J.M., ed., MIT Press, Cambridge, Massachusetts, pp. 112-158.

[Bar91] Barnard, P.J., "The Contributions of Applied Cognitive Psychology to the Study of Human-Computer Interaction", *Human Factors for Informatics Usability*, Shackel, B., Richardson, S., eds., Cambridge University Press, Cambridge, England, 1991, pp. 151-182.

[Bar95] Bardik, Y., Kallistratov, E., Machnachev, A., Matiouchine, A., Obukhov, G., "Microcontroller Applications for IHEP Accelerator Control", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Ben88] Bennett, S., *Real-Time Control: An Introduction*, Prentice Hall, New York, 1988.

[Bir82] Birnbaum, J.S., "Computers: A Survey of Trends and Limitations", *Science*, vol. 215, 1982, pp. 760-765.

[Bir95] Birke, T., Lange, R., Muller, R., "Object Oriented API Layers Improve Modularity of Applications Controlling Accelerator Physics", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Boe76] Boehm, B.W., "Software Engineering", *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 1226-1241, December, 1976.

[Boe86] Boehm, B.W., "A Spiral Model of Software Development and Enhancement", *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 4, pp. 14-24, August, 1986.

[Boo86] Boose, J.H., *Expertise Transfer for Expert System Design*, Elsevier Science Publishing, Amsterdam, Netherlands, 1986.

[Bur90] Burns, A., Wellings, A., *Real-time Systems and Their Programming Languages*, Addison-Wesley Publishing Co., Wokingham, England, 1990.

[Car83] Card, S.K., Moran, T.P., Newell, A., *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[Cav67] Cavanagh, J.M.A., "Some Considerations Relating to User-System Interaction in Information Retrieval Systems", *Information Retrieval: The User's Viewpoint*, Tonik, A.B., ed., Fourth Annual National Colloquium on Information Retrieval, International Information Inc., Philadelphia, 1967.

[Cha91] Chapanis, A., "Evaluating Usability", *Human Factors for Informatics Usability*, Shackel, B., Richardson, S., eds., Cambridge University Press, Cambridge, England, 1991, pp. 359-395.

[Che95] Chen, J., Akers, W., Heyes, G, Wu, D., Watson, C., "An Object-Oriented Class Library for Developing Device Control Application", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, October 29-November 3, 1995.

[Che95] Chen, J., Akers, W., Heyes, G., Wu, D., Watson, C., "An Object-Oriented Class Library for Developing Device Control Application", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[dAg87] d'Agapeyeff, A., Hawkins, C.J.B., *Report to the Alvey Directorate on the Second Short Survey of Expert Systems in UK Business*, IEEE on behalf of the Alvey Directorate, United Kingdom, 1987.

[Dau92] Daugherity, W.C., Rathakrishnan, B., Yen, J., "Performance Evaluation of a Self-Tuning Fuzzy Controller", *Proceedings of the IEEE Conference on Fuzzy Systems*, March 8-12. San Diego, CA, 1992, pp. 389-397.

[DeM91] DeMooy, S., *Development Towards a Control Program for a Model KN Van de Graaff Particle Accelerator*, M.Sc. Thesis, Dept. of Physics, McMaster University, 1991.

[Des62] Descartes, R., *Traité de l'homme*, 1662. Translated by Haldane, E.S., Ross, G.R.T., Cambridge University Press, Cambridge, England, 1911.

[Dic95] Dickey, C., Burnham, B., Carter, F., Fricks, R., Litvinenko, V.N., Nagchaudhuri, A., Morcombe, P., Pantazis, R., O'Shea, P., Sachtschale, R., Wu, Y., "EPICS at Duke University", *Proceedings of the 1995 Particle Accelerator Conference*, Dallas, Texas, 1995, pp. 2217-2219.

[Dic95] Dickey, C., Burnham, B., Carter, F., Fricks, R., Litvinenko, V.N., Nagchaudhuri, A., Morcombe, P., Pantazis, R., O'Shea, P., Sachtschale, R., Wu, Y., "EPICS at Duke University", *Proceedings of the 1995 Particle Accelerator Conference*, Dallas, Texas, 1995, pp. 2217-2219.

[DiM95] Di Maio, F., Goetz, A., "Towards a Common Object Model and API for Accelerator Controls", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, October 29-November 3, 1995.

