LABORATORY AUTOMATION AND REAL-TIME COMPUTING

Duncan A. Mellichamp
University of California
Santa Barbara, CA 93106

Babu Joseph
Washington University
St. Louis, MO 63130

Abstract

We discuss the history and CACHE contributions to the field of laboratory automation and real-time computing. The evolution of the field through the last few decades is examined, and notable historical milestones are mentioned. The chapter concludes with a brief look at the future of the field.

Introduction

The use of computers for data acquisition and control in the chemical engineering laboratory was very quickly viewed by university faculty as having great potential for application in both teaching and research. Indeed, the early days of CACHE coincide with the first commercial developments of minicomputers, the innovation that would make possible practical university applications in this area. Thus, it should be no surprise that one of CACHE's first new activities, following the initial focus on large-scale computing applications, was on "real-time computing," i.e., the use of digital computers for data acquisition and control and for special purpose dynamic simulation (provided primarily via analog computers prior to that time).

This chapter looks back over those early beginnings in the late 1960s, the development of the minicomputer-based applications that occurred during the 1970s, the microcomputer revolution of the 1980's, and finally where the field appears to be going in the 1990s. The entire area of laboratory automation can now be fairly characterized as representing a mature field, i.e., one in which the emphasis is on utilizing an inexpensive, well understood, and rapidly expanding technology (rather than on trying to understand a brand new technology, acquire it, and bend it to purpose).

One of CACHE's most significant publication projects is represented by the CACHE Monograph Series in real-time computing, eight volumes written by early specialists in the field and edited by Mellichamp (1977-1979). These provided a major impetus at the time for prospective new users to begin work in this new area. Clear evidence of the relative maturity of the field just one decade later is provided by the CACHE *Anthology of Applications in On-Line Automation*, edited by Mellichamp and Cinar (1988) and containing nearly two dozen specific examples of undergraduate laboratory projects utilizing the computer.

Early Developments (1960s)

Pioneering academic workers in this area came primarily from one of two groups: (1) those having experience with industrial digital computer control projects, and/or (2) those with a background in analog hybrid computing. During the early 1960s industrial control projects had progressed from the first supervisory control projects (in which a large-scale digital computer was connected to analog instrumentation for the purpose of optimizing the placement of controller set-points) to so-called direct digital control systems (in which similarly large computers converted process measurements directly to digital form and manipulated the process actuators directly, as well). In either case, the computer systems used to carry out industrial control were quite large, costing on the order of hundreds of thousands of dollars; nevertheless academic appetites were greatly whetted by experiences with such systems, particularly those individuals interested in carrying out control research or in monitoring complex laboratory processes.

Once reliable (solid-state) and relatively inexpensive computers became available, monitoring and control facilities were established in chemical engineering departments at several universities where access to necessary funding was found. One cites, in particular, the early work of Grant Fisher at Alberta and Roger Schmitz, then at Illinois, both able to build facilities around the IBM 1800, a machine of modest capabilities (by today's standards) adapted from the earlier stand-alone Model 1620. Other academics came to laboratory real-time computing via a less expensive route during this period, by utilizing the digital computer part of an analog hybrid computer — the sort used for simulation purposes in a number of departments in the early 1960s. One can cite Cecil Smith at Louisiana State as one who followed this path.

By the end of this decade, a number of chemical engineering faculty were utilizing any available computer, e.g., an IBM 1800 linked from another building or across campus, as at Santa Barbara, facilities shared with analytical chemists, or whatever. All of the facilities of this period suffered from high costs and continued low reliability and from hardware and software limitations that, in retrospect, seem overwhelming by today's measures. On the other hand, some significant research was performed; most often this involved dedicated technicians and graduate students who practically lived with the systems.

The Age of the Minicomputer (1970s)

The first practical minicomputer, the DEC PDP-8, appeared commercially in the mid-1960s. This 12-bit, single-accumulator machine was purchased and used by a relatively large number of laboratory chemists but, for some unexplained reason, the PDP-8 was largely ignored by chemical engineers. However, the Data General NOVA, with its larger 16-bit word size and more advanced multiple-accumulator architecture, was introduced around 1970 and did catch on with a number of innovators. Among the first were Joe Wright at McMaster and Duncan Mellichamp at U.C. Santa Barbara. Wright was involved in the early development of

Data General’s Real-Time Operating System (RTOS), the first in a series of minicomputer software developments that allowed programs for laboratory experiments to be written in an organized (structured) manner.

When DEC later released the more powerful PDP-11 with its own real-time operating system, a second group of chemical engineering faculty chose to build on that system. Ed Freeh and John Heibel at Ohio State were workers in this group. Both the early DG and DEC systems supported real-time processing functions — analog-to-digital and digital-to-analog converters, time-keeping and interrupt-service routines directly accessible to the user/programmer, etc. During the early 1970s Hewlett-Packard, a traditional major player in laboratory instrumentation, introduced a line of special purpose computers for process monitoring and control, as did several other minor players who have long since disappeared into the economic black hole of technical successes but business failures.

Soon after 1970, the CACHE Real-Time Task Force was formed and began its early efforts to define the field and to provide guidance to those faculty looking for ways to utilize these new computer capabilities. At that time, at least four different manufacturers were providing computers roughly equivalent in terms of capability but totally incompatible with each other; the small chemical engineering user community was badly splintered as a result. Thus CACHE’s first efforts to provide guidance in this area were significantly complicated by the lack of standardization in the hardware area. Early efforts of the new Real-Time Task Force attempted to focus on two “demonstration projects,” the interfacing and use of a digital computer with (1) a gas chromatograph and (2) a laboratory-scale distillation column. Beyond several descriptive overviews and surveys, no specific CACHE products came out of this early period, although several workshops dealing with the principles of this new field were held under CACHE auspices.

Just as chemical engineering real-time users tended to fall into groups depending on their original choice of hardware — the PDP-11 users, the DG Nova/Eclipse users, and the rest (mostly IBM 1800 and HP stand-alone digital instrumentation systems) — the choice of software for user application programs was split, as well. The earliest applications on the smaller machines were forced to utilize assembly language as the programming medium because of the small amount of available main memory (initially 8 Kbytes). However, as soon as memory prices dropped (mid-70s), most users followed the IBM 1800 approach of using real-time versions of the FORTRAN language. One exception involved the use of various versions of real-time Basic, an interpreted language that was/is particularly useful in an educational format. During this period several academic projects led to published descriptions of undergraduate laboratory experiments suitable for demonstrations. For example, an NSF supported instructional project at UC Santa Barbara illustrated the use of a digital computer for the most important operations — analog data acquisition, binary input, output and logic, process control, etc. — with applications to three bench-scale undergraduate laboratory experiments. Documentation of this work was distributed to all CACHE participating schools at the close of the project.

The members of the NSF-required overview panel for the Santa Barbara project were (no accident here) virtually contiguous with the CACHE Real-Time Task Force. At one of their early meetings, this group discussed the idea of putting together a set of written materials that would deal with the many key principles and issues in this new area. Out of these discussions (and many future meetings of the group!), the CACHE Monograph Series in real-time computing, a set of eight volumes, was edited by Mellichamp and published by CACHE between 1977 and 1979. This series, in somewhat updated form, was published by Van Nostrand Reinhold as a professional text (Mellichamp, 1983). The book was chosen almost immediately as an "alternate selection" by one of the major computer book clubs. However, the chief importance of these materials was as a relatively early primer that helped many new users become established in the area.

Two stumbling blocks prevented many departments from introducing computer control in their laboratories during this period. One was the high cost of the hardware; a PDP-11 system with real-time hardware peripherals sold for about $20,000 in the late 1970s. The second problem was the high overhead in terms of effort required by faculty to become familiar enough with the hardware and software that it could be effectively used in a classroom or laboratory. A few chemical engineering faculty, nevertheless, came to real-time computing through research that required computer-based data acquisition and control facilities and through the efforts of their graduate students.

Change occurred rapidly following the introduction of the microprocessor. Industrially, control instrumentation hardware began to be replaced with microprocessor-based distributed computer control systems. These systems, however, still carried a heavy price tag due to their orientation toward large-scale continuous processing plants. The situation for smaller-scale academic research and instructional applications changed radically with the introduction of microprocessor-based personal computers, the topic of the next section.

The Microcomputer Revolution (1980s)

The late 1970s saw the introduction of the first microcomputers. While no different conceptually than their minicomputer predecessors, microcomputers possessed distinct advantages of scale and cost. Thus, many of the trends in laboratory automation one notes today did not come from the minicomputer applications of the 1970s, i.e., large, user-written programs (e.g., in FORTRAN), but rather reflect the microcomputer philosophy of very large main memory (> 1 Mbyte of RAM) used in conjunction with commercial software specifically tailored for data acquisition and control and for ease of user interactions.

The Apple II computer was introduced in 1977, the brainchild of two computer hackers working out of a garage. This microcomputer in a new package represented an astonishing step forward in that for the first time the average professional user could afford to purchase a computer, set it up, and run it without much prior knowledge of hardware and software. It was never really intended to be used for real-time computing applications in the laboratory. It did not even have a FORTRAN compiler, but it did come with a built-in Basic interpreter.

However, in addition to its ease of use, the designers of this revolutionary machine provided it with an open architecture that quickly led to the birth of a subsidiary multibillion dollar industry: making peripherals that extended the power of the basic system. Within a very short time period, new start-up companies began to manufacture real-time peripherals for the Apple II. As with the computer itself, these boards were priced low enough that one could set up a computerized data acquisition and control system at prices starting below $5000.

Many universities took advantage of these reduced costs and increased ease of use to equip their control laboratories with personal computer based systems. Joseph and Elliot (1984, 1985) describe an undergraduate process control laboratory built using bench-scale units interfaced to Apple II computers. One of the chemical engineering faculty members who embraced this new microprocessor technology was Peter Rony at Virginia Polytechnic Institute and University located in Blacksburg, Virginia. Under his leadership CACHE organized workshops on Microcomputer Interfacing for interested chemical engineering faculty.

IBM introduced their first personal computer in 1981, providing yet another breakthrough in cost and performance. Again, although this computer was intended to be a hobby and recreational machine, it found wide applications in professional and other areas, including the laboratory. Academic users desiring flexibility and low-cost preferred a data acquisition system built around these low-cost personal computers rather than around a dedicated industrial grade computerized Data Acquisition and Control (DAC) system. The latter provided performance and ease of use only with a large price tag and lack of flexibility. With each generation of microprocessor development, the increasing power of these personal systems accompanied by ever decreasing costs led to rapid popularity among academic scientists and engineers.

Simultaneously, another revolution was taking place in the software available for data acquisition and control applications. The need to write code in assembly language was replaced, first by Basic interpreter-based applications programming capability. With a suddenly large installed base of microcomputer users, many software companies were established that catered specifically to the Data Acquisition and Control market. Prominent among them was a company begun by Fred Putnam, a chemical engineering professor at MIT, now one of the leading suppliers of PC-based DAC software to industry (Laboratory Technologies, Inc.). Software developments have kept pace with the increasing power available as each successive generation of microprocessors has been commercialized.

With the availability of real-time control software, it became possible to introduce concepts of distributed computer control in undergraduate chemical engineering courses. Alan Foss at the University of California, Berkeley, developed a software package for his process control course which is currently available to other chemical engineering departments under license (Foss, 1989). The book by Joseph (1988) provides a good summary of the technology that was available in the 1980s for laboratory automation. The CACHE Anthology of Applications in On-Line Laboratory Automation (Mellichamp and Cinar, 1988) provides a number of illustrations of chemical engineering undergraduate laboratory experiments that were developed during this period.

Laboratory Automation Today (1990s)

Today, it is not surprising to walk into an undergraduate chemical engineering laboratory or a graduate research laboratory and see many of the experiments interfaced to microcomputer-based DAC systems. Computers are used routinely for acquiring data, processing it, and reporting the results. The software running the actual data acquisition and control functions hides much of the detail, freeing the user to focus on the experiment and the data rather than how it is acquired. These menu-driven packages provide “visual programming,” usually a way of connecting blocks or icons into an instrumentation block diagram. They offer on-line help to set up the DAC application so that a DAC system can be up and running in a few hours instead of the days, or even months, required in the past.

A typical laboratory data acquisition system today will be built around a desk top microcomputer package containing an Intel Pentium, IBM/Motorola Power PC, or equivalent class microprocessor with a few real-time peripherals, usually including industry standard 12-bit converters. A single card handling 16 channels of analog input, 4 channels of analog output, and 16 channels of binary input/output can be purchased for under $600. One probably will pay somewhat more for the software that is needed to set up the DAC system. DAC computers now can be easily networked to share data among multiple machines and to do remote data acquisition. Real-time software packages allow true distributed computing using these networked PC's. Interestingly, this concept is now used in industry to construct low cost distributed control systems. Thus the functionalities now available to the academic user are in no way restricted, either by hardware or by software, vis-à-vis industrial applications systems.

A key recent development on the DAC scene is the arrival of visual programming. Companies like National Instruments, Hewlett Packard, Iconics, Data Tranlation and Keithly-Metrabyte now offer software (LabView, LabWindows, HPVEE, Genesis, etc.) which further remove the programmer from the low-level assembly language and high-level FORTRAN programming of the past. These applications allow the user to construct a DAC program by connecting blocks together visually, in much the same way one would construct a functional block diagram in which each block provides certain capabilities such as analog read or write, data manipulation, data storage, or a simple controller function. Such applications quickly and easily allow the user to provide the DAC program a graphical front-end for communication with the operator — complete with indicators, strip-chart recorders, buttons, switches, and the like — all of which will be maintained in real-time during runs, including the provision of animation features, changing colors, etc. Thus the DAC programmer now constructs complete applications without the need to type-in long programs of code with the inevitable endless series of bugs and faults that must be removed before useful runs can be initiated. Since the "block diagram" now becomes both documentation and run-time program, a single session at the computer can encompass programming, running, debugging, and documenting "program" and results. This single incredible step has revolutionized DAC programming, making the writing and altering of programs extraordinarily easy and fast.

Another relatively recent development has been the increased use of realistic simulation exercises to teach principles of control. For example, the UC-ONLINE package from UC/Berkeley provides software that can simulate the start-up and control of distillation units, steam generation systems, inventory control and other processes. Purdue University has released a number of simulated "process modules" that students can exercise to obtain realistic plant data.

A second interesting addition has been the increased use of control design packages such as MATLAB (Mathworks, Inc.), MATRIX$_X$ (Integrated Systems, Inc.), and ACSL (MGA, Inc.), which reduce the time required to design and test new concepts and ideas. All of these systems are not yet fully integrated to real-time peripherals but soon will be. MATLAB is already provided with DACs capabilities, and in work conducted at the Ecole Polytechnique in Lausanne (Gillet et al, 1993), a version of MATLAB was interfaced to LabView, thus providing process interfacing flexibility with high-level analytical capabilities.

Finally, there has been a recent development in the field of software for the steady-state design of chemical plants and processes, e.g., ASPEN (AspenTech), HYSIM (Hyprotech), Chemcad (Chemstations) and PRO II (Simulation Sciences). At least two of these companies have announced the availability of dynamic simulators which offer the prospect of students having the capability of working with realistic plant-scale facilities and of investigating the effect of choosing alternative control structures, control algorithms, or controller settings on plant operability and profitability. These entirely realistic studies offer the advantage of providing industrial-level problems without having to provide inordinately expensive facilities (then only at pilot scale) in university laboratories.

The Future

The computer has now become an integral part of the chemical engineering laboratory both in data acquisition and in data processing. The power of both hardware and software continues to grow at an ever increasing pace. Certainly these developments are going to impact even more on the future chemical engineering laboratory.

We expect to see more and more realistic simulation of both laboratory-scale equipment as well as plant wide simulations, so that students will be able to experiment with new concepts and ideas in the laboratory — assembling and building, then collecting data on simulated experimental setups. Data analysis and simulation tools will enable students to match their models with the data that they collect. Similarly they will be able to experience a plant start up and the procedure of bringing it under control from the computer console.

It is likely that sensors of the future will have their own computers to do self diagnosis, calibration, and error detection. Interfacing will become synonymous with data communication. These changes generally will continue to reduce the time and effort needed to set up DAC systems.

