6

What Is Computer Science and Engineering?

Chapter 1 provided a brief sketch of computer science and engineering as an intellectual discipline. This chapter elaborates on that discussion, discusses some key structural features of the field, and provides some history on some of the major intellectual accomplishments of the field in a few selected areas. For the reader's convenience, the Chapter 1 section "Computer Science and Engineering" is reproduced in its entirety here.

COMPUTER SCIENCE AND ENGINEERING

Computational power—however measured—has increased dramatically in the last several decades. What is the source of this increase?

The contributions of solid-state physicists and materials scientists to the increase of computer power are undeniable; their efforts have made successive generations of electronic components ever smaller, faster, lighter, and cheaper. But the ability to organize these components into useful computer hardware (e.g., processors, storage devices, displays) and to write the software required (e.g., spreadsheets, electronic mail packages, databases) to exploit this hardware are primarily the fruits of CS&E. Further advances in computer power and usability will also depend in large part on pushing back the frontiers of CS&E.

Intellectually, the "science" in "computer science and engineering" connotes understanding of computing activities, through mathematical and engineering models and based on theory and abstraction. The term "engineering" in "computer science and engineering" refers to the practical application, based on abstraction and design, of the scientific principles and methodologies to the development and maintenance of computer systems—be they composed of hardware, software, or both.[1] Thus both science and engineering characterize the approach of CS&E professionals to their object of study.

What is the object of study? For the physicist, the object of study may be an atom or a star. For the biologist, it may be a cell or a plant. But computer scientists and engineers focus on information, on the ways of representing and processing information, and on the machines and systems that perform these tasks.

The key intellectual themes in CS&E are algorithmic thinking, the representation of information, and computer programs. An algorithm is an unambiguous sequence of steps for processing information, and computer scientists and engineers tend to believe in an algorithmic approach to solving problems. In the words of Donald Knuth, one of the leaders of CS&E:

> CS&E is a field that attracts a different kind of thinker. I believe that one who is a natural computer scientist thinks algorithmically. Such people are especially good at dealing with situations where different rules apply in different cases; they are individuals who can rapidly change levels of abstraction, simultaneously seeing things "in the large" and "in the small."[2]

The second key theme is the selection of appropriate representations of information; indeed, designing data structures is often the first step in designing an algorithm. Much as with physics, where picking the right frame of reference and right coordinate system is critical to a simple solution, picking one data structure or another can make a problem easy or hard, its solution slow or fast.

The issues are twofold: (1) how should the abstraction be represented, and (2) how should the representation be properly structured to allow efficient access for common operations? A classic example is the problem of representing parts, suppliers, and customers. Each of these entities is represented by its attributes (e.g., a customer has a name, an address, a billing number, and so on). Each supplier has a price list, and each customer has a set of outstanding orders to each supplier. Thus there are five record types: parts, suppliers, customers, price, and orders. The problem is to organize the data so that it is easy to answer questions like: Which supplier has the lowest price on part P?, or, Who is the largest customer of supplier S? By clustering related data together, and by constructing auxiliary indices on the data, it becomes possible to answer such questions quickly without having to search the entire database.

The two examples below also illustrate the importance of proper representation of information:

- A "white pages" telephone directory is arranged by name: knowing the name, it is possible to look up a telephone number. But a "criss-cross" directory that is arranged by number is necessary when one needs to identify the caller associated with a given number. Each directory contains the same information, but the different structuring of the information makes each directory useful in its own way.
- A circle can be represented by an equation or by a set of points. A circle to be drawn on a display screen may be more conveniently represented as a set of points, whereas an equation may be a better representation if a problem calls for determining if a given point lies inside or outside the circle.

A computer program expresses algorithms and structures information using a programming language. Such languages provide a way to represent an algorithm precisely enough that a "high-level" description (i.e., one that is easily understood by humans) can be mechanically translated ("compiled") into a "low-level" version that the computer can carry out ("execute"); the execution of a program by a computer is what allows the algorithm to come alive, instructing the computer to perform the tasks the person has requested. Computer programs are thus the essential link between intellectual constructs such as algorithms and information representations and the computers that enable the information revolution.

Computer programs enable the computer scientist and engineer to feel the excitement of seeing something spring to life from the "mind's eye" and of creating information artifacts that have considerable practical utility for people in all walks of life. Fred Brooks has captured the excitement of programming:

> The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds castles in the air, creating by the exertion of the imagination. . . . Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. . . . The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.[3]

Programmers are in equal portions playwright and puppeteer, working as a novelist would if he could make his characters come to life simply by touching the keys of his typewriter. As Ivan Sutherland, the father of computer graphics, has said,

> Through computer displays I have landed an airplane on the deck of a moving carrier, observed a nuclear particle hit a potential well, flown in a rocket at nearly the speed of light, and watched a computer reveal its innermost workings.[4]

Programming is an enormously challenging intellectual activity. Apart from deciding on appropriate algorithms and representations of information, perhaps the most fundamental issue in developing computer programs arises from the fact that the computer (unlike other similar devices such as non-programmable calculators) has the ability to take different courses of action based on the outcome of various decisions. Here are three examples of decisions that programmers convey to a computer:

- Find a particular name in a list and dial the telephone number associated with it.
- If this point lies within this circle then color it black; otherwise color it white.
- While the input data are greater than zero, display them on the screen.