[DiM95] Di Maio, F., Goetz, A., "Towards a Common Object Model and API for Accelerator Controls", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Dor77] Dorf, R.C., *Computers and Man*, 2nd ed., Boyd & Fraser Publishing Company, San Francisco, 1977, p. 21.

[Dri93] Driankov, D., Hellendoorn, H., Reinfrank, M., *An Introduction to Fuzzy Control*, Springer-Verlag, New York, 1993.

[Ell91] Ellison, D., *Understanding occam and the transputer*, Sigma Press, Wilmslow, England, 1991.

[Eng67] Enge, H.A., "Magnetic Spectrographs", *Physics Today*, July 1967, pp. 65-75.

[Epa95] Epaud, F., "Beamline Control at ESRF", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Fai85] Fairley, R.E., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.

[Gai91] Gaines, B., "Designing Expert Systems for Usability", *Human Factors for Informatics Usability*, Shackel, B., Richardson, S., eds., Cambridge University Press, Cambridge, England, 1991, pp. 207-246.

[Gal97] Galitz, W.O., *The Essential Guide to User Interface Design : An Introduction to GUI Design Principles and Techniques*, Wiley Computer Pub., New York, 1997.

[Gil89] Gilmore, W.E., The User-Computer Interface in Process Control : A Human Factors Engineering Handbook, Academic Press, Boston, 1989.

[Gla82] Gladden, G.R., "Stop the Life Cycle, I Want to Get Off", *ACM Software Engineering Notes*, vol. 7, no. 2, 1982, pp. 35-39.

[Gle81] Gleitman, H., *Psychology*, W.W. Norton and Company, New York, 1981.

[Gol83] Goldberg, A., Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

[Gou83] Gould, J.D., Drongowski, P., "Designing for Usability — Key Principles and What Designers Think", *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems*, ACM, New York, 1983, pp. 50-53.

[Hal92] Halang, W.A., Sacha, K.M., *Real-Time Systems, Implementation of Industrial Computerised Process Automation*, World Scientific, Singapore, 1992.

[Har88] Harmon, P., Mans, R., Morrissey, W., *Expert Systems: Tools and Applications*, John Wiley and Sons, New York, 1988.

[Hol91] Hollnagel, E., "The Phenotype of Erroneous Actions: Implications for HCI Design", *Human-Computer Interaction and Complex Systems*, Weir, G. R. S., Alty, J. L., eds., Academic Press Inc., San Diego, California, 1991, pp. 73-121.

[Hwa84] Hwang, Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, New York, 1984.

[IEE83] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 729-1983.

[IEE92] *Proceedings of the IEEE Conference on Fuzzy Systems*, March 8-12, San Diego, CA, 1992.

[Jac92] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Publishing Co., Wokingham, England, 1992.

[Jen93] Jennings, N.R., "The ARCHON System and its Applications", *Second International Working Conference on Cooperating Knowledge Based Systems*, Keele, UK, 1994, pp. 13-29.

[Jen94] Jennings, N.R., "The ARCHON System and its Applications", *Second International Working Conference on Cooperating Knowledge Based Systems*, Keele, UK, 1994, pp. 13-29.

[Jen96] Jennings, N.R., Corera, J., Laresgoiti, I., Mamdani, E.H., Perriolat, F., Skarek, P., Varga, L.Z., "Using ARCHON to develop real-word DAI applications for electricity transportation management and particle accelerator control", *IEEE Expert*, 1996.

[Jen96] Jennings, N.R., Corera, J., Laresgoiti, I., Mamdani, E.H., Perriolat, F., Skarek, P., Varga, L.Z., "Using ARCHON to develop real-word DAI applications for electricity transportation management and particle accelerator control", *IEEE Expert*, 1996.

[Kos92] Kosko, B., *Neural Networks and Fuzzy Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[Kuz95] Kuznetsov, S., "C++ Library for Accelerator Control and On-line Modeling", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Lap85] Laprie, J.C., "Dependable Computing and Fault Tolerance: Concepts and Terminology", *Digest of Papers, The Fifteenth Annual International Symposium on Fault-Tolerant Computing*, Michigan, USA, 1985, pp. 2-11.

[Lar81] Larsen, P.M., "Industrial Applications of Fuzzy Logic Control", *Fuzzy Reasoning and Its Applications*; Mamdani, E.H., Gaines, B.R., eds., Academic Press, New York, 1981, pp. 335-342.