The control laboratory will most likely utilize interfaces similar to those found in industrial DCS installations. Thus students will be able to configure, implement, and test control system designs on processes similar to the ones they will encounter when they go into practice. The availability of advanced artificial-intelligence-based software integrated with control system design and simulation packages should allow students to develop and test complex control system designs in short laboratory sessions.

In summary, the future looks quite exciting for laboratory automation. The features mentioned here are on the verge of reality even now.

References

Aspen Technology, Inc., *Aspen Plus* Software, 10 Canal Park, Cambridge, MA 02141

Chemstations, Inc., *Chemcad* Software, 10375 Richmond Ave, Suite 1225, Houston, TX 77042

Data Translation, *DT-VEE* Software, 100 Locke Dr., Marlboro, MA 01752

Foss, A. (1987). *UC-ONLINE*: Berkeley's Multiloop Computer Control Program, *Chemical Engineering Education,* Summer.

Gillet, D., C. Salzmann, R. Longchamp and D. Bonvin (1993). A Methodology for Development of Scientific Teachware with Application to Adaptive Control, *American Control Conference*, San Francisco.

Hewlett Packard, *HP-VEE* Software, 1501 Page Mill Road, Palo Alto, CA 94304

Hyprotech Inc., *Hysim* Software, Alberta, Canada

Iconics, Inc., *Genesis Control Series*, 100 Foxborough Blvd., Foxborough, MA 02035.

Integrated Systems, Inc., *MATRIXX*, Santa Clara, CA 95054

Joseph, B. (1988). *Real-Time Personal Computing for Data Acquisition and Control*, Prentice Hall, New Jersey.

Joseph, B. and D. Elliott (1984). Experiments in Process Control Using Personal Computers, *Chemical Engineering Education*, **18**(4).

Joseph, B., D. Millard and D. Elliott (1985). Experiments in Temperature Measurement and Control Using Microcomputers, *Control Systems*, **5**(3).

Keithly-Metrabyte, *ViewDAC* Software, 440 Myles Standish Blvd, Taunton, MA 02780

Laboratory Technologies Inc., *LabTech* Software, 400 Research Dr., Wilmington, MA 01887

Mathworks, Inc., *MATLAB* Software, Cochituate Place, 24 Prime Parkway, Natick, MA 01760

Mellichamp, D.A., Ed. (1977-1979). Monograph Series in *Real-Time Computing*, Vol's I-VIII, *CACHE*, Austin, TX.

Mellichamp, D.A., Ed. (1983). *Real-time Computing with Applications to Data Acquisition and Control*, Van Nostrand Reinhold, New York.

Mellichamp, D.A., and A. Cinar, Ed's. (1988). On-Line Computer Applications in the Undergraduate Chemical Engineering Laboratory: A CACHE Anthology, *CACHE*, Austin, TX.

MGA, *ACSL* Software, 200 Baker Ave., Concord, MA 01742.

National Instruments, *LabWindows* Software, 6504 Bridge Point Parkway, Austin, TX 78730.

CACHE AND THE USE OF COMPUTER NETWORKS FOR EDUCATION

Peter R. Rony
Virginia Polytechnic Institute
Blacksburg, VA 24061

Abstract

Though CACHE was a pioneer in the use of electronic mail in chemical engineering education, it did not play a role in the broader networking community. The promise of worldwide networking and communication of digital text, image, animation, audio, and video files has arrived with World Wide Web (WWW). Internet software – Mosaic, Netscape, HGopher, WS_FTP, and others – for a Windows platform is described.

CACHE Contributions to Chemical Engineering Electronic Communications

Within the field of chemical engineering, CACHE preceded AIChE and other chemical engineering organizations in championing electronic communications, *viz*., electronic mail. CACHE, to use a familiar cliché, kept the torch lit within chemical engineering; CACHE, to use an unfamiliar cliché, was probably the first chemical engineering organization to pave its minuscule portion of the information superhighway with a bit of asphalt, which rapidly developed potholes. The role of CACHE in the broader networking community within science and engineering was negligible.

Phase 1. The CACHE National Electronic Mail Experiment: ITT Compmail+™

To quote the April 9, 1985 CACHE Electronic Mail Task Force proposal (Rony, 1985):

> "The objectives of the CACHE National Electronic Mail Experiment are: (1) to catalyze the creation of a widely used national chemical engineering electronic mail network between academic, industrial, and government sites, (2) to issue a report that documents concrete examples of how to use electronic mail, identifies potential chemical engineering applications, and lists typical costs, (3) to extend such a network to our international colleagues, (4) to publicize the use of electronic mail in CACHE, CAST, AIChE, and other chemical engineering organizations, (5) to identify important uses for electronic mail in chemical engineering, (6) to perform limited tests of alternative mail services, and (7) to publish an article on electronic mail in *Chemical Engineering Progress*."

Initial publicity for the CACHE National Electronic Mail Experiment appeared in *CACHE News* (September 1984) and at the CACHE Reception, 1984 AIChE Annual Meeting

(San Francisco, November 1984). The participation of CACHE in electronic communications within chemical engineering can be subdivided into at least four phases:

The first CACHE experiment in electronic communications was conducted during 1984-85 (Rony, Hale and Wright, 1986a). Though members of the chemical engineering community were invited to participate, few colleagues did so, other than members of the CACHE Board of Trustees, each of whom was given both a Compmail+ account and instructions how to use it. As of January 31, 1986, CACHE sponsored 55 Compmail+ accounts, only 20 of which were ever used. The experiment was not successful. The most successful use of Compmail+ was the transfer of files for *CAST Communications*. By April 1986, the experiment was concluded.

Phase 2. The CACHE National Electronic Mail Experiment: BITNET

Phase 2 of the CACHE National Electronic Mail Experiment started with the publication of excerpts from the *Science* article, “Computer Networking for Scientists,” (Rony, Hale and Wright, 1986b) in the April 1986 issue of *CACHE News* (Rony, 1986). Several steps were recommended for chemical engineering departments, including the submission of faculty userids to the CACHE Electronic Mail Task Force. For historical purposes, it is appropriate to note that the initial list of BITNET userids (Rony, 1986) included eighteen names at fifteen universities and one corporation in four countries. Technion - Israel Institute of Technology - was the leader, with three faculty represented.

Late in 1984, when the CACHE National Electronic Mail Experiment was first proposed, John Seinfeld (Cal. Tech.) suggested that the Electronic Mail Task Force find ways to involve students. The result was the CACHE Computer Networking Student Contest, which was first announced in the September 1986 issue of *CACHE News* (CACHE, 1986). Unfortunately, little interest was generated, and the minimum requirement of at least 4 student-chapter entries was not met.

Phase 3. The CACHE National Electronic Mail Experiment: IBM Grand at Louisiana State

A serious attempt was made by IBM and Louisiana State University to promote wide-area networking within chemical engineering through the testing of experimental IBM network server software called GRAND. Status reports, announcements, a user’s manual, and userid lists were made available by CACHE through CACHE News, the CACHE reception at Annual AIChE meetings, and by direct mail to the CACHE representatives in chemical engineering departments (Rony, 1987; Rony, Wright and Rawson, 1987a, 1987b; Reible, 1987, 1988; Cutlip, 1987; Reible and Rony, 1988). Unfortunately, the IBM/LSU GRAND experiment was not successful; GRAND was infrequently used by chemical engineers and was eventually abandoned.

Phase 4. Creation and Maintenance of BITNET/Internet Userid Lists

Though not successful as a wide-area network file server for the chemical engineering community, the IBM/LSU GRAND experiment did succeed in promoting substantial interest in BITNET among chemical engineering faculty and departments. Perhaps the most significant contribution to *CACHE News* by the task force was the two-part “A User’s Guide to Electronic Mail” (AAS, 1989, 1990), which was published through the permission and kindness of the American Astronomical Society (AAS). Concerning electronic communications, it is worth noting how forward looking the AAS was with its 1989 *Electronic Mail Directory*, a distinct contrast to the extreme conservatism (which continued through 1993) of the AIChE.

Starting in fall 1989, the focus of the CACHE Electronic Mail Task force shifted to the creation and maintenance of BITNET/Internet userid lists for both chemical engineering faculty and chemical engineering departments. The Fall 1989 issue of *CACHE News* reported on the validation of BITNET userids, including the significant contribution of Robert Brodkey of Ohio State University to this effort (Rony, 1989b).

During Phase 4, the only constant about the task force identity was change: the task force name changed from the “Electronic Mail Task Force” at its creation in 1985 to the “Electronic Communications Task Force” in 1989 and finally to the “Electronic Networking Task Force” in 1990. The use of the term *networking* marked a degree of maturity in the task force’s perception of its technological niche. The task force chairman (Rony) would like to acknowledge his task force colleagues during the first four phases of task-force activities: Joe Wright, John Hale, Norman Rawson, Robert Brodkey, John Hassler, and Wayne Crowl. All were influential in identifying the task force mission statement, vision, and key results.

John Hassler, in the fall 1989 issue of *CACHE News*, contributed comments on UUNET (Unix Users Net), sometimes known as USENET, which was probably the second nationwide network after ARPANET. Having done so, he inquired, “Don’t ChE types have anything to discuss?” (Rony, 1989b) Richard Zollars established a discussion group, IFPHEN-L, for those interested in interfacial phenomena. At the invitation of the task force, he published an article entitled “E-mail Discussion Groups: A List Owner’s Perspective.” Zollars (1990) wrote: “At the moment, I receive approximately one to two requests per week for subscriptions and only about one message per month. Thus, the time that I spend administering the mail list is less than an hour every month. I would prefer more business …”

Phase 5 (see below) was previewed by a March 6, 1989 report to the CACHE Executive Committee by the Electronic Communications Task Force. From an article by Susan Winitsh, the following were noted (Rony, 1989a; Wintsch, 1989):

> “A new paradigm is emerging: the *national collaboratory*, a framework in which scholars across the nation interact as if they were across the hall from one another, and alternatively called ‘geographically-distributed problem solving’ by Peter Denning.”
>
> “Another scenario: An accelerated version of the age-old scholarly process of acquiring and disseminating knowledge and information.”

Additional observations and conclusions were (Rony, 1989a):

> “Most (if not all) recent chemical engineering efforts to develop ChE-specific bulletin boards and wide-area network file servers - to state the matter bluntly - have not been successful, nor is there any reason to believe that they will become successful during the next 12 months.”
>
> “Most chemical engineers, so far, have not adapted to the emerging style of rapid, interactive, electronic communications and collaboration. We probably are not the only academic discipline in which such a situation exists.”

Phase 5. Chemical Engineering Educators on Internet

Phase 5 will succeed for CACHE when the organization, or its members, actively use the broader electronic capabilities of the Internet, not just electronic mail. Such capabilities are discussed in "Current Practice" later in this article. Several contributions to *CACHE News* (Rony, 1991; Kim, 1992a, 1992b), and a proposal by a CACHE trustee (Kantor, 1993), predicted a significant shift in task force interests from those in Phase 4.

Anonymous FTP servers were the subject of Sangtae Kim's article in the Fall 1992 issue of *CACHE News* (Kim, 1992a).

> "There are now literally thousands of anonymous FTP servers in the USA and beyond, containing a veritable smorgasbord of software packages: plotting routines, 3-D graphics and visualization packages, spreadsheets, symbolic algebra programs, compilers, desktop publishing software, and yes, even some nifty games. The only way to keep track of them is with the help of (what else?) anonymous FTP servers!"

At the summer 1993 Mt. Crested Butte CACHE trustees meeting, Jeffrey Kantor proposed that CACHE support the electronic distribution of its software and documentation via the Internet (Kantor, 1993).

> "The types of information amenable to electronic distribution include: (a) case studies, (b) educational software, (c) phone book information on CACHE contributors, (d) articles from past issues of *CACHE News*, (e) meeting information and announcements, and (f) links to other sources of information."
>
> "The initial task of establishing a Gopher and FTP site is straightforward and would be easily accomplished. The more difficult issues deal with the copyright and ownership of freely-accessible information …"

Sangtae and Jeffrey were on target. FTP, Gopher, name servers, and other Internet software and protocols - especially the World Wide Web (WWW) - represent the expanded world of electronic networking for chemical engineering educators.

Current Practice: The Wide, Wide World of Internet

What is Internet?

According to the AAS, in *A User's Guide to Electronic Mail* (Part 1) (AAS, 1989):

> "The Defense Advanced Research Project Agency (D)ARPA network was initially set up by the U.S. Department of Defense in 1969. It is now a part of the ARPA Internet, which uses *TCP/IP* (Transmission Control Protocol/Internet Protocol) communications and includes over 30,000 hosts (1987) and more than 570 networks in several domains: **COM** (commercial organizations), **EDU** (educational/research organizations), **GOV** (civilian government organizations), **MIL** (Department of Defense), **ORG** (other organizations), **NET** (network resources)."
>
> "Most Internet network sites that astronomers communicate with will be in the **EDU** domain (universities, national observatories). There are additional do-

> mains for countries outside the USA, e.g., **UK** (United Kingdom) and **AU** (Australia). Internet includes some transcontinental and transatlantic satellite links (SATNET). Typical delivery time on Internet is a few minutes."
>
> "In Internet, individual computers are assigned numerical addresses within a hierarchical system, with the first number in the address being the number of the individual network on Internet. For example, 4.0.0.0 is SATNET, 10.0.0.0 is the ARPA network, 128.112.0.0 is the Princeton network, and 128.112.24.2 is an individual machine at Princeton. These addresses are mapped against alphanumeric addresses via host tables. Thus, the machine 128.112.24.2 corresponds to pupgg.princeton.edu."
>
> "Internet is the fastest growing of the United States networks and presently is supported by DARPA, the National Science Foundation, NASA, the Department of Energy, and the United States Geological Service. NSF has the mandate to support national networking for the scientific research community …"

As additional examples, the author's Internet nodes are the IBM mainframe at Virginia Tech., VTVM1.CC.VT.EDU (128.173.4.1) and a Hewlett-Packard server, VT.EDU. In February 1994, he installed an IBM ValuePoint 486DX, which functions as an Internet node named RONY.CHE.VT.EDU (128.173.164.0), in an undergraduate controls laboratory.

What are the Basic Applications of Internet?

A superb, early text on the subject of Internet on Unix platforms is *The Whole Internet* by Ed Krol (Krol, 1992). In subdividing the world of Internet, important categories for the individual user include (a) computer platform, (b) communications hardware, and (c) Internet software. For category (c), Krol identified the following Internet applications and created the following individual chapters for each: Remote Login, Moving Files, Electronic Mail, Network News, Finding Software, Finding Someone, Tunneling Through the Internet, Searching Indexed Databases, Hypertext Spanning the Internet, and Other Applications.

For readers who either have Unix workstations or plan to use Unix for Internet communications and desire a comprehensive perspective on most key aspects of Internet, Krol's book is excellent (Krol, 1992). Rather than re-invent the wheel in this page-limited chapter, the author has decided to ignore Unix-workstation-based Internet and instead to focus on an alternative, more popular platform - Windows - for Internet applications. Current Internet software development for Windows is in a rapid state of flux; it is an exciting time for chemical engineering colleagues who use IBM PC machines or clones.

Our discussion of Internet for Windows starts with Winsock, which is the key to all other Internet applications software. The focus then shifts to three interesting Internet software packages for Windows, namely, WS_FTP, HGopher, and Mosaic. Space is not available to discuss the remaining Windows Internet applications software: Trumpet, QWS3270 Telnet, Eudora, WFTPD, Ping, Finger, and Archie. Krol's outstanding book is extensively excerpted in this chapter (Krol, 1992).