When a program does not involve such decisions, the exact sequence of steps (i.e., the "execution path") is known in advance. But in a program that involves many such decisions, the sequence of steps cannot be known in advance. Thus the programmer must anticipate all possible execution paths. The problem is that the number of possible paths grows very rapidly with the number of decisions: a program with only 10 "yes" or "no" decisions can have over 1000 possible paths, and one with 20 such decisions can have over 1 million.

Algorithmic thinking, information representation, and computer programs are themes central to all subfields of CS&E research. Box 6.1 illustrates a typical taxonomy of these subfields. Consider the subarea of computer architecture. Computer engineers must have a basic understanding of the algorithms that will be executed on the computers they design, as illustrated by today's designers of parallel and concurrent computers. Indeed, computer engineers are faced with many decisions that involve the selection of appropriate algorithms, since any programmable algorithm can be implemented in hardware. Through a better understanding of algorithms, computer

BOX 6.1 A TAXONOMY OF SUBFIELDS IN CS&E

- Algorithms and data structures
- Programming languages
- Computer architecture
- Numeric and symbolic computation
- Operating systems
- Software engineering
- Databases and information retrieval
- Artificial intelligence and robotics
- Human-computer interaction

Each of these areas involves elements of theory, abstraction, and design. Theory is based on mathematics and follows the mathematician's methodology (defining objects, proving theorems); abstraction is based on the investigative approach of the scientist (hypothesizing, making predictions, collecting data); design is based on the methodology of the engineer (defining requirements and specifications, implementing a system, testing a system).

SOURCE: Peter Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, Joe Turner, and Paul R. Young, "Computing as a Discipline," *Communications of the ACM*, Volume 32(1), January 1989, pp. 9-23.

engineers can better optimize the match between their hardware and the programs that will run on them.

Those who design computer languages (item two in Box 6.1) with which people write programs also concern themselves with algorithms and information representation. Computer languages often differ in the ease with which various types of algorithms can be expressed and in their ability to represent different types of information. For example, a computer language such as Fortran is particularly convenient for implementing iterative algorithms for numerical calculation, whereas Cobol may be much more convenient for problems that call for the manipulation and the input and output of large amounts of textual data. The language Lisp is useful for manipulating symbolic relations, while Ada is specifically designed for "embedded" computing problems (e.g., real-time flight control).

The themes of algorithms, programs, and information representation also provide material for intellectual study in and of themselves,

BOX 6.2 ABOUT THE STUDY OF ALGORITHMS

How many steps are necessary to solve a given problem? This question led to the development of the area known as computational complexity. Consider alphabetizing a list of 1000 names. A straightforward algorithm ("insertion sort") takes on the order of a million (i.e., 1000 × 1000) one-to-one comparisons of names in the worst case, but a clever algorithm ("heap sort") would take just 10,000 comparisons in the worst case ($1000 \times \log_2 1000$ or about 1000×10). Further, this is the best possible result, for it has been shown that sorting a list of n items requires $n \log_2 n$ pair-wise comparisons in the worst case, no matter what algorithm is used. Theoreticians have found arguments that apply to whole classes of algorithms and problems, opening questions about computing that have not yet been solved.

often with important practical results. The study of algorithms within CS&E is as challenging as any area of mathematics; it has practical importance as well, since improperly chosen algorithms may solve problems in a highly inefficient manner, and problems can have intrinsic limits on how many steps are needed to solve them (Box 6.2). The study of programs is a broad area, ranging from the highly formal study of mathematically proving programs correct to very practical considerations regarding tools with which to specify, write, debug, maintain, and modify very large software systems (otherwise called software engineering). Information representation is the central theme underlying the study of data structures (how information can best be represented for computer processing) and much of human-computer interaction (how information can best be represented to maximize its utility for human beings).

ABSTRACTIONS IN COMPUTER SYSTEMS

While algorithmic thinking, computer programs, and information representation are the key intellectual themes in the study of CS&E, the design, construction, and operation of computer systems require talents from a wide range of fields, such as electrical engineering for hardware, logic and mathematical analysis for writing programs, and psychology for the design of better user interfaces, to name just a few. This breadth reflects the fact that computer systems are among

the most complicated objects ever created by human beings. For example, the fastest computers today have around 100 billion transistors, while the personal computer on one's desk may have "only" a few million. People routinely use computer programs that are hundreds of thousands of lines long, and the largest software systems involve tens of millions of lines; printed at 50 lines per paper page, such a system might weigh several tons.

One of the most effective ways to cope with such complexity is to use abstractions. Abstraction is a generic technique that allows the human scientist or engineer to focus only on certain features of an object or artifact while hiding the others. However, while scientists in other disciplines typically use abstractions as a way to simplify calculations for purposes of analysis, those in CS&E are concerned with abstractions for purposes of synthesis: to build working computer systems. Other engineering disciplines also use abstractions as the basis of synthesis, but the "stuff" of these disciplines—engineered and created artifacts—is ultimately governed by the tangible reality of nature, which itself imposes structure on these abstractions. Computer programs are not similarly limited; instead, they are built out of ideas and information whose structuring is constrained only by human imagination. This extraordinary flexibility in structuring information has no analog in the material world.

The focus of the computer scientist or engineer in creating an abstraction is to hide the complexity of operation "underneath the abstraction" while offering a simple and useful set of services "on top of it." Using such abstractions is CS&E's principal technique for organizing