[Law92] Lawson, H.W., *Parallel Processing in Industrial Real-Time Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Lev86] Leveson, N.G., "Software Safety: what, why and how", *ACM Computing Surveys*, vol. 18, no. 2, 1986, pp. 125-163.

[Lev93] Leveson, N.G., Turner, C.S., "An Investigation of the Therac-25 Accidents", *Computer*, August, 1993, pp. 18-41.

[Lew95] Lewis, J., Skarek, P., Varga, L., "A Rule-Based Consultant for Accelerator Beam Scheduling used in the CERN PS Complex", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, October 29-November 3, 1995.

[Lew95] Lewis, J., Skarek, P., Varga, L., "A Rule-Based Consultant for Accelerator Beam Scheduling used in the CERN PS Complex", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Lic65] Licklider, J.C.R., "Man-computer partnership", *International Science and Technology*, vol. 41, pp. 18-26.

[Lin88] Lingarkar, R., *Automation of the Low-energy Steerers for the FN Tandem Accelerator*, M.Sc. dissertation, McMaster University, Hamilton, Ontario, 1988.

[Lin91] Lind, P.C., Poehlman, W.F.S. and Stark, J.W., "Implementation Considerations for PACES: The KN-3000 Particle Accelerator Control Expert System", *Proceedings of the Third Symposium/Workshop on Expert Systems in the Department of National Defence*, Royal Military College, Kingston, Ontario, Canada, pp. 17-36, May 2-3, 1991.

[Lin92a] Lind, P.C., Poehlman, W.F.S., Stark, J.W. and Cousins, T., "Knowledge Engineering for PACES: The KN-3000 Particle Accelerator Control Expert System", *Proceedings of the Fourth Symposium/Workshop on Expert Systems in the Department of National Defence*, Royal Military College, Kingston, Ontario, Canada, pp. 133-152, April 23-24, 1992.

[Lin92b] Lind, P.C., Poehlman, W.F.S., "Design of an Expert-System-based Real-Time Control System for a Van de Graaff Particle Accelerator", *Applications of Artificial Intelligence in Engineering*, Grierson, D.E., Rzevski, G., and Adey, R.A., eds., Elsevier Applied Science, NY, pp. 317-334, 1992.

[Lin93a] Lind, P.C., Poehlman, W.F.S. and Stark, J.W., "The KN-3000 Particle Accelerator Control Expert System (PACES)", *IEEE Transactions on Nuclear Science*, vol. 40, no. 6, December, 1993.

[Lin93b] Lind, P.C., Poehlman, W.F.S., "Fuzzy Logic for Particle Accelerator Control", *Proceedings of the Fifth Symposium/Workshop on Expert Systems in the Department. of National Defence*, Westin Hotel, Ottawa, Ontario, Canada, November 14-17, 1993.

[Lin94] Lind, P.C., Poehlman, W.F.S., "Technology Insertion and Performance Enhancement for a Van de Graaff Particle Accelerator", *Proceedings of the Workshop on Performance Support Systems for Nuclear Power Plants*, Applied Computersystems Group, McMaster University, July 15-17, 1994.

[Mam81a] Mamdani, E.H., "Advances in Linguistic Synthesis of Fuzzy Controllers", *Fuzzy Reasoning and Its Applications*; Mamdani, E.H., Gaines, B.R., eds., Academic Press, New York, 1981, pp. 325-334.

[Mam81b] Mamdani, E.H., Assilian, S., "An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller", *Fuzzy Reasoning and Its Applications*; Mamdani, E.H., Gaines, B.R., eds., Academic Press, New York, 1981, pp. 311-323.

[Man97] Mandel, T., *The Elements of User Interface Design*, John Wiley and Sons, New York, 1997.

[McC82] McCracken, D.D., Jackson, M.A., "Life Cycle Concept Considered Harmful", *ACM Software Engineering Notes*, vol. 7, no. 2, 1982, pp. 29-32.

[McI96] McIlwain, A.K., Lind, P.C., "An Intelligent Windows-based Interface for the KN4000 Van de Graaff Accelerator", *Symposium of North Eastern Accelerator Personnel*, TUNL, North Carolina, 1996.

[Mej95] Mejuev, I., Abe, I., Nakahara, K., "AI-Based Accelerator Control", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Mil94] Personal communication, Dr. P. Milgram, Associate Professor, Department of Industrial Engineering, University of Toronto, September 1, 1994.