The Standard TCP/IP Windows Interface (TRUMPET WINSOCK Software)

According to Harry Kriz in his useful electronic document, “Windows and TCP/IP for Internet Access,” (Kriz, 1994):

> “‘Winsock’ is the buzzword that dominates discussion about TCP/IP and Windows … applications … In order to get these applications working, there are only a few things that an end-user needs to know about Winsock and how it supports Windows applications.
>
> “In layman’s terminology, the term ‘Winsock’ refers to a technical specification that describes a standard interface between a Windows TCP/IP application (such as a Gopher client) and the underlying TCP/IP protocol stack that takes care of transporting data on a TCP/IP network such as the Internet. When I invoke a program such as HGopher, it calls procedures from the WINSOCK.DLL dynamic link library. These procedures in turn invoke procedures in the drivers supplied with the TCP/IP protocol stack. The TCP/IP drivers communicate with the computer’s Ethernet card through the packet driver. For serial line communications, the TCP/IP drivers communicate with a SLIP driver to enable network communications through the serial port.
>
> “The WINSOCK.DLL file is not a generic file that can be used on any system. Each vendor of a TCP/IP protocol stack supplies a proprietary WINSOCK.DLL that works only with that vendor’s TCP/IP stack.
>
> “The advantage to the developer of a Winsock-compliant application program is that the application will work with any vendor’s Winsock implementation … It is this aspect of the Winsock standard that has resulted in the blossoming of Winsock-compliant shareware applications since the summer of 1993.”

The dominant Winsock was Peter Tattam’s *Trumpet Winsock*, which included a TCP/IP protocol stack and basic clients such as Telnet, FTP, Ping, and Archie. For a shareware fee of $20, version 1.0A was available by anonymous FTP as files twsk10a.zip (February 3, 1994) and winapps.zip (November 30, 1993) by anonymous FTP from FTP.UTAS.EDU.AU in subdirectory /PC/TRUMPET/WINSOCK or by Gopher from INFO.UTAS.EDU.AU under menu item Utas FTP Archive (Kriz, 1994).

What is a Packet Driver?

In order to run Trumpet Winsock, you must either have a serial communications port, which permits use of a serial-link interface protocol (SLIP), or preferably an Ethernet hardware communications card, which requires the use of a packet driver for both the card and for WINPKT.COM, a virtual packet driver interface for Windows 3.1.

A *packet driver* is “a small piece of software that sits between your network card (e.g., Etherlink III) and your TCP program. This driver provides a standard interface that many programs can use in a manner analogous to BIOS calls using software interrupts.” (Tattam, 1994). For the 3Comm Etherlink III interface card, two lines in the DOS file, AUTOEXEC.BAT, do the trick:

```
3C509 0x60
WINPKT 0x60
```

0x60 is a software interrupt vector with a hexadecimal code of 60H. Other types of packet drivers can be obtained from the Crynwr Packet Driver Collection available over Internet.

Moving Files: Anonymous FTP (WS_FTP Software)

One of the jewels in the Internet application suite is a program that allows you to use anonymous FTP to transfer files from another Internet node to your node.

> "The essential idea of anonymous FTP is that user accounts and passwords are not required. The user's account is replaced by the generic name account, *anonymous*." (Kim, 1992a).
>
> "The File Transfer Protocol was originally a Unix utility used for interactively transferring files. High-quality, public-domain clients and servers are available for most computing platforms. The advantages of FTP are its interactivity and the ability to transfer binary files. The main disadvantage is that the required interactive access to the Internet is universal. FTP is probably the most widely used mechanism for software distribution on the Internet" (Kantor, 1993).

With DOS-based anonymous FTP, a knowledge of FTP commands - e.g., **ftp, get, put, quit, bye, cd, binary, ascii, dir, ls, lcd, mput, mget, close** - is required. With Windows-based FTP, and specifically with John Junod's **WS_FTP** software (Junod, 1994), intuitive mouse point-and-click operations, menus, and other graphical-user-interface (GUI) features substantially simplify the task of transferring files over the Internet. The detailed treatment of Unix instructions provided in Chapter 6 of reference (Krol, 1992) is no longer required for the Windows platform. FTP client **WS_FTP** is available as file ws_ftp.zip (February 9, 1994) by anonymous FTP from FTP.USMA.EDU in subdirectory /PUB/MSDOS/WINSOCK.FILES (Kriz, 1994).

Tunneling Through the Internet: Gopher (HGopher Software)

> "Gopher, or more accurately, 'the Internet Gopher,' allows you to browse for resources using menus. When you find something you like, you can read or access it through the Gopher without having to worry about domain names, IP addresses, changing programs, etc. For example, if you want to access the on-line library catalog at the University of California, rather than looking up the address and telnetting to it, you find an entry in a menu and select it. The Gopher then 'goes fer' it.
>
> "The big advantage of Gopher isn't so much that you don't have to look up the address or name of resources, or that you don't have to use several commands to get what you want. The real cleverness is that it lets you browse through the Internet's resources, regardless of their type, like you might browse through your local library with books, filmstrips, and phonograph records on the same subject grouped together …
>
> "… Think of the pre-Gopher Internet as a set of public libraries without card catalogs and libraries. To find something, you have to wander aimlessly until you stumble on something interesting. This kind of library isn't very useful, unless you already know in great detail what you want to find, and where you're likely to find it. A Gopher server is like hiring a librarian, who creates a card catalog

subject index … Unfortunately, Gopher services did not hire highly trained librarians. There's no standard subject list, like the Library of Congress Subject Headings, used on Gophers to organize things … Gopher does not allow you to access anything that couldn't be made available by other means. There are no specially formatted 'Gopher resources' out there for you to access, in the sense that there are FTP archives or white pages directories … Gopher knows which application (telnet, FTP, white pages, etc.) to use to get a particular thing you are interested in and does it for you. Each type of resource is handled a bit differently. However, they are all handled in an intuitive manner consistent with the feel of the Gopher client you are using" (Krol, 1992).

Martyn Hampson's *HGopher*, Version 2.4, is currently the best Gopher+ client for Windows 3.1 and Winsock (Hampson, 1994). It is available by anonymous FTP as file hgopher2.4.zip from BOOMBOX.MICRO.UMN.EDU in subdirectory /PUB/GOPHER/WINDOWS. Mr. Hampson suggests that you donate $10.00 to your favorite charity if you like *HGopher* (Kriz, 1994). [*Editor's note:* The appearance of "search engines" such as *Yahoo* and AltaVista during 1995 and 1996 has made Gopher largely obsolete.]

Hypertext Spanning the Internet: World-Wide Web (Mosaic Software)

The *World-Wide Web (WWW)* is based upon hypertext, a software technology first made popular on the Macintosh platform.

"Hypertext is a method of presenting information where selected words in the text can be 'expanded' at any time to provide other information about the word. that is, these words are links to other documents, which may be text, files, pictures, anything" (Krol, 1992).

Krol has done an effective job of distinguishing between WWW and Gopher; rather than rephrase his comparison, we quote it directly (Krol, 1992).

"What is WWW about? It's an attempt to organize all the information on the Internet, plus whatever local information you want, as a set of hypertext documents. You traverse the network by moving from one document to another via 'links.' … Your home page is the hypertext document you see when you first enter the Web.

"… While there are a lot of similarities, the Web and Gopher differ in several ways. First, the Web is based on hypertext documents, and is structured by links between pages of hypertext. There are no rules about which documents can point where - a link can point to anything that the creator finds interesting … The Gopher just isn't as flexible. Its presentation is based on individual resources and servers. When you're looking at an FTP resource, this may not make much of a difference; in either case, you'll see a list of files. But the Gopher doesn't know anything about what's inside of files; it doesn't have a concept of a 'link' between something interesting on one server, and something related somewhere else.

"Second, the Web does a much better job of providing a uniform interface to different kinds of services. Providing a uniform interface is also one of the Gopher's goals; but the hypertext model allows the Web to go much further. What does this mean in practice? For one thing, there are really only two Web commands: follow a link … and perform a search … No matter what kind of resource you're using, these two commands are all you need. With Gopher, the interface tends to change according to the resource you're using."

The markup language used by the World-Wide Web is called *HTML*, or *HyperText Markup Language. A Beginner's Guide to HTML* describes the minimal HTML document, including titles, headings, and paragraphs; linking to other documents, including the uniform resource locator (URL) and anchors to specific sections; additional markup tags for lists, preformatted text, extended quotes, character formatting, and special characters; in-line images; external images; troubleshooting; and ends with a comprehensive example (NCSA, 1993).

NCSA *Mosaic* for Windows is available by anonymous FTP server as file mos_20a1.zip from NCSA's FTP server, FTP.NCSA.UIUC.EDU in subdirectory PC/MOSAIC (Wilson and Mittelhauser, 1994). Reference (NCSA, 1993), on HTML, is available from PUBS@NCSA.UIUC.EDU. [*Editor's note:* A commercial browser, *Netscape*, available for virtually all hardware platforms, replaced Mosaic as the dominant Web browser during 1995 and 1996. A new Microsoft browser, *Explorer*, targeted for use with Windows on PC compatibles, was introduced in 1996, and has achieved significant success in the marketplace.]

Other Sources for Winsock and Internet Information

It is appropriate to conclude this section of the chapter with a final mention of Harry Kriz's, *Windows and TCP/IP for Internet Access* (Kriz, 1994). Harry (1) identified key Windows/Winsock software and described his experience with each briefly, (2) identified for each software package the author, version, license arrangement, file name/date, and availability at a specific anonymous FTP or Gopher site, and (3) provided readers with suggestions for other sources of Winsock information. For additional information, contact Harry M. Kriz, University Libraries, Virginia Polytechnic Institute & State University, Blacksburg, VA 24061-0434. Email: hmkriz @ vt.edu. In 1996, Harry received about 20,000 WWW "hits" per month.

Internet reference books were a growth industry in 1994. Consider *The Windows Internet Tour Guide* by Michael Fraase (Fraase, 1994); *Using the Internet* by Tolhurst, Pike, Blanton, and Harris (Tolhurst et. al., 1994); *Navigating the Internet* by Mark Gibbs and Richard Smith (Gibbs and Smith, 1993); and *The Internet Directory* by Eric Braun (Braun, 1994).

The Future: Internet and Chemical Engineering Education

How will Internet and networking affect chemical engineering education? The author suspects that Internet has not had the impact within both chemical engineering education and research that it has already had in other fields – e.g., computer science, physics, astronomy, electrical engineering – that are more attuned to the computer networking revolution. Our national society, AIChE, has been slow to participate in Internet. Where on the information superhighway have been our profession's leaders? Perhaps riding bicycles on side streets.

A revolution in the convenience of Internet software is now occurring. Command-line oriented DOS and Unix Internet clients are rapidly giving way to Unix, Macintosh, and Windows graphical user interface (GUI) clients. Internet users on the Windows platform now have the convenience of software such as Trumpet Winsock, Trumpet, Mosaic, HGopher, WS_FTP, WS_PING, WFTPD, Eudora, Netscape, and others. Internet usage has become fun, as opposed to being tedious under DOS and under Unix before X-Window. Most of the new Internet Windows/Winsock software is either freeware, shareware, or requires a nominal license fee of under $50.

Internet users no longer need to participate in the Unix culture in order to enjoy the benefits of Internet; Unix workstations are no longer required, nor is there a need to remember numerous Unix commands. Internet software on both the Macintosh and Windows platforms is already excellent and will soon be better. With the expected merging of platforms, e.g., as promoted by the PowerPC chip vendors, many more users will be able to select the platform "identity" of their PC without purchasing entirely different hardware. Internet applications software will be outstanding on any platform that a user selects.

When we consider the potential impact of Internet on chemical engineering education, it is appropriate to consider this issue from the point of view of our customers, students. There are far more questions than answers. Some of the questions are provocative.

Will student access to the Internet be direct or indirect? In other words, will students (a) own their personal computers and use them in their rooms to access the Internet anytime during the day? (b) use personal computers networked to the Internet only in university computer laboratories? (c) depend upon intermediaries - e.g., chemical engineering faculty - to utilize Internet on their behalf?

Who will pay the costs of access to Internet? Will students (a) pay a monthly charge for high-speed modem or local-area network access to the Internet directly from their own rooms? (b) pay a semester laboratory charge for access to the Internet through university laboratories? (c) depend upon the chemical engineering department to pay for access to the Internet?

What will be the trade-off between computer storage (memory) costs for digital information and the convenience of Internet access for such information? Will the convenience and low cost of write-once or read/write optical memory be such that downloaded files will be stored optically rather than on more expensive magnetic disk? Will the convenience, speed, accessibility, and low cost of Internet be such that neither optical nor magnetic local storage are required at all? In other words, have the CD-R and CD-ROM already been supplanted by the Internet? Is mass local storage at an individual personal computer required anymore?

What types of information will be available to students and faculty through the Internet? Will changes in educational paradigms occur as a direct consequence of the Internet? How will the Internet affect the creation, testing, marketing, distribution, sale, and use of textbooks? How is the Internet already affecting the creation, testing, marketing, distribution, sale, and use of computer data (programs, images, video, audio, text, presentations, executables, and so forth)? How will the Internet affect the need for a student to physically attend lectures on a university campus?

Recall the article by Susan Winitsh (Wintsch, 1989; Rony, 1989b) in which it was noted that a new paradigm is emerging:

> "the national collaboratory, a framework in which scholars across the nation interact as if they were across the hall from one another."

Will the Internet allow several chemical engineering departments, including both faculty and students, to collaborate during a semester in a semester course that is jointly offered by all of the departments? Will we have an "educational collaboratory?" How will the Internet affect the need for a chemical engineering department to maintain departmental faculty competence to teach all required courses in the curriculum? Once the Internet becomes widely available to students with fast service, will a chemical engineering department need as many teaching faculty as it has in the past?

In chemical engineering education, what are the educational values of a printed textbook, of an electronic textbook the pages of which can be printed on demand, of personal contact with a faculty member, of impersonal contact with a faculty member in a lecture with large class size, of personal contact with graduate teaching assistants, of chemical engineering applications software, of homework assignment evaluation, of taking handwritten notes in a lecture, and so forth?

Greater attention should be paid to electronic information as a valuable resource. Chemical engineering authors should be careful to archive files of their manuscripts no matter what application software generated them. Handwritten answer books for chemical engineering textbooks should be scanned, compressed, archived, and made available to faculty in electronic form. Entire textbooks should be made available as electronic documents that can be communicated over the Internet under controlled conditions.

Page limitations prohibit speculation on most of the above questions. However, one speculation is offered.

Speculation: The Marketing of Chemical Engineering Textbooks

Assume that you desire to create a new chemical engineering textbook. Your primary obstacle will not be the creation of the textbook (it probably is already 50% finished based upon your lecture notes over the past several years) nor the creation of camera-ready copy (you are already proficient at desktop publishing). The primary obstacle will be the *marketing* of the textbook. You need to convince a major technical publishing company - e.g., McGraw-Hill, Prentice-Hall, Wiley, or several others in chemical engineering - that it should publish your new textbook. The likelihood is that none of them will do so. Why not? They already each have their in-house example of a textbook in your field, and do not need to add a second (or third) title.

The major publisher may have your textbook proposal reviewed. All such reviews will be subjected to the prevailing dogma of current experts in the field. Proposals that either deviate significantly, or do not present any change from current dogma and paradigms, will likely be rejected. You may feel completely dependent today on the marketing strengths of the major publishers of chemical engineering textbooks. If no publisher is interested in your proposal, is your wonderful work dead in the water?

Perhaps not. Enter the Internet. Make the assumption that Internet nodes are widely available at all major chemical engineering departments and that, at long last, most ChE faculty are both computer-literate and Internet-literate. Does such a situation permit a new and different marketing paradigm for chemical engineering textbook authors? Certainly. It is a new marketing paradigm that does not depend upon any of the major publishers, as we currently know them.

Consider software shareware and freeware. In each case, copyright control does exist and is held by one or a few individuals (or, on occasion, companies). Yet the marketing (distribution) mechanism is by the Internet or by CD-ROM. Potential users have the opportunity to test the software before deciding to use it. If it is freeware, no royalty payment is required. If it is shareware, a specified royalty payment is requested. An example is Peter Tattam's Trumpet Winsock, which requires a $20 license fee after 30 days of use.

The identification and announcement of shareware licensing fees already means that the Internet is widely used for the marketing of software. This precedent being established, it should be possible to use the Internet for a productive educational and scholarly purpose, namely, the marketing of "textbook shareware" to universities. The broader ethics and value of marketing textbook shareware via the Internet is identical, in the author's opinion, to the ethics and value of marketing computer shareware. Universities can easily police the unauthorized use of textbook shareware in a course.

Consider a textbook that is normally marketed by a major publisher in printed form for a list price of $60. Add state tax to the $60. Assume a royalty payment to the author of 10% of the wholesale price of $45, or $4.50 royalty per book.

Now consider the same textbook marketed and distributed in electronic form - for example, Adobe Acrobat Portable Document Format (PDF) - over the Internet for a single-user license (royalty) fee of $4.50. The recipient of the electronic textbook has the option of printing it or retaining most pages in electronic form. The cost of such printing could be $5 to $20. The result of this alternative paradigm is that the textbook author receives an identical royalty, but students pay less than half the cost for a textbook that is not quite as convenient to use and certainly not as permanent.

The trade-off? Textbook cost to student versus convenience, form, and permanence. Will this paradigm occur? Certainly. Will it eventually replace the printed textbook publisher? Who knows?

For the textbook author there are several advantages associated with the new paradigm. No longer will the major textbook publishers have veto power over the publication of a new textbook. License (royalty) fees per electronic textbook could well be higher than for a textbook printed by traditional publisher. The textbook could be revised annually. Copyright control of the textbook could remain in the hands of the author(s). The textbook could contain substantial color, audio, and video by virtue of its electronic form. A new marketing channel, the Internet, makes this all possible.

When you think of the Internet, think of *marketing*. Whether or not the rules of the Internet permit such marketing today, it is clear that the needs of the information superhighway will change such that marketing will become a (perhaps *the*) major component in the future. And when you think of marketing, think of *entrepreneurship*. Internet potentially is liberating to textbook authors, just as it has been for computer shareware authors.

Jeffrey Kantor is certainly correct in his assessment of the Internet situation (Kantor, 1993). The time has come for CACHE to support the electronic distribution of documentation, and perhaps selected software, via the Internet. FTP, Gopher, WWW, Netscape, WAIS and other services can all be seriously considered today. Anticipated problems with copyright issues or the possible loss of revenue for software are non-issues, in the author's opinion. There is sufficient electronic information that can be freely disseminated to anyone, not just to chemical engineers, to create a viable CACHE.ORG server. Software that has commercial value or copyright sensitivity can be distributed by more traditional, controllable channels (e.g., diskettes, CD-ROMs). The emergence of CD-Rs and CD-ROMs and low-cost, high-capacity removable magnetic diskettes (e.g., Iomega's Zip "floppies") eliminate the 1.44-MB barrier of high-density diskettes for program storage.

Addendum

This manuscript was originally prepared in 1994. As of late 1996, substantial changes have occurred in the computer and networking landscape. The author would like to connect some points made in the paper to current trends in the computing environment:

1. Within chemical engineering education, the use of the Internet - WWW, email, and FTP - has become both obvious and pervasive. The change in computing perspective, away from laboratory-based PCs, may well become a "paradigm shift" in higher education by the end of the century.
2. Netscape and Microsoft Explorer have replaced Mosaic as the dominant WWW browsers.
3. New WWW data types – real audio, VRML (Virtual Reality Markup Language), and perhaps others – have appeared.
4. The programming language, *Java*, introduced in early 1996 by Sun Microsystems, now represents a potential threat to Microsoft's dominance. The author hopes that Netscape Corporation and Sun MicroSystems succeed in their revolutionary transformation of Internet usage.
5. The author's "Speculation: The Marketing of Chemical Engineering Textbooks" has led directly to an entrepreneurial activity to promote such a possibility. The author has applied for a Federal trademark at the USPTO for the service mark, Sharebook (TM), and also has acquired the domain name, sharebook.com. A sharebook site at URL http://www.sharebook.com is currently under construction in late1996.
6. A new class of computer hardware, the "network computer" or "NC", is about to emerge on the computing scene. It represents one answer to a question posed in the original manuscript: Is local optical or magnetic storage required on every computer?
7. The dominant software applications in 1996 are word processing, email, the World Wide Web, the spreadsheet, and presentation software.
8. The 1980s ethical concerns about "marketing" on the Internet have disappeared by 1996. The dominant domain is already *.COM, which has supplanted the *.EDU and *.ORG domains in number of servers by perhaps a factor of ten to one hundred.
9. Secure, perhaps encrypted, electronic "digital cash" should arrive on the network

computing scene sometime during 1997. The potential for entrepreneurial activity on the Internet will become compelling.

10. The original manuscript statement, “When you think of the Internet, think of marketing,” remains accurate as of late 1996.
11. The late 1990s is an exciting time for young, energetic, computer-savvy, and market-savvy chemical engineering professionals.
12. Because of the impact of the Internet, the future of the traditional academic department in higher education as we knew it during the 1980s is questionable during the early decades of the new millennium.
13. Because of its exponentially increasing popularity, the Internet within the United States may develop frequent instances of “network gridlock” by 1997. Already, local Internet gridlock instances are being discussed using weather as a metaphor. Internet “caches” will become commonplace by the end of the millennium.
14. The historical examples of the early failures of CACHE Corporation projects – which were designed to stimulate the use of email – provide a useful lesson for the future. As Santayana stated, “Those who cannot remember the past are condemned to repeat it.”
15. The questions that were asked in 1994 concerning the potential impact of the Internet on chemical engineering education remain valid.
16. With improving price/performance ratios for both hardware and software, it is much more likely during the late 1990s that a chemical engineering student will have, during the undergraduate years, both (a) his/her own multimedia computer and (b) either a modem or Ethernet link to an Internet server. In other words, student access to the Internet will be direct during the late 1990s.
17. We started with the desktop PC during the late 1970s and early 1980s, and ended the millennium with the desktop PC, the laptop PC, and the NC. The author suggests that, as the technology wheel turns, the next innovation will become the KISSPC, namely, the keep-it-simple-stupid-PC. Users may ultimately revolt against the unstated expectation that a computer user must become a one-person computer center. Software has become too complex, bloated, and buggy.

References

American Astronomical Society (1989). A User's Guide to Electronic Mail (Part 1), *CACHE News*, **29**, 22-26.

American Astronomical Society (1990). A User's Guide to Electronic Mail (Part 2), *CACHE News*, **30**, 9-17.

Braun, Eric. *The Internet Directory*, Fawcett Columbine, NY, 1994.

CACHE Electronic Mail Task Force (1986). CACHE Computer Networking Student Contest, *CACHE News*, **23**, 3-5.

Cutlip, Michael D. GRAND Software: Visit to City University of New York Computer Center, *CACHE News*, **25**, 35 (Fall 1987).

Fraase, Michael. *The Windows Internet Tour Guide: Cruising the Internet the Easy Way*, Ventana Press, Chapel Hill, NC, 1994.

Gibbs, Mark, and Richard Smith, *Navigating the Internet*, SAMS Publishing, Carmel, IN, 1993.

Hampson, Martyn. *A Gopher Client for Windows 3.1, Version 2.4*, 1994.

Junod, John. *Windows Sockets FTP Client Application WS_FTP, Version 94.02.08*, February 8, 1994.

Kantor, Jeffrey C. *CACHE on the Internet*, Proposal to CACHE trustees, July 22, 1993.

Kim, Sangtae (1992a). Anonymous FTP Servers, *CACHE News*, **35**, 10-11.

Kim, Sangtae (1992b). Name Servers: or Electronic Mail Made Easy, *CACHE News*, **35**, 12.

Kriz, Harry M. Windows and TCP/IP for Internet Access, Release 3, February 9, 1994; electronic document available from hmkriz@vt.edu.

Krol, Ed. *The Whole Internet: User's Guide & Catalog*, O' Reilly & Associates, Inc. (103 Morris Street, Suite A, Sebastopol CA 95472), 1992.

NCSA, *A Beginner's Guide to HTML*, 1993.

Reible, D. D. (1987). CACHE Bulletin Board Service, *CACHE News*, **25**, 30-31. Note: This article was mistakenly attributed to Peter R. Rony.

Reible, Danny D (1988). LSU/CACHE Bulletin Board: General Information and User's Manual, *CACHE News*, **26**, 24-26.

Reible, Danny, and Peter Rony (1988). Status of GRAND Wide-Area Network File Server at Louisiana State University, *CACHE News*, **26**, 16-17.

Rony, Peter R (1985). The CACHE National Electronic Mail Experiment, CACHE Communications Task Force Proposal, April 9.

Rony, Peter R (1986). The CACHE National Electronic Mail Experiment: Part 2. Bitnet, Arpanet, and NSFNet, *CACHE News*, **22**, 10-12.

Rony, Peter R (1987a). Progress Toward a Wide-Area Network for Chemical Engineers, *CACHE News*, **25**, 36-38.

Rony, Peter R (1988). Standard List of Bitnet Userid Nicknames, *CACHE News*, **26**, 17-18.

Rony, Peter R (1989a). *Electronic Communications Task Force*, Memorandum to CACHE Executive Committee, March 6.

Rony, Peter R (1989b). Electronic Mail Task Force, *CACHE News*, **29**, 17-18.

Rony, Peter (1991). NSF Electronic Proposal Submission Project: A Report, *CACHE News*, **32**, 5-6.

Rony, Peter R., J.C. Hale, and J.D. Wright (1986a). The CACHE National Electronic Mail Experiment: Part 1. Compmail, *CACHE News*, **22**, 9-10.

Rony, Peter R., J.C. Hale, and J.D. Wright (1986b). Computer Networking for Scientists. *CACHE News*, **22**; comments on article from *Science* (February 26).

Rony, Peter, J. Wright, and N. Rawson (1987a). *Bitnet Wide-Area Network File Server for Chemical Engineering,* Memorandum sent to chemical engineering departments, June 20.

Rony, Peter, J. Wright, and N. Rawson (1987b). Bitnet User Identification Numbers, *CACHE News*, **25**, 28-29.

Tattam, Peter R (1994). *Trumpet Winsock, Version 1.0*, Trumpet Software International.

Tolhurst, William A., Mary Ann Pike, Keith A. Blanton, and John R. Harris (1994). *Using the Internet, Special Edition*, QUE Corporation, Indianapolis, IN.

Wintsch, Susan (1989). Toward a National Research and Education Network, National Science Foundation MOSAIC, 20 (4), 32-42.

Wilson, Chris, and Jon Mittelhauser (1994). *NCSA Mosaic for Microsoft Windows, Installation and Configuration Guide*.

Zollars, Richard L (1990). E-mail Discussion Groups: A List Owner's Perspective, *CACHE News*, **31**, 36-37.

INTELLIGENT SYSTEMS IN PROCESS OPERATIONS, DESIGN AND SAFETY

Steven R. McVey and James F. Davis
Ohio State University
Columbus, OH 43210

Venkat Venkatasubramanian
Purdue University
West Lafayette, IN 47907

Abstract

Many of the problems in process operations, design and safety are ill-structured and share certain generic characteristics that make the traditional approaches to automation very difficult if not impossible. The domain of intelligent systems offers a set of powerful concepts and techniques that are indispensable in addressing the challenges faced in process engineering. In particular, intelligent systems are poised to play a central role in the emerging paradigm of computer integrated manufacturing (CIM) applied to chemical process industries. Partial CIM implementations are quite prevalent and have resulted in the anticipated benefits of improved efficiency, better product quality, and lower operating costs.

In this paper, we trace the development of intelligent systems through the needs for their presence and their defining characteristics, examine a number of applications in process operations, design and safety, and articulate the value added.

Introduction

Over the recent decade artificial intelligence (AI) has emerged as an important problem-solving paradigm in process systems engineering. Intelligent systems is a branch of artificial intelligence with an interface to some applications area such as process engineering. While it is difficult to define or quantify precisely what intelligence is, whether it is natural or artificial, it is, however, useful to have some working definition of artificial intelligence and intelligent systems. In this sense, AI may be defined as the study of mental faculties through the use of computational models (Charniak and McDermott, 1985). Intelligent systems are defined in a similar vein as systems that exhibit the characteristics we associate with intelligence in human behavior - understanding language, learning, reasoning, solving problems, and so on (Barr and F.A. Feigenbaum, 1989). In recent practice, however, the term "intelligent systems" has come to be used to refer to a wide variety of artificial intelligence-based methodologies that include knowledge-based systems (a.k.a. expert systems), neural networks, genetic algorithms, fuzzy logic, machine learning, qualitative simulation, and others. In this paper, we review the origins, evolution and the applications of intelligent systems in the context of process systems engineering such as in process operations, process and product design and process safety.

The origins of intelligent systems research and development in process systems engineering can be traced to the late 1970's and early 1980's. A landmark is FOCAPD'83 (Conference on Foundations of Computer-Aided Process Design, held in 1983) where the keynote speaker proposed the application of expert systems in process operations. The CACHE/CAST sponsored conference sparked an interest that resulted in many systems-oriented researchers in academia initiating investigations. Early industrial implementations, while holding promise, foundered because of unmet performance expectations, difficult maintenance, high cost of specialized computers and software, lack of appropriate interfaces, and lack of integration. To reflect changes in perspective, the term "expert system" evolved to "knowledge-based system". This terminology attempted to capture a perspective that "expert systems" were not as much "expert" as they were viable and practical approaches that draw upon and encode expert knowledge and experience. The CACHE monograph series *Artificial Intelligence in Process Systems Engineering*, (Stephanopoulos and Davis, 1990-1992) provides details on knowledge-based system development through four monographs covering (1) an overview (Stephanopoulos, 1990), (2) rule-based systems (Davis and Gandikota, 1990), (3) knowledge representations (Ungar and Venkatasubramanian, 1990) and (4) object-oriented programming (Forsythe et al., 1992).

While the early research focus was on expert systems, recent breakthroughs in computer technologies brought neural networks into practical consideration with a resulting research shift toward this approach. While knowledge-based systems took advantage of expert knowledge and experience, neural networks provided the mechanisms for identifying the inherent correlations in process operating data. As was the case with knowledge-based systems, neural networks experienced an overshoot in expectations followed by a reconciliation of advantages and limitations. The combined experiences with these technologies have resulted in a much more rational stance that there is no single, uniform solution to modeling the behaviors of chemical processes. In fact, together these two technologies help harness information from both the process and the expert.

The current view recognizes that the artificial intelligence techniques while useful, must be considered as components of hybrid systems (Antsaklis and Passino, 1993; Rao et al., 1993). AI approaches such as knowledge-based systems, neural networks, genetic algorithms, etc., have to be integrated with more traditional techniques based on mathematical models, statistical tools and so on for a successful approach to computer-aided problem-solving in process systems engineering. The term "intelligent system" attempts to recognize this realization while also noting the inclusion of approaches that enable the integrated system to "automate tasks not covered by traditional numerical algorithms (Stephanopoulos and Han, 1994)."

Process Operations, Design and Safety

Many of the problems in process operations, process and product design and process safety are ill-structured and share certain generic characteristics that make the traditional approaches to automation very difficult if not impossible (Venkatasubramanian, 1994). These may be summarized as the following list of needs that one faces in many practical situations:

- *To reason with incomplete and/or uncertain information about the process or the product(s).* For example, in fault diagnosis, the sensor data may be uncertain due to noise and incomplete as all the relevant information may not be available. In hazard and operability analysis, one trys to identify abnormal situations based on qualitative information. These necessitate the use of qualitative modeling and reasoning.
- *To understand, and hence represent, process behavior at different levels of detail depending on the context.* For example, in process design it is important to be able to reason in a hierarchical manner. The understanding of the cause and effect interactions in a complex process system requires the generation of explanations hierarchically at different levels of detail.
- *To make assumptions about a process when modeling or describing it.* One then has to ensure the validity and consistency of these assumptions under dynamical conditions during the course of the problem-solving process. This also leads to adaptation when the underlying assumptions change. Again, this is a commonly found situation in design and model development.
- *To integrate different tasks and solution approaches.* This requires integrating different problem-solving paradigms, knowledge representation schemes, and search techniques. This is an important requirement for the development of an approach to integrate planning, scheduling, supervisory control, diagnosis and regulatory control.
- *To keep the role of the human in the decision-making process active and engaged.* This is crucial for the success of the system in many applications such as fault diagnosis.