[Nic77] Nickerson, R.S., Pew, R.W., "Person-Computer Interaction", Chapter 6, *The C³-system User: vol. 1. A Review of Research on Human Performance as it Relates to the Design and Operation of Command, Control and Communication Systems*, (Report no. 3459), Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1977.

[Nic86] Nickerson, R.S., *Using Computers: The Human Factors of Information Systems*, MIT Press, Cambridge, Massachusetts, 1986.

[Nor83] Norman, D.A., "Design Principles for Human-Computer Interfaces", *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems*, ACM, New York, 1983, pp. 1-10.

[Nor86] Norman, D.A., "Cognitive Engineering", *User Centered System Design*, Norman, D.A., Draper, S.W., eds., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 31-62.

[Par88] Partington, D., "Artificial Intelligence in Process Control", *Measurement and Control*, vol. 21, no. 6, July-August 1988, pp. 177-178.

[Pas86] Pascoe, G.A., "Elements of Object-oriented Programming", *BYTE*, vol. 11, no. 8, August 1986, pp. 139-144.

[Pea64] *Pears Cyclopaedia*, Seventy-Third Edition, 1964-1965.

[Poe91] Poehlman, W.F.S., Garland, Wm., Stark, J., "Design Principle Employed in Aid of Real-Time Expert Control Systems Development", *Proceedings of the Fourth Atrificial Intelligence Symposium*, University of New Brunswick, Fredericton, NB, September 20-21, 1991, pp. 109-119.

[Pro87] *Proceedings of the 7th International Workshop on Expert Systems and Their Applications*, Avignon, France, May 1987.

[Riv95] Rivers, M., "Beamline Control and Data Acquisition at the Advanced Photon Source", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Rod97] Rodriguez, B.J., *An Embedded Temporal Expert for Control of a Tandem Accelerator*, Ph.D. dissertation, McMaster University, Hamilton, Ontario, 1997.

[Roy70] Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques", *Proceedings WESCON*, pp. 1-9, 1970.

[Ryb95] Rybin, V.M., Rybina, G.V., "Use of Expert System for Beam Diagnostics", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Sag90] Sage, A.P., Palmer, J.D., *Software Systems Engineering*, John Wiley and Sons, New York, 1990.

[Sak95] Sakaki, H., Hori, T., Yoshikawa, H., Suzuki, S., Taniuchi, T., Kuba, A., Yokomizo, H., "Conditioning of the SPring-8 LINAC RF System Using Fuzzy Logic", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Sha81] Shackel, B., "The Concept of Usability", *Proceedings of IBM Software and Information Usability Symposium*, Poughkeepsie, New York, Sept. 15-18, 1981, pp. 1-30.

[Sha91a] Shackel, B., Richardson, S., "Human Factors for Informatics Usability - Background and Overview", *Human Factors for Informatics Usability*, Shackel, B., Richardson, S., eds., Cambridge University Press, Cambridge, England, 1991, pp. 1-19.

[Sha91b] Shackel, B., "Usability - Context, Framework, Definition, Design and Evaluation", *Human Factors for Informatics Usability*, Shackel, B., Richardson, S., eds., Cambridge University Press, Cambridge, England, 1991, pp. 21-37.

[She82] Sheil, B.A., "Coping with Complexity", *Information Technology and Psychology: Prospects for the Future*, Kasschau, R.A., Lachman, R., Laugherty, K.R., eds., Praeger, New York, 1982.

[She83] Sheil, B., "Power Tools for Programmers", *Datamation*, vol. 29, no. 2, 1983, pp. 131-144.

[Shn91] Shneiderman, B., "A Taxonomy and Rule Base for the Selection of Interaction Rules", *Human Factors for Informatics Usability*, Shackel, B., Richardson, S., eds., Cambridge University Press, Cambridge, England, 1991, pp. 325-342.

[Sho76] Shortliffe, E.H., *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York, 1976.

[Sid94] Siddall, E., *The Engineering of CANDU Shutdown Systems Using Computers in Automatic Protective Functions*, Report prepared for AECL Research in part-fulfillment of Contract No. UC85929, Revision 1, Department of Systems Design Engineering, University of Waterloo, Waterloo, Ontario, Canada, May 2, 1994.

[Sil88] Silbar, R.R., Schultz, D.E., "Automation of Particle Accelerator Control", *Proceedings of Computers in Engineering*, San Francisco, CA, July 31-Aug. 4, 1988.