Thus, it can be seen that the automation of process operations, design and safety require approaches that address such needs and demands. These are the challenges that intelligent systems can address. We will elaborate on this by describing five important characteristics of an intelligent system. These by no means represent an exhaustive characterization of system intelligence, but are intended to provide some measures of the extended value brought about by intelligent system approaches.

Qualitative Modeling

Inspired by the successes of CIM in discrete parts manufacturing, attempts have been made to generalize this concept to chemical process industries (Williams, 1989, Venkatasubramanian and Stanley, 1994, Macchietto, 1994). The application of CIM to the chemical process industries introduces challenges that do not exist for discrete parts manufacture (Madono and Umeda, 1994). These challenges result from complex chemical and physical processes and their interactions. Although mathematical models for these processes may exist, they are often inadequate for describing or predicting many phenomena of critical interest in the efficient operation of a plant. For example, complex chemical, kinetic and transport effects are often ill-understood quantitatively. As a result, mathematical models that approximate behaviors may be insufficient for activities such as control, planning, design or diagnosis. Other physical processes such as corrosion, fouling, mechanical wear, equipment failure and fatigue occur by poorly-understood mechanisms and are thus very difficult to model mathematically. Likewise, design selection decisions like sieve tray spacing, or unit configuration do not lend themselves to mathematical models and optimization.

Experienced plant operating personnel or experienced designers, on the other hand, are quite adept at making appropriate decisions regarding complex situations through qualitative considerations. These decisions are based on qualitative models derived from information gained directly from observed behaviors of a process operation (historical) and from experience with similar process components (design/predictive).

Intelligent system approaches provide the capability for encoding various forms of qualitative models, thereby extending modeling capacity beyond mathematical description. This qualitative modeling aspect is a primary distinction between intelligent systems and other, strictly mathematical approaches. The immediate value of an intelligent system (component) is apparent when the underlying model is describing (or predicting) useful plant phenomena and/or assimilating information into active decisions that cannot be modeled effectively using mathematical representations. The breadth of modeling is brought out by recent work on modeling languages such as DESIGN-KIT (Stephanopoulos et al., 1987), MODEL.LA (Stephanopoulos, 1990), and Functional Representation (Chandrasekaran, 1994; Goodaker and Davis, 1994).

Diagnosis is certainly an activity that lends itself to qualitative models since failure situations are very difficult to model mathematically. As examples, CATCRACKER (Ramesh et al., 1992) and CATDEX (Venkatasubramanian, 1988) are knowledge-based systems for diagnosing problems in fluid catalytic cracking units. The complexity of multiphase chemical reactions, fluid mechanics, thermal effects, and mass transport makes it nearly impossible to develop mathematical models of failure conditions that are of sufficient numerical resolution. However, qualitative models in the form of cause-and-effect relationships and that use qualitative interpretations of the process data are quite adequate in isolating failure conditions.

Similarly, pattern recognition neural networks make it possible to formulate black box models that map between process data and diagnostic conditions based on past occurrences (e.g., Leonard and Kramer, 1993; Davis and Wang, 1993; Vaidhyanathan and Venkatasubramanian, 1992). If there is sufficient operating data available, these models can be quite adequate for diagnosis when mathematical models are not. In process design, knowledge-based system approaches are very effective for modeling selection decisions, designing alternatives when constraints fail and managing the use of mathematical simulation. Qualitative models, based on experience, provide near optimized selections and effectively manage the complexity of the design problem.

Process hazards review and safety analysis, such as the HAZOP analysis, is another area where industrial practitioners have long used qualitative modeling and reasoning extensively to identify abnormal process situations. Recent work on this problem has demonstrated that the intelligent systems approach can be used successfully to automate HAZOP analysis of large-scale process P&IDs with significant gains in efficiency, accuracy and documentation (Venkatasubramanian and Vaidhyanathan, 1994). The success stems from the judicious use of qualitative modeling and object-oriented knowledge representation techniques.

The value of any intelligent system component first lies in its underlying model. It must be a model of some aspect of the process or decision making that cannot be modeled adequately using mathematics. Ill-conceived intelligent systems attempt to use qualitative models when mathematical models exist and are adequate or when there is not a good knowledge source to build an adequate model.

Cognition

An advantage of intelligent systems is that the techniques can be used not only to model process behaviors qualitatively but they can also simultaneously be used to model decision-making processes. Reasoning processes can take the form of simple mappings such as those found in neural networks or table look-up approaches or more complex processes that make use of specific knowledge organizations and inference strategies such as hierarchical classification, fault tree diagnosis, constraint-based reasoning, model-based reasoning, case-based reasoning, and so on.

Regardless of the approach, there is an added level of value associated with cognition. Cognition is the ability of an intelligent system to provide output that is at a symbol level, directly understandable to the user without further deliberation or interpretation. Cognition then is a measure of a system's ability to both explain its reasoning and communicate with the user in an understandable form.

Cognition can be broken down into two types, both of which may be present in an intelligent system. A cognitive system is one that: (i) provides conclusions in an interpreted, qualitative form that requires no further deliberation, and/or (ii) maintains this qualitative form throughout its manipulation and representation.

A system that gives results in an understandable form has, in a sense, performed the interpretation that otherwise would be left to the user. Cognition is therefore a characteristic of practical value in that results can be understood and used directly. In contrast, a numerical simulation that provides its results as a table of numbers or a graphics presentation leaves a great deal of analysis and interpretation to the person receiving the output.

An important practical implication of the second type of cognition is explanation capability. The reasons a system reaches a specific conclusion are readily understood when a qualitative representation is used. Although such systems may not be programmed to produce an explanation, the transparency of their qualitative representation facilitates understanding of their reasoning.

We can see the elements of cognition in any intelligent system component. For example, neural networks are well suited to process monitoring, diagnosis, sensor validation and data interpretation where pattern recognition plays a role. Generally, these systems classify data patterns into categories that are defined during a training phase (e.g., Tsen et al., 1994; Whiteley and Davis, 1994; Bakshi and Stephanopoulos, 1994; Kavuri and Venkatasubramanian, 1993 and 1994). QI-Map is a specific application example that illustrates the cognitive nature of neural networks (Whiteley and Davis, 1994; Davis and Wang, 1993). Implemented in ART2 (Carpenter and Grossberg, 1987), QI-Map has been used for data interpretation, monitoring and detection, and diagnosis. Trained using process data, the network maps from multisensor data input to appropriate labels that provide interpretations for immediate use. Training is supervised and the source of knowledge for the appropriate labels is expertise. For detection, the labels are "normal" and "not normal"; for data interpretation, there are many possible labels such as skewed, pulsing, cycling, tail, etc.; for diagnosis, the labels are specific root causes such as "plugged line," "contaminated feed," and so on. All of these labels are directly understandable without further deliberation. Neural net systems, however, are unable to explain their interpretations and would require additional symbolic-reasoning layers or modules to provide for that capability.

Similarly, the cognitive nature of fuzzy reasoning approaches derives from the assignment of qualitative, linguistic labels to numerical data based on membership functions (e.g., Roffel and Chin, 1991; Huang and Fan, 1993; Yamashita et al., 1988). Promising applications of fuzzy reasoning have been seen in the area of process control. Controller actions appear as the consequence of qualitative rules that are invoked by the presence of certain "fuzzy" conditions that are so labeled. As implied previously, the labels can be whatever is needed to be understandable and appropriate for taking action. Unlike neural networks, process data are not used directly to establish the membership functions. Rather they are constructed based on expertise with the process and the data. This additional emphasis on expert knowledge in the construction of the membership functions provides a capacity for explaining interpretations that neural networks do not have.

Knowledge-based systems also can exhibit one or both types of cognition. In table look-up, systems map directly from input data to conclusions by explicitly enumerating all input-output combinations (Saraiva and Stephanopoulos, 1994). The output is in an understandable form but there is no additional information inherent in the system to explain the conclusions. Alternatively, knowledge-based systems that are based on knowledge organizations to represent fragments of information and then search to piece information together provide a greater capability for explanation (Davis, 1994).

All intelligent system components offer cognition of the first type. Cognition together with qualitative modeling therefore define the basic value of any intelligent system. Cognition of the second type represents a higher degree of functionality, since explanation is possible. Systems exhibiting cognition of the second type involve a greater emphasis on expert knowledge.

Deliberation to Achieve Generalization

Deliberation to achieve generalization is associated with a significantly greater level of system performance. It refers to the ability of a system to make use of a representation and manipulation by some search mechanism (deliberation) to reach correct conclusions without having explicitly enumerated all the possible relationships between attribute combinations and conclusions (generalization). A very significant emphasis on various forms of expert knowledge is required, but the benefits are that the system can reach conclusions about situations that were not explicitly covered in the construction of the system and that may not have been previously considered.

As an illustration, consider a knowledge-based system whose representation consists of a hierarchy of nodes representing plant systems and subsystems. Each node contains evaluation information that results in the node being accepted or rejected based on the presence of some attributes. Acceptance of a given node results in the search continuing to its successors, each of which is constructed similarly. In constructing the representation, one defines each node separately with its own attribute of focus and then links it to other nodes. Provided the nodes are defined and linked together appropriately, the knowledge-based system will be able to deliberate on all combinations of attributes, even though the combinations are not explicitly enumerated. As the number of nodes and attributes increases, the capacity of representation and search becomes enormous.

In contrast, consider a spreadsheet package that provides a macro command, which executes a search for an input string in the left-hand column of a table and returns the string in the corresponding right-hand column. Usually called "table-lookup," this procedure produces the same results as a "deliberative" knowledge-based system, if the pre-enumeration is complete. The drawback is that all possible attribute combinations must be entered into the table along with the corresponding conclusions. Even simple problem domains that require a relatively few number of attributes and conclusions are not amenable to solution via table-lookup.

Good examples of applications that benefit from deliberation and generalization are fault diagnostic knowledge-based systems such as FALCON (Rowan, 1988, 1992), PX (Prasad et al., 1993), Shell KB (Kumar and Davis, 1994), CATCRACKER (Ramesh et al., 1992), MODEX2 (Venkatasubramanian and Rich, 1988), and Faultfinder (Hunt et al., 1993). All of these systems emphasize knowledge bases, various representations and corresponding inference mechanisms that make it possible to accommodate and react to a very wide range of possible operating situations without having to pre-enumerate every combination. These representations are usually some form of directed graph in which nodes contain separate pieces of qualitative knowledge about process faults. Different inference mechanisms are used to traverse the representations, examine the nodes and draw conclusions.

FALCON is a system that relies on a representation that combines qualitative knowledge about the process with mathematical simulations. Generalization occurs through selective simulation of various normal and failure conditions coupled with appropriate interpretation. PX, Shell KB, and CATCRACKER are all systems based on hierarchical classification. Hierarchical classification leads to generalization by decomposing the process into a hierarchical set of subcategories, i.e., subsystems and/or fault categories and then pre-enumerating the evaluation patterns for each. Generalization occurs as a result of the interactions between the subcategories as the diagnosis proceeds. In addition to its establish-refine mechanism, the Shell KB system utilizes a technique called hypothesis queuing (HQ), which prioritizes malfunction hypotheses based on their likelihood in a given situation and directs the search accordingly. HQ extends the existing deliberation mechanism and thus enhances the performance of the system.

Faultfinder, MODEX2, and HAZOP (Catino et al., 1991) are examples of systems that achieve generalization by linking device models together according to the topology of the process. By propagating effects forward and backward from unit model to unit model, the system is able to generate process behaviors without pre-enumerating all possible symptom patterns.

An intelligent system designed to achieve generalization represents a significant increase in the value of the system. This level of system functionality builds upon the elements of modeling and cognition to provide the added capacity of reaching correct conclusions about situations that have not been considered previously. Various intelligent systems for diagnosis, hazard identification, design and planning have demonstrated this capability.

Adaptation

Adaptation (synonymous with learning) refers to the ability of a system to make changes to its representation in response to new information through deliberation. This definition allows us to differentiate between a deliberative adaptive system and say, a model identification problem, which is merely a parameterized model that automatically adapts to new process data. For example, systems such as neural adaptive, model-predictive, and dynamic matrix controllers update model parameters or network weights as process data are received so that the controllers use the latest model of the process. Although these systems do, in a sense, adapt by changing parameter values, no deliberation, no cognition and no qualitative modeling are involved. As a result, these systems, while important components of an integrated intelligent system are not considered "intelligent system" components in themselves.

Unlike the identification techniques mentioned above, systems that deliberatively adapt by updating their own knowledge representations are scarce. There is current work on adaptation of process monitoring systems (e.g., Davis et al., 1994; Bakshi and Stephanopoulos, 1994). It is important for a monitoring system to incorporate new patterns in its representation as they occur. At present, these systems are able to partially automate the adaptation process but elements of supervision are still required. There are only a few examples of work on learning of knowledge-based systems in process engineering (e.g., Modi et al., 1993, Rich and Venkatasubramanian, 1989).

Autonomy

Autonomy refers to the ability of a system to achieve a fixed high-level goal through appropriate accomplishment and/or revision of subgoals without needing outside intervention. It represents an ultimate capability for an intelligent system. The ability of a system to perform its task autonomously depends greatly on how well it copes with uncertain information and changing circumstances. Autonomy is generally considered to have a greater role in large, distributed systems containing several component modules that must communicate with each other effectively in order for the system to attain its goal (Antsaklis and Passino, 1993). The absence of human intervention places great demands on integrated intelligent systems, which must draw upon all the previously mentioned characteristics in order to effect autonomy. In particular, the notion of cognition must be extended to encompass the communication among subsystems. In addition to producing results that are understandable to humans, these systems also manage information that is passed from one module to another. The inputs and outputs must at some point appear in forms that are meaningful to other modules, intelligent or otherwise.

Systems pointed toward autonomy are at the early stages of development. Redmill et al. (1994) have considered autonomous control of rocket engines in unmanned space vehicles. Marchio and Davis (1994) have prototyped a procedure management system for plant start-ups and cyclical operations. Bhatnagar et al. (1990) have developed a system for failure action management in nuclear power plants. Each of these systems exhibit aspects of autonomy through sophisticated forms of deliberation. Full implementation of CIM systems today is confounded by difficulties in integrating lower-level supervisory and control execution activities with the higher-level planning and scheduling functions. As a consequence, conventional agents, i.e., humans, are currently used to fill the “gap.” Autonomous systems, by definition, cannot rely on this intervention. Adaptation is of major importance to maintaining autonomy, as is generalization. These must exhibit a high degree of development in order to successfully automate human problem solving and decision making tasks. Clearly, the achievement of autonomy is an extremely challenging objective that will become increasingly relevant as work towards integrating intelligent systems continues.

Conclusions

In this paper we have attempted to define the value added by intelligent systems in terms of five characteristics: qualitative modeling, cognition, deliberation to achieve generalization, adaptation, and autonomy. These incorporate major aspects of “intelligent” functionality observed in the different techniques and applications that have appeared thus far. Qualitative modeling represents an essential component of cognition and generalization and underlies all aspects of system intelligence. The benefits of these characteristics culminate in autonomy, which embodies all of them to some degree. As each of these is “built into” a system, the improved performance brought by the system to the particular application increases.

Current research is generally focused on development of the qualitative, generalization, and cognitive aspects of intelligent applications. In the past few years, however, research efforts have also begun to emphasize integration and, in particular, the challenges presented by integrated intelligent systems. Adaptation techniques are in the very early stages of development. Autonomy of large integrated intelligent systems that can effectively manage complex interactions of disparate information and techniques remains a goal that has yet to be realized.

Acknowledgments

We would like to thank Dr. B. Chandrasekaran (Computer and Information Science) and Dr. U. Ozguner (Electrical Engineering) for their helpful participation in our discussions on intelligent systems.

References

Antsaklis, P. and K. Passino (1993). Introduction to intelligent control systems with high degrees of autonomy, in *An Introduction to Intelligent and Autonomous Control*, P. Antsaklis and K. Passino (eds.), Kluwer Academic Publishers, Boston.

Barr, A. and F.A.Feigenbaum (1989). *The Handbook of Artificial Intelligence, Vol. 1*, William Kaufman Pub.

Bhatnagar, R., D. Miller, B. Hajek, and J. Stasenko (1990). An integrated operator advisor system for plant monitoring, procedure management, and diagnosis. *Nuclear Tech.*, **89**, 281-317.

Bakshi, B. and G. Stephanopoulos (1994). Representation of process trends - III. Multiscale extraction of trends from process data. *Computers Chem. Engng.*, **18**, 4, 267-302.

Bakshi, B. and G. Stephanopoulos (1994). Representation of process trends - IV. Induction of real-time patterns from operating data for diagnosis and supervisory control. *Computers Chem. Engng.*, **18**, 4, 303-332.

Calandranis, J., G. Stephanopoulos and S. Nunokawa (1990). DiAD-Kit/Boiler: on-line performance monitoring and diagnosis. *Chem. Engng. Prog*., **86**, 1, 60-68.

Carpenter, G. and S. Grossberg (1987). ART2: self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, **26**, 4919-4930.

Catino, C., S. Grantham and L. Ungar (1991). Automatic generation of qualitative models of chemical process units. *Computers Chem. Engng.*, **15**, 8, 583-599.

Chandrasekaran, B. (1994). Functional representation and causal processes, in *Advances in Computers*, 38, Academic Press, 73-143.

Charniak and McDermott (1985). *Introduction to Artificial Intelligence*. Addison-Wesley Pub.

Cott, B. and S. Macchietto (1989). An integrated approach to computer-aided operation of batch chemical plants. *Computers Chem. Engng*., **13**, 11/12, 1263-1271.

Crooks, C. A., K. Kuriyan and S. Macchietto (1992). Integration of batch plant design, automation, and operation software tools. *Computers Chem. Engng.*, **16S**, S289-S296.

Davis, J. (1994). On-line knowledge -based systems in process operations: the critical importance of structure on integration. *Proceedings IFAC Symposium on Advanced Control of Chemical Processes, ADCHEM'94*, Kyoto, Japan.

Davis, J. and M. Gandikota (1990). Rule-based systems in chemical engineering, *CACHE Monograph Series, AI, in Process Systems Engineering. vol. II*, G. Stephanopoulos and J. Davis (eds.) CACHE.

Davis, J. and C. Wang (1993). Adaptive resonance theory for on-line detection and sensor validation: A practical view. *AIChE Annual Meeting*, St. Louis MO, November 1993.

Davis, J., C. Wang and J. Whiteley (1994). Real-time knowledge-based interpretation and adaptation of process data. *Proceedings Fifth International Symposium on Process Systems Engineering*, Seoul, Korea, 1153-1159.

Forsythe, R., S. Prickett, and M. Mavrovouniotis (1992). An introduction to object-oriented programming in process engineering, *CACHE Monograph Series, AI in Process Systems Engineering. vol. IV*, G. Stephanopoulos and J. Davis (eds.) CACHE.

Goodaker, A. and J. Davis (1994). Sharable engineering knowledge databases in support of HAZOP analysis of process plants. *AIChE Annual Meeting*, San Francisco, November 1994.

Huang, Y. and L. Fan (1993). A fuzzy-logic-based approach to building efficient fuzzy rule-based expert systems. *Computers Chem. Engng*., **17**, 2, 181-192.

Hunt, A., B. Kelly, J. Mullhi, F. Lees, and A. Rushton (1993). The propagation of faults in process plants: 6, overview of and modeling for, fault tree synthesis. *Reliability Engng. and Proc. Safety*, **39**, 2, 173-194.

Hunt, A., B. Kelly, J. Mullhi, F. Lees, and A. Rushton (1993). The propagation of faults in process plants: 7, divider and header units in fault tree synthesis. *Reliability Engng. and Proc. Safety*, **39**, 2, 195-209.

Hunt, A., B. Kelly, J. Mullhi, F. Lees, and A. Rushton (1993). The propagation of faults in process plants: 8, control systems in fault tree synthesis. *Reliability Engng. and Proc. Safety*, **39**, 2, 211-227.

Hunt, A., B. Kelly, J. Mullhi, F. Lees, and A. Rushton (1993). The propagation of faults in process plants: 9, trip systems in fault tree synthesis. *Reliability Engng. and Proc. Safety*, **39**, 2, 229-241.

Hunt, A., B. Kelly, J. Mullhi, F. Lees, and A. Rushton (1993). The propagation of faults in process plants: 10, fault tree synthesis - 2. *Reliability Engng. and Proc. Safety*, **39**, 2, 243-250.

Kramer, M. and R. Mah (1994). Model-based monitoring. *Proceedings of the Second International Conf. on FOCAPO*, Crested Butte CO, CACHE/Elsevier, 45-68.

Kumar, S. and J. Davis (1994). Knowledge-based problem solving in after-cycle, root cause diagnosis of batch operations. submitted to *Industrial and Chemistry Research.*

Kavuri, S. and V. Venkatasubramanian (1993). Using fuzzy clustering with ellipsoidal units in neural networks for robust fault classification. *Computers Chem. Engng.*, **17**, 8, 765-784.

Kavuri, S. and V. Venkatasubramanian (1994). Neural Network Decomposition Strategies for Large Scale Fault Diagnosis.*Special Issue of Intl. J. Control*, (Ed.) Morari and Morris, **59**, 3, 767-792.

Leonard, J. and M. Kramer (1993). Diagnosing dynamic faults using modular neural nets. *IEEE Expert*, **8**, 2, 44-53.

Macchietto, S. (1994). Bridging the gap - integration of design, scheduling, and process control. *Proceedings of the Second International Conf. on FOCAPO*, Crested Butte, CO, CACHE/Elsevier, 207-230.

Madono, H. and T. Umeda (1994). Outlook of process CIM in Japanese chemical industries. *Proceedings of the Second International Conf. on FOCAPO*, Crested Butte, CO, CACHE/Elsevier, 163-177.

Marchio, J. and J. Davis (1994). A generic knowledge-based architecture for batch process diagnostic advisory systems. paper presented at *AIChE National Spring Meeting*, 1994.

Modi, A., A. Newell, D. Steier, and A. Westerberg (1993). Building a chemical process design system within Soar: part 2 - learning issues. to be published in *Computers Chem. Engng*.

Prasad, P. and J. Davis (1993). A framework for knowledge-based diagnosis in process operations, chapter in *An Introduction to Intelligent and Autonomous Control*, P. Antsaklis and K. Passino (eds.), Kluwer Academic Publishers, Boston.

Prasad, P., J. Davis, H. Gehrhardt and B. Feay (1993). Generic nature of task-based framework for plant-wide diagnosis. in revision *Computers Chem. Engng*.

Prasad, P., Y. Jirapinyo and J. Davis (1993). Fault trees for knowledge acquisition and hierarchical classification for diagnosis. *IEEE Expert*, July 1993.

Psichogios, D. and L. Ungar (1992). A hybrid neural network - first principles approach to process modeling. *AIChE Journal*, **38**, 10, 1499-1511.

Ramesh, T., J. Davis and G. Schwenzer (1992). Knowledge-based diagnostic systems for continuous process operations based on the task framework. *Computers Chem. Engng.*, **16**, 2, 109-127.

Rao, M., Q. Wang and J. Cha (1993). *Integrated Distributed Intelligent Systems in Manufacturing*. Chapman & Hall, London.

Redmill, K., U. Ozguner, J. Musgrave, W. Merrill (1994). Design of intelligent hierarchical controllers for a space shuttle vehicle. *Proceeding 1993 IEEE International Symposium on Intelligent Control*, 64-69.

Rich, S. H. and V. Venkatasubramanian (1989). Causality-based Failure driven Learning in Diagnostic Expert Systems, *AIChE Journal*, **35,** 6, 943-950.

Roffel, B. and P. Chin (1991). Fuzzy control of a polymerization reactor. *Hydrocarbon Processing.*, **70**, 6, 47-49.

Rowan, D. (1988). AI enhances on-line fault diagnosis. *InTech*, **35**, 5, 52-55.

Rowan, D. (1992). Beyond Falcon: Industrial applications of knowledge-based systems. *Proceedings IFAC On-line Fault Detection and Supervision in the Chemical Process Industries*, 215-217.

Saraiva, P. and G. Stephanopoulos (1994). Data-driven learning frameworks for continuous process improvement. *Proceedings Fifth International Symposium on Process Systems Engineering*, Seoul, Korea, 1275-1281.

Stephanopoulos, G. (1990). Brief overview of AI and its role in process systems engineering, *CACHE Monograph Series, AI in Process Systems Engineering. vol. I*, G. Stephanopoulos and J. Davis (eds.) CACHE.

Stephanopoulos, G. and C. Han (1994). Intelligent systems in process engineering: a review. *Proceedings Fifth International Symposium on Process Systems Engineering*, Seoul, Korea, 1339-1366.

Stephanopoulos, G., G. Henning, and H. Leone (1990). MODEL.LA. A modeling language for process engineering - I. The formal framework. *Computers Chem. Engng.*, **14**, 8, 813-846.

Stephanopoulos, G., G. Henning, and H. Leone (1990). MODEL.LA. A modeling language for process engineering - II. Multifaceted modeling of processing steps. *Computers Chem. Engng.*, **14**, 8, 847-869.

Stephanopoulos, G., J. Johnston, T. Kriticos, R. Lakshmanan, M. Mavrovouniotis, and C. Siletti (1987). DESIGN-KIT: an object-oriented environment for process engineering. *Computers Chem. Engng.*, **11**, 6, 655-674.

Tsen, A., S. Jang, D. Wong and B. Joseph (1994). Predictive control of quality in batch polymerization using artificial neural network models. *Proceedings Fifth International Symposium on Process Systems Engineering*, Seoul, Korea, 899-911.

Ungar, L. and V. Venkatasubramanian (1990). Knowledge representation, *CACHE Monograph Series, AI in Process Systems Engineering. vol. III*, G. Stephanopoulos and J. Davis (eds.) CACHE.

Vaidhyanathan, R. and V. Venkatasubramanian (1992). Representing and diagnosing dynamic process data using neural networks. *Engng. Appl. of Artif. Intell.*, **5**, 11-21.

Venkatasubramanian, V. (1988). CATDEX: An expert system for diagnosis of a catalytic cracking operation. *CACHE Case-Studies Series*, Knowledge-Based Systems in Process Engineering, G. Stephanopoulos (ed.), vol. II, CACHE.

Venkatasubramanian, V. (1994). Towards Integrated Process Supervision: Current Trends and Future Directions, Keynote Address, *Second IFAC International Conference on AI/KBS in Process Control,* University of Lund, Lund, Sweden.

Venkatasubramanian, V. and S. Rich (1988). An object-oriented two-tier architecture for integrating compiled and deep-level knowledge for process diagnosis. *Computers Chem. Engng.*, **12**, 9, 903-921.

Venkatasubramanian, V. and G. Stanley (1994). Integration of process monitoring, diagnosis and control: issues and emerging trends. *Proceedings of the Second International Conf. on FOCAPO*, Crested Butte CO, CACHE/Elsevier, 179-206.

Venkatasubramanian, V. and R. Vaidhyanathan (1994). A Knowledge-Based Framework for Automating HAZOP Analysis, *AIChE Journal*, **40**, 3, 496-505.

Whiteley, J. and J. Davis (1994). A similarity-based approach to interpretation of sensor data using adaptive resonance theory. *Computers Chem. Engng.*, **18**, 7, 637-661.

Williams, T. (ed.) (1989). *A Reference Model for Computer Integrated Manufacturing (CIM)*. Instrument Society of America, Research Triangle Park NC.

Yamashita, Y., S. Matsumoto and M. Suzuki (1988). Start-up of a catalytic reactor by fuzzy controller. *Journal of Chem. Engng. of Japan*, **21**, 3, 277-282.

LANGUAGES AND PROGRAMMING PARADIGMS

George Stephanopoulos and Chonghun Han
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Are programming and computer science relevant to the education and professional needs of the vast majority of chemical engineers? In this chapter we explore this question in the presence of shifting paradigms in programming languages and software design. After a brief history in the evolution of programming languages, high-level, chemical engineering-oriented languages are discussed as the emerging programming paradigm, and certain conclusions are drawn on the importance of computer science as a pivotal element in engineering education and practice.

Introduction

The explosive growth in the use of the digital computer is a natural phase in the saga of the Second Industrial Revolution. If the First Industrial Revolution in 18th century England ushered the world into an era characterized by machines that extended, multiplied and leveraged the humans' *physical capabilities*, the Second, currently in progress, is based on machines that extend, multiple and leverage the humans' *mental abilities.* The thinking man, homo sapiens, has returned to its Platonic roots where, "*all virtue is one thing, knowledge*." Using the power of modern computer science and technology, software systems are continuously developed to; preserve knowledge for it is perishable, clone it for it is scarce, make it precise for it is often vague, centralize it for it is dispersed, and make it portable for it is difficult to distribute. The implications are staggering and have already manifested themselves, reaching the most remote corners of the earth and inner sancta of our private lives.

Central to this development has been the story of the continuously evolving programming languages and programming paradigms. From FORTRAN 0 (in 1954) to FORTRAN 90, from ALGOL 58 (in 1958) to BASIC and PL/1, and through ALGOL 68, to Pascal and C, from ALGOL 60 through SIMULA I to Smalltalk 80 and C++, from LISP through ZetaLisp to Common LISP and CLOS, the programming languages have been evolving like the natural languages of humans; to fit the evolving needs in representing the world and articulating ideas. From the 0, 1 alphabet to modern modeling languages, computer scientists and engineers continue to enrich and formalize the way humans communicate with a computer. There is no end to this process, as there is no end to the evolution of the human languages.

The expressive power of a programming language influences the breadth and depth of computer-aided human solutions and their communication to others, very much as the progress of human civilization moved in tandem with the growing expressiveness and precision of human languages. The following corollary is a little discussed understatement, and, for some, a trivial redundancy: *limited grasp of programming languages limits the sophistication of computer-aided engineering solutions.* If this is true, then the theoretical foundations of a computer language are as central to engineering education as physics, mathematics, and chemistry.

As the growing pervasiveness of computers in present-day life demands from humans increased familiarity and higher-level communication skills with machines, the above corollary becomes all too important. It suggests that only natural language-like communication will allow humans to fully exploit the capabilities of a digital computer; *not in executing prepackaged programs, but in becoming an integral part of and extending the human ability.* A student of chemical engineering should not be forced to express his/her ideas in 0 and 1's, FORTRAN statements, or any other linguistic straight jacket. Instead, he/she should be able to communicate naturally with machines, using the language of chemical engineering science and practice. Therefore, the natural form of computer programming for a chemical engineer is worlds apart from the current paradigm, which is full of intellectually stifling coding and, at times unbearably, painful debugging.

The evolution of programming languages has been followed by a shift in the programming paradigm, i.e., the process of designing and developing software systems. From the unstructured and convoluted spaghetti-like designs of earlier times (which can still be seen today in commercially successful software products), to modular and structured-programming products, to easily reusable and maintainable object-oriented codes, the process of designing a software system has been largely streamlined and rationalized. Today, the development of a complex software system is not in the realm of a creative "hacker," but instead, is the result of teamwork, and resembles more the production line of a modern manufacturing plant, whose units are not constrained by geographic location.

In the midst of all these developments, what is the role that programming languages and programming paradigms can play in the education of a chemical engineer? How do they affect the way that a professional chemical engineer carries out his/her work; product/process research, development or engineering? In the following sections of this chapter we will try to discuss the above issues. Specifically, the next section will provide a brief overview of the historical evolution of programming languages, and will attempt to underline the foundations which may constitute an educational component in undergraduate engineering curricula. The third section will focus on "modeling languages," a natural evolution of programming languages that seems to serve better present and anticipated future needs for chemical and other types of engineers. The fourth section introduces the task-oriented languages which complement the modeling languages to form the paradigm of future programming. The shift in software design paradigms will be discussed in the fifth section. Section six will attempt to draw some conclusions, which may have an effect on the computer-aided education and practice of chemical engineers.