[Sin83] Sinha, N.K., Kuszta, B., *Modeling and Identification of Dynamic Systems*, Van Nostrand Reinhold Co., New York, 1983.

[Sma94] Small, P., "Ergo.. what? It's the future", *The Toronto Star*, August 21, 1994, p. A2.

[Sta88] Stankovic, J.A., "Misconceptions about Real-time Computing", *Computer*, vol. 21, no. 10, 1988, pp. 10-19.

[Sta95] Stanek, M., "Development of Operator and User Requested Control System Applications; Experience with the SLC Control System at SLAC", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Str92] Personal communication, Mr. F. Strain, Accelerator operator (retired), Nuclear Effects Division, Defence Research Establishment Ottawa, July 5, 1992.

[Swa94] Swainson, G., "Ambulance system 'unacceptable'", *The Toronto Star*, September 27, 1994, p. A2.

[Tan95] Tang, J., Shoaee, H., "A Comparative Study of Fuzzy Logic and Classical Control with EPICS", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Tau89] Taunton, C., "Expert Systems in Process Control: An Overview", *ISA '89 Advanced Control Conference*, NEC, Birmingham, 1989, pp. 5.1-5.5.

[Umb81] Umbers, I.G., King, P.J., "An Analysis of Human Decision-Making in Cement Kiln Control and the Implications for Automation", *Fuzzy Reasoning and Its Applications*, (Received May 25, 1973); Mamdani, E.H., Gaines, B.R., eds., Academic Press, New York, 1981, pp. 369-381.

[Vio93] Viot, G., "Fuzzy Logic in C", *Dr. Dobb's Journal*, no. 197, February, 1993, pp. 40-49.

[Wan94] Wang, C.J., Chen, J., Chen, J.S., Jan, G.J., "The Design Schemes of Graphic User Interface Database and Intelligent Local Controller in the SRRC Control System", Nuclear Instruments and Methods in Physics Research, North-Holland, A 352, 1994, pp. 300-305.

[Wes95] Westervelt, R.T., Klein, W.B., "Framework for a General Purpose, Intelligent Control System for Particle Accelerators", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Wil84] Willis, J., Milier, M., *Computers for Everybody: 1984 Buyer's Guide*, Dilithium Press, Beaverton, Oregon.

[Wu95] Wu, Y., Burnham, B., Litvinenko, V.N.,."The Duke Storage Ring Control System", *Proceedings of the 1995 Particle Accelerator Conference*, Dallas, Texas, 1995, pp. 2214-2216.

[Wu95] Wu, Y., Burnham, B., Litvinenko, V.N., "The Duke Storage Ring Control System", *Proceedings of the 1995 Particle Accelerator Conference*, Dallas, Texas, 1995, pp. 2214-2216.

[Yas85] Yasunobu, S., Miyamoto, S., "Automatic Train Operation System by Predictive Fuzzy Control", *Industrial Applications of Fuzzy Control*, Sugeno, M., ed., Elsevier Science Publishers B.V., North-Holland, 1985, pp. 1-18.

[Yos95] Yoshikawa, H., Itoh, Y., Sakaki, H., Taniuchi, T., Kodera, M., Tamezane, K., Kuba, A., Hori, T., Suzuki, S., Yanagida, K., Mizuno, A., Yokomizo, H., "Software Project for SPring-8 Linac Control", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[You82] Yourdon, E.N., *Managing the System Life Cycle: A Software Development Methodology Overview*, Yourdon Press, New York, 1982.

[Zad65] Zadeh, L.A., "Fuzzy Sets", *Information and Control*, vol. 8, 1965, p. 338.

[Zad73] Zadeh, L.A., "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes", *IEEE Trans.*, SMC-3-1, 1973, p. 28.

[Zha95] Zhao, J., Wang, B., Wang, C., Geng, X., Xu, J., "New Man-Machine Interface at the BEPC", *The 1995 International Conference on Accelerator and Large Experimental Physics Control Systems*, Chicago, Illinois, October 29-November 3, 1995.

[Zie91] Ziegler, J., Bullinger, H.-J., "Formal Models and Techniques in Human-Computer Interaction", *Human Factors for Informatics Usability*, Shackel, B., Richardson, S., eds., Cambridge University Press, Cambridge, England, 1991, pp. 183-206.

[Zol82] Zoltan, E., Chapanis, A., "Strategic Command, Control, Communications and Intelligence", *Science*, vol. 224, 1982, pp. 1306-1311.