Programming Languages

The history of the evolution of programming languages is a fascinating subject that transgresses the boundaries of interests of computer scientists, and offers useful insights on the interplay of intellectual contributions, commercial interests, and human inertia. The encyclopedia edited by Wexelblat (1981) offers a rich panorama of this history. The book by Sebesta (1993) provides the reader with a shorter version of this history, and an educational exposition to programming languages. For a formal exposition to the theory of programming languages, the book by Meyer (1990) is an exceptional source. The first digital computers were constructed in the 1940s in response to the needs of scientific applications, which involved the solution of large and complex numerical problems. *PlanKalcul* was the first high-level programming language. A fairly rich language, it was composed by the German scientist Konrad Zuse in 1945, but was not published until 1972 (Zuse, 1972). Although Zuse wrote many programs, using PlanKalkul, and executed them on Z4, a computer he constructed with electromechanical relays, his work did not have any impact on the subsequent development of programming languages, since it was not widely known for a long time.

FORTRAN - The Paradigm of Imperative Programming

The decisive step forward was the development of FORTRAN (FORmula TRANslator), in response to the new capabilities offered by the IBM 704 computer. The road to the formulation of FORTRAN had been paved by a number of earlier and parallel research efforts, such as: John Mauchly's interpreted *ShortCode* (Wexelblat, 1981), developed in 1949 and run on BINAC and UNIVAC I computers; John Backus' *Speedcoding* interpreter with the ability to process floating-point data; Grace Hopper's development of advanced compiling systems for the UNIVAC computers; Alick Glennie's *Autocode* compiler for the Manchester Mark I computer.

The development of FORTRAN 0 began in 1954 (IBM, 1954) and was implemented the following year. Subsequent modifications led to FORTRAN I, whose compiler was released in 1957. FORTRAN II was introduced the following year, it was succeeded by FORTRAN III in 1959, and reached maturity with FORTRAN IV in 1962. FORTRAN IV, one of the most widely used programming languages, became the standard for an explosive growth in scientific computing during the period 1962 to 1978, when it was succeeded by FORTRAN 77, and which in turn was followed by FORTRAN 90 (ANSI, 1990). The transformation of FORTRAN over a period of 25 years is remarkable. Becoming the standard language for scientific computing, it was continuously under pressure to adapt and include new features, which were required by the expanded needs of its users. Other programming languages, developed during the period 1960-1990, had many inherent design advantages over FORTRAN, but none could replace it, since FORTRAN kept adapting to include the most attractive of the missing features.

FORTRAN is also important from another point of view. It represents the prototypical paradigm of *Imperative Programming Languages*. The imperative style of programming is characterized by the presence of constructs describing commands issued to a machine (Meyer, 1990). The construct that makes a programming language most prominently imperative is the *instruction*. Thus, programs written in an imperative language are sequences of instructions, with each instruction describing a set of actions to be performed. Each instruction changes the state of the program, by modifying the values of variables and defining the sequence of subsequent instructions to be executed.

The Gap Between Mathematics and Imperative Languages

Mathematical notation is *referentially transparent* (Meyer, 1990). For example, the relationships, x = 1 and x + z < 10, imply that, 1 + z < 10. Although it is made explicit, referential transparency is extremely important in mathematical reasoning, since all FORmula manipulations (arithmetic and logical) rely on it. Imperative programming languages, like FORTRAN, on the other hand, violate referential transparency, because they permit side effects on the variables of a program. For example, consider the following sequence of FORTRAN instructions, defining a function, F(x): Instruction-i; y=y+1 Instruction-(i+1); F=y+x Instruction-(i+2); End. Although called function, F(x) is not a function in the mathematical sense. It does not produce a result computed from its arguments, but changes the state of the program, by modifying the value of the variable y every time that the function is called. As a consequence, referential transparency among different expressions is lost. *Aliasing*, the access of a given object through more than one name, is another feature of the imperative programming languages that generates a gap between mathematics and the encoded computer programs.

ALGOL and Its Offsprings - More Imperative Programming Languages

The Association for Computing Machinery (ACM) in the U.S. and the Society for Applied Mathematics and Mechanics (GAMM) in Germany, decided, in 1958, to join efforts and produce a universal, machine-independent programming language, which would be as close as possible to “standard mathematical notation.” Thus, ALGOL 58 (ALGOrithmic Language) came into existence, formalizing the concept of data type, adding compound statements, expanding the number of array dimensions, allowing nested conditional statements, and a number of other highly desirable features. In 1960, a six-day meeting in Paris produced ALGOL 60, which was the first language to be described in detailed form, using the Backus-Naur syntax (Naur, 1960) that is still widely used. ALGOL 60 became one of the most pivotal contributions in the evolution of programming languages, introducing the concept of “Block structure,” implementing recursion, and allowing the passing of parameters to subprograms by value or by name. Its direct descendant, ALGOL 68, introduced the revolutionary innovations of “user-defined data types,” and “dynamic arrays,” and thus led to the design of two new offsprings with significant influence; Pascal, and C. Although Pascal and C did not introduce any significant innovations, they both became far more popular than ALGOL 68. Pascal (Wirth, 1971) became the language of choice for the teaching of programming, while C evolved into the language of choice for the professional systems programmers. C became the first truly portable language, and in addition to ALGOL 68, it owes a lot of its character to languages such as, CPL, BCPL, and B. Ada is also a descendant of ALGOL 68, being based on Pascal.

Data Abstraction - From SIMULA to Smalltalk and C++

SIMULA I is another language that draws its ancestry from ALGOL 60. It was designed and implemented in 1964 to support system simulation applications. Its successor, SIMULA 67 (Dahl and Nygaard, 1967) introduced one of the most pivotal innovations, the *class* construct, which became the basis for data abstraction and the subsequent maturation of object-oriented programming ideas. A “class” was a construct, which packaged into one entity both (i) the data structure that defined the attributes of the class, and (ii) the routines that manipulated the data structure.

The “class” concept of SIMULA 67 became the pivotal element for the development of *Dynabook* (Kay, 1969), a computer program that used the idea of a “desk-top” on a graphic screen, to simulate the working environment of future computer users. Thus, the concept of “windows,” currently visible in every personal computer, was born. Moving to Xerox, Kay led a group effort, whose objective was to create a language that would support the highly iconic and graphic paradigm of the man-machine interaction, and build a computer which would deliver it. Smalltalk-72, -74, -76, -78 represent the early versions of a revolutionary programming language, which took on its matured form in Smalltalk-80.

All program units in Smalltalk are *objects*, and can be viewed as instances of abstract data types, called *classes*. The Smalltalk classes are very similar to those of SIMULA 67. Each class contains a set of attributes, which define the data structure of the class, and a series of procedures, called *methods*, which operate on the attributes. During the execution of a Smalltalk program the objects “call and ask” other objects to carry out specific tasks. The calling object does not need to know “how” the receiver of its message will carry out the task, thus leading to complete encapsulation. Smalltalk introduced two innovative ideas, *inheritance* and *dynamic typing*, which, along with the concept of class, formed the basis for what it has come to be known as, *object-oriented programming* (OOP). The notion of inheritance allows the straightforward extension of existing classes to new ones with specialized features. Thus, a set of existing classes can be continuously extended with new classes by making full utilization of existing libraries in a very efficient manner. This reusability and extensibility is at the core of vast improvements in software development efficiency, which has been realized in recent years.

Although object-oriented programming has become and will remain for the next decade the dominant paradigm, Smalltalk is not the language of choice. Instead, C++, an evolutionary object-oriented version of C seems to become the standard. The first steps towards C++ were taken by Bjarne Stroustrup at Bell Laboratories in 1980 (Stroustrup, 1983). He introduced a new data type, called “class,” which captured the essential character of classes as it was designed by SIMULA67 and Smalltalk. Additional modifications of C brought in inheritance, but not dynamic typing. Between 1986 and 1994, C++ continued to evolve and today, commercial C++ versions includes; multiple inheritance, and a growing number of product-specific classes which encapsulate elements of graphic interfaces (e.g. windows, menus, buttons, tables, lists, icons, spreadsheets, graphs, etc.), procedures for solving numerical problems, interfaces to databases, and many others. The reusability and extensibility of C++ has made it the replacement of C for software engineering. The growth rate on the number of C++ classes within specific programming environments has been phenomenal. Small companies appear everyday, marketing C++ classes for specific market needs.

C++ is a successful hybrid of imperative and object-oriented programming. Through evolution has moved the programming to the next phase, i.e. the OOP. It seems that evolution is a safer road to commercial success than revolution, a feature attempted by Smalltalk.

Borland’s Turbo Pascal is another example of an imperative language’s evolution to include object-oriented ideas. How far behind is Object-FORTRAN?

The universal acceptance of the OOP paradigm led to a fury of commercial activity during the last decade, which led to a number of OOP languages, like; CLOS, ACTOR, Eiffel, Objective-C and others. The unavoidable standardization has recently produced a commercial streamlining, with C++ taking the dominant position and Smalltalk proving its endurance through a smaller circle of programming artists.

LISP - The Paradigm of Functional Programming

Imperative languages are "polluted by side effects, assignments, explicit sequencing, and other repugnant features" (Meyer, 1990), all of which have led to a serious gap between the language of mathematics and that of the corresponding imperative examples, e.g. FORTRAN, Pascal, and C. LISP is an example of a programming language which proves that all the above weaknesses are not necessary evils. Other examples are FP and MIRANDA, data flow languages, such as LUCID, or logic programming languages, such as PROLOG.

LISP (LISt Processing) was developed by John McCarthy in 1958 and was aimed at supporting the computational needs of the new domain called Artificial Intelligence. It is the first *functional programming* language, and as such mimics quite closely the language of mathematics. A purely functional programming language does not use variables or assignment statements, like an imperative language. Instead, programs written in a functional language are function definitions and function application specifications. Execution of such programs can be viewed as evaluation of function applications.

LISP, like a typical functional programming language, provides a set of primitive functions and a set of functional forms, which are used to construct more complex functions from the primitive ones. In addition, it possesses a function application operation, eval, and two specific structures for storing data, namely, atoms and lists. Its attractive functional programming character and its subsequent standardization in *Common Lisp*, have made LISP one of the most enduring languages, although its popularity is far from that of the dominant imperative languages. Its recent evolution to CLOS (Common List Object System) has integrated functional with object-oriented programming.

PROLOG - The Paradigm of Declarative Programming Languages

PROLOG (PROgramming LOGic) is a nonprocedural language. Each statement in a PROLOG program is a *declaration* of a logical relation, *proposition*, among variables, functions, or/and parameters. Predicate calculus is the notation used for the expression of logical statements, and provides the *resolution*, which is the inference technique during the execution of a PROLOG program. A product of joint work of the Artificial Intelligence groups at the Universities of Edinburgh and Aix-Marseille, PROLOG was conceived to support the logical inferencing required in AI applications (Roussel, 1975). It is the pre-eminent example of a purely *declarative programming* language, and allows a straightforward juxtaposition to imperative and functional languages.

Other Milestones in the Evolution of Programming Languages

Languages like, BASIC, PL/I, COBOL and others, have played an important role in the saga of programming languages and the evolution in the use of computers. BASIC (Mather and

Waite, 1971), for example, was designed in the early 1960s to be simple and easy to be used by non-computer science oriented students from remote terminals in a *time-shared* environment. Shunned over 20 years by programmers, it has recently been revived, as *Visual Basic* with tremendous popularity, thanks to the marketing genius of Microsoft. COBOL is a very interesting example of a deliberate design effort. Designed by a committee over a short period of time to address the computing needs of business applications (Department of Defense, 1960), COBOL became an ANSI standard in 1968, and today is still one of the most widely used languages. It introduced a number of innovations, which subsequently were adapted by other languages, e.g. hierarchical data structures.

PL/I was the product of an effort to produce a language which would serve the needs of a broad spectrum of applications. Known initially as NPL (New Programming Language), PL/I included the best features of FORTRAN IV, COBOL, and ALGOL60. It introduced facilities to handle a large number of run-time errors, create concurrently executing tasks, carry out recursion, etc. PL/I has been a convincing example of the assertion that complexity and redundancy in the formulation of a language are serious drawbacks. As a result, most of the present-day programming languages are based on a logically firm foundation, which serves efficiently a set of similar-type inferencing, symbolic manipulations, etc.

Ada (Goos and Hartmanis, 1983) has come as the product of the "history's largest design effort" (Sebesta, 1993). Developed by the Department of Defense, it involved hundreds of participants from several dozens of organizations, and went through a 3-phase process before its design was selected and frozen for development. Named after the mathematician Augusta Ada Byron (the poet's daughter), considered by historians as the first programmer, Ada is another example of exercise in complexity. It offers a tremendous set of facilities with imperative, functional and object-oriented programming features, but its future viability is in doubt.

From Programming Languages to Modeling Languages

Engineering is the task of creating new or revamping existing artifacts, using the fundamental analytical insights offered by science and relying on the synthetic skills of the human engineer. If computers are to play a significant role in engineering work, the man-machine interaction must change. Central to this change is the role of *modeling languages*.

Every task that an engineer is involved in includes *modeling*, i.e. the activity of creating representations of the artifacts under construction. Models are needed to represent (a) molecules, (b) mixtures of chemicals, (c) physical and chemical properties, (d) processing units, (e) operations, (f) control loops, etc. Such *declarative models* provide the list of parameters and variables that reveal the state of an artifact (data models), or the set of declarative relationships (mathematical models) which describe the behavior of the artifacts. The imperative, or procedural character of the dominant programming languages, e.g. FORTRAN, has had a profound effect on the style of models created by chemical engineers. They have all been procedural models, and as a result they can have a limited scope of applicability, covering only the intended domain of use. Thus, we can explain the explosive proliferation of different models for the same artifact.

The discussion in the previous section has hinted that *declarative programming* has the ability to generate generic data models and mathematical models with virtually unlimited scope of applicability. This observation has led to the creation of modeling languages, as natural descendants of the declarative programming languages.

A. N. Whitehead observed that: "*By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems* …." Indeed, a chemical engineering student or practitioner should not need to master all intricacies of FORTRAN 90 or ANSI C in order to create and run computer programs. The programming language "… *is not just a way of getting a computer to perform operations, but rather it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute*" (Abelson et al., 1985). Thus, by defining appropriate primitives, means of combination, and means of abstraction, we can create domain-specialized, problem-oriented languages which can provide high-level modeling facilities. Such languages already exist and form the intellectual backbone of all engineering domains, e.g. the languages of: (i) "electrical networks" for modeling devices in terms of discrete electrical elements; (ii) "civil engineering structures," using girders, rods, plates, beams, shells; (iii) "unit operations" in chemical engineering. Today's complexities of the processing systems have induced a refinement of the language of unit operations into that of "*foundation phenomena*." The computer-aided deployment of such domain-specific languages has led to a new generation of programming languages, called *modeling languages*.

MODEL.LA. (Stephanopoulos, et al., 1990a and b) is a computer-aided implementation of the phenomena-based language for chemical engineers. It is characterized by its capacity to support the articulation and accept the student's declarations of his or her knowledge of the model context, including assumptions, simplifications, and scope of task. The language is, for the user, fully declarative. In the next section we will discuss its design characteristics that support the articulation of procedural tasks. The user of MODEL.LA. "writes" programs in an, almost, English-like, natural language, employing known syntax and chemical engineering-oriented vocabulary. Interpretation of the "sentences," composed by the user, allows MODEL.LA. to identify the boundaries of a system, relevant streams and the chemicals flowing with them, the set of physical and chemical phenomena occurring in the system (e.g. reaction, separation, heat exchange), the mechanisms and the expressions that should be used to describe rate-based phenomena (e.g. chemical reaction, diffusion, heat transfer), and equilibrium conditions, the expressions that should be used to compute physical properties etc.

Modeling languages, like MODEL.LA., do relieve the user's brain from unnecessary work, so that the user can concentrate on the engineering problem and its requirements. During the last 5 years we have witnessed a significant increase in the number of modeling languages, all of which, with variable degree of success, provide the declarative paradigm that will dominate the future work. Typical examples of such languages are the following: ASCEND (Piela et al., 1991), Omola (Nilsson, 1993), Modass (Sorlie, 1990), HPT (Woods, 1993), gPROMS (Barton, 1992; Pantelides and Barton, 1993), DIVA (Kröner et al., 1990), LCR (Han et al., 1995a).

Generic modeling languages should conform with the *declarative and functional programming paradigms.* In the previous paradigms we discussed the need for the former, but the requirement for functional programming is almost as important. In assembling models from the elementary foundation phenomena, the primary task for a computer is *symbolic manipulations.* The functional programming paradigm maintains the consistency between the "computational artifact" called model and its mathematical (arithmetic and logical) counterpart. Thus, we can maintain the *referential transparency* between mathematics and computer programs, a feature which is lost with procedural programming. Deployment of modeling languages through a procedural programming language, e.g. FORTRAN, C, Pascal, is either limited in scope or/and prone to failures, as the complexity of the language increases to serve the ever expanding needs.

MODEL.LA. was developed in an object-oriented dialect of LISP, a language which through its conformity to LISP is functional, while its object-oriented character provides the declarative programming facility.

Task-Oriented Languages

If modeling languages elevate the forms, used to describe engineering artifacts, to the level of a natural language, *task-oriented languages* expand those facilities and allow the high-level description of engineering methodologies, all of which are procedural in character. For example, the conceptual design of a chemical process (Douglas, 1988), involves the transformation of functional specifications into realizable physical objects (i.e. processing units), properly interconnected and operated to yield the desired process. The methodology to carry out the above transformation is clearly a procedure, and involves a series of executable tasks. It is very natural to consider imperative programming as the natural setting for creating a program that would implement the design methodology.

Upon closer inspection of Douglas' hierarchical design methodology, we realize that the resulting software would be a typical spaghetti-like design, unmaintainable, hard to expand with new design knowledge, and virtually impossible to track its logic and thus guarantee its integrity. Here is where a significant lesson can be learned from our experience with programming languages, namely, "the essential material (of an introductory course in computer science) is not the syntax of particular programming-language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems" (Abelson et al., 1985).

Consequently, as the intellectual complexity of engineering methodologies increases, it is important that we develop programming media, i.e. languages, which allow us to control the complexity and maintain their integrity. Here is where the task-oriented languages have started playing a significant role. Consider, for example, the *design-agents* used by Han (Han, 1993; Han et al. 1995b and 1995c), to model the design activities during the conceptual design of a chemical process. Each design-agent is a computational object with a specific set of data and a list of procedures, which, upon execution, define what the design-agent can do. By stringing generic design-agents together, we can create very complex programs, which emulate very complex design methodologies, and still maintain control of complexity and easy verification of the programs' (i.e., of the methodologies) integrity. As new knowledge becomes available, requiring a change on how a design-agent carries out its tasks, one only needs to modify the set of attributes and the algorithms of the procedures, characterizing the "skills" or "behavior" of the design-agent. Similarly, one can refine a design-agent by another one carrying out similar tasks by different approaches, expand the design methodology with new design-agents, or "re-wire" the overall design methodology by reconnecting the agents in different ways.

The modularized construction of complex procedures that started with the concept of a "subroutine" and a "function," has been brought up to levels of advanced sophistication with the object-oriented character of *task-oriented languages*. These languages provide linguistic constructs for the definition of "tasks" as generic programming elements that possess specific behavior. Any engineering methodology that requires the solution of a set of linear equations, will simply send the appropriate message to the "task" called *LINEAR-EQUATION-SOLVER*. Similarly, engineering methodologies may invoke instances of the generic tasks *SEQUENCE-DISTILLATION-COLUMNS*, *SELECT-REACTOR*, *FORMULATE-BALANCES*, *COST-HEAT-EXCHANGER*, etc.

So, task-oriented languages provide the means for the description of engineering procedures, while modeling languages support the declarative description of engineering artifacts. When taken together as the two complements of one *engineering programming language*, they provide the paradigm of future programming. A paradigm which frees the engineer from the drudgery of lower-level programming and allows him/her to concentrate at the engineering task at hand.

The first examples of such languages are already here (although not fully declarative or functional): the graphic user interfaces that support the creation of programs around relational database management systems using SQL; the visual programming in Visual Basic or Visual C++; the script-based language of MATLAB; the algebraic language, GAMS; the symbolic language of MACSYMA; the object-oriented language of G2.

Paradigms of Software Design

Software productivity has progressed much more slowly than the hardware developments. Despite the development of high-level, and then higher-level languages, structured methods for developing software, and tools for configuration management and tracking requirements, bugs, and changes, the net gain is nowhere close to that realized in hardware developments. This is due to the fact that programming remains basically unchanged, and remains as it has been formed by the needs of numerical computations, i.e. a person-centered activity which results in a highly detailed formal object. Figure 1 illustrates the philosophy of the traditional paradigm for software development. It starts with an informal statement of the desired requirements, e.g. schedule batch operations in an open flow-shop with intermediate storage, using heuristic implicit enumeration according to branch-and-bounding conditions A and B. It proceeds to analyze these informal requirements and define formal specifications which will be converted into a software system. During this state we have the formal statement of the solution methodology, as this is expressed by the resulting logical flow diagram. As soon as this step has been completed, the software designer has locked himself/herself into a specific problem-formulation and problem-solution and the coding can start, leading to the final source program. This manual conversion from informal requirements to programs is error prone and labor intensive. Once the source program has been completed, validation, testing and tuning begins. These activities are carried out at the source-code level, and experience has amply demonstrated that they are quite labor-intensive and painful. Any change in the problem-formulation or problem-solving strategy requires major surgery of the source code, amplifying the inefficiency of the software development process.

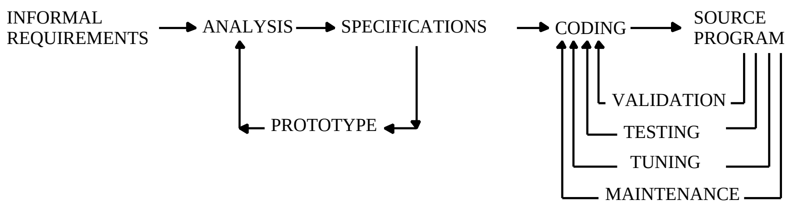

Figure 1. The traditional paradigm of software design.

Such philosophy for software design and development, although it may be acceptable for numerically intensive computer programs, is very poor and unacceptable for applications, which involve problem-specific contextual interaction between the human and the computer. For example, depending on the impact that various operating conditions have on the economics of process operations, the designer will determine the next state of activities during the design of control configurations for a chemical plant. Furthermore, the scheduling of batch operations is strongly influenced by plant-specific, operations-specific, and product-specific considerations. Consequently, we cannot have an all purpose problem-formulation and as a result the problem-solving methodology varies. Similar aspects one encounters in the formulation and solution of other problems such as; synthesis of operating procedures, diagnosis of plant or operating faults, identification of potential hazards in chemical plants, etc. Of course, one could try to capture all possible scenarios for all possible types of problem-formulations and direct each to a separate branch involving a problem-solving methodology with distinct features, thus creating a more complex structure of the logical flow diagram for a, say, FORTRAN program. But, this alternative is almost, by definition, impossible. Even more, such a system will lack facilities to "explain" the rationale of the problem-solving methodology (e.g. design, planning, diagnosis), and does not allow the human's own knowledge from past experience to be used effectively for the simple reason that they have not been anticipated by the program and have not been included in it.

By way of a summary, all the above discussion indicates that programs designed and developed using a model of fixed procedural knowledge are not satisfactory to easily capture varying problem-solving algorithms. A different paradigm is needed, along the lines of that shown in Figure 2. This flexible paradigm carries out the validation, testing, adaptation and maintenance of the software system within a symbolic, interpretive environment. Consequently, these tasks can be carried out at a higher level of abstraction, before the formal specifications have been cast in concrete. The human user's intervention to offer subjective preferences and decisions is allowed before the coding starts (see Figure 2), thus influencing decisively the set of formal specifications upon which the code will be generated. Thus, only the tuning of the code is done at the program's source level, in order to improve speed and efficiency. Such a programming paradigm allows a rather explicit description of what the resulting program does and why it does not require extensive work if changes must be made and provides a fairly precise framework for the simultaneous development of several of its parts by different programmers. Almost all software systems are presently designed and developed using the flexible paradigm of Figure 2. Finally, it should be noted that as the technology of automatic programming is advancing, allowing the automatic computer-driven conversion of formal specifications into a source code, the paradigm of Figure 2 will dominate over the traditional one of Figure 1 even for the development of self-standing numerically intensive software systems.

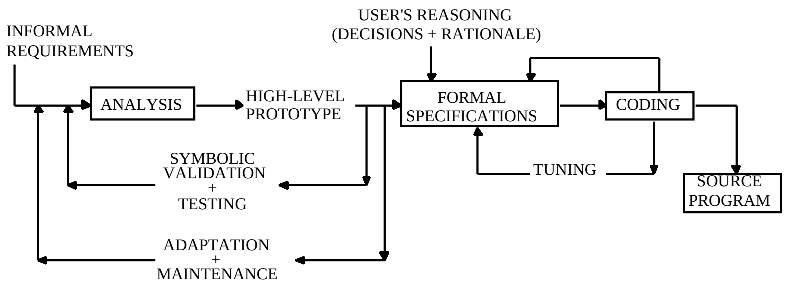

Figure 2. The evolving new paradigm for software design.

Programming and Computer Science for Chemical Engineers?

Chemical engineers are supposed to work on the creation of new products and processes and the optimal revamping of existing ones. Through the use of computers, they are expected to enhance their skills and abilities to carry out their main tasks, as described above. So, is there any role that programming and computer-science could play in their education and professional advancement? We believe that the answer is yes, but the rationale is more important than the answer itself.

The vast majority of chemical engineers are not and will not be writing programs, using the prevailing programming languages, as a regular part of their responsibilities. They will be using the computers, though, far more tomorrow than today, tackling a far broader number of problems than they do today. The implication is two-fold: (1) Since they will not be writing programs, they do not need to be taught computer-science and programming. (2) Some one must create the high-level languages that will support modeling and task-oriented description of engineering methodologies. Let us look at each one a little more closely.

The first implication is correct if programming in, say, C was the objective. But, “every computer program is a model, hatched in the mind, of a real or mental process” (Abelson et al., 1985) and through each program one must express in a disciplined and formal manner, poorly-understood and sloppily-formulated ideas. Such discipline is an essential ingredient of good engineering and therein lies the value of computer-science for a chemical engineer. In building such discipline one learns the concepts of; building abstractions with data or procedures, the interchangability of data and procedures, recursion, or iteration, modularity, reasoning in logic, representation of data, controlling complexity, and many others all of which play a central role in every engineering activity beyond programming itself. All concepts, taught in such a computer-science course, help strengthen the dual pivot of engineering work; problem formulation and problem-solving. Therefore, current curricula emphasizing the syntax of particular language or giving the students an uncritical exposure to the use of spreadsheets, databases, graphics, and prepackaged numerical algorithms, provide a short-term skill and not long-term sustenance in a computer-driven professional world.

The second implication has some interesting consequences. Chemical engineers and not computer-scientists must drive the process for the creation of chemical engineering-oriented modeling and task-oriented languages. They are the ones who know what these languages ought to be. But, chemical engineers do not, normally, possess the education and sophistication to create formal languages. Collaboration with computer scientists is not only necessary, but also desirable; being forced to formalize our ideas on modeling and engineering methodologies, we may learn a better process to teach and practice modeling and problem-solving.

References

Abelson, H. and G. Sussman (1985). *Structure and Interpretation of Computer Programs*, Cambridge, MA, MIT Press.

Barton, P.I. (1992). *The Modeling and simulation of combined discrete/continuous processes*. Ph.D. thesis, University of London.

Dahl, O.-J. and K. Nygaard (1967). *SIMULA 67 Common Base Proposal*, Norwegian Computing Center Document, Olso.

Department of Defense (1960). *COBOL, Initial Specifications for a Common Business Oriented Language*.

Douglas, J. M. (1988). *Conceptual Design of Chemical Processes*, McGraw-Hill.

Goos, G. and J. Hartmanis (eds.) (1983). *The Programming Language Ada Reference Manual*, American National Standards Institute, ANSI/MIL-STD-1815A-1983, Lecture Notes in Computer Science 155, Springer-Verlag, New York.

Han, C. (1993). *Human-Aided, Computer-Based Design Paradigm: The Automation of Conceptual Process Design*, Ph.D. Thesis, Department of Chemical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Han, C., C. Nagel, and G. Stephanopoulos (1995a). Modeling Languages: Declarative and Imperative Descriptions of Chemical Reactions and Processing Systems, *Paradigms of Intelligent Systems in Process Engineering*, Academic Press.

Han, C., J. Douglas, and G. Stephanopoulos (1995b). Automation in Design: The Conceptual Synthesis of Chemical Processing Schemes, in *Paradigms of Intelligent Systems in Process Engineering*. (G. Stephanopoulos and C. Han, Eds.), Academic Press.

Han, C., J. Douglas, and G. Stephanopoulos (1995c). Agent-based approach to a design support system for the synthesis of continuous chemical processes, ESCAPE-95.

Kay, A. (1969). *The Reactive Engine*, Ph.D. Thesis, University of Utah, September.

Kröner, A., P. Holl, W. Marquardt and E. D. Gilles (1990). DIVA - An open architecture for dynamic process simulation, *Comput. Chem. Engng*, **14**, 1289-1295.

Mather, D.G. and S. V. Waite (Eds.) (1971). *BASIC*, 6th ed. University Press of New England, Hanover, NH.

Meyer, B. (1990). *Introduction to the Theory of Programming Languages*, Prentice Hall Inc., Englewood Cliffs, NJ 07632.

Naur, P. (ed.) (1960). Report on the Algorithmic Language ALGOL 60, *Commun. ACM*, **3**(5), 299-314.

Nilsson, B., *Object-Oriented Modeling of Chemical Processes*, Doctoral thesis, Department of Automatic Control, Lund Institute of Technology (1993).

Pantelides, C.C. and P. Barton (1993). Equation-oriented dynamic simulation: current status and future perspectives, *Comp. Chem. Engng*., **17S**, 263-285.

Piela, P. C., T. G. Epperly, K. M. Westerberg, and A. W. Westerberg, ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis: The Modeling Language, *Comp. Chem. Engng*, **15**(1) p.53 (1991).

Roussel, P. (1975). *PROLOG: Manual de Reference et D'utilisation, Research Report*, Artificial Intelligence Group, Univ. of Aix-Marseille, Luming, France.

Sebesta, R. W. (1993). *Concepts of Programming Languages*, 2nd ed., The Benjamin/Cummings Publishing Co.

Sørlie, C. F., *A Computer Environment for Process Modeling*, Ph.D. thesis, Department of Chemical Engineering, The Norwegian Institute of Technology (1990).

Stephanopoulos, G., G. Henning and H. Leone (1990a). MODEL.LA.: A Modeling Language for Process Engineering. Part I: The Formal Framework, *Comput. Chem. Engng.*, **14**, 813-846.

Stephanopoulos, G., G. Henning and H. Leone (1990b). MODEL.LA.: A Modeling Language for Process Engineering. Part I: The Formal Framework, *Comput. Chem. Engng.*, **14**, 847-869.

Stroustrup, B. (1983). Adding Classes to C: An Exercise in Language Evolution, *Software-Practice and Experience*, **13**.

Wirth, N. (1971). The Programming Language Pascal, *Acta Informatica*, **1**(1), 35-63.

Wexelblat, R. L. (ed.) (1981). *History of Programming Languages*, Academic Press, New York.

Woods, E. A., *The Hybrid Phenomena Theory: A framework Integrating Structural Descriptions with State Space Modeling and Simulation*, Ph.D. thesis, Department of Engineering Cybernetics, The Norwegian Institute of Technology (1993).

Zuse, K. (1972). Der Plankalkül, Manuscript prepared in 1945, published in *Berichte der Gesellschaft fur Mathematik und Datenverarbeitung*, No. 63 (Bonn, 1972); Part 3, 285 pp. English translation of all but pp. 176-196 in No. 106 (Bonn, 1976), pp. 42-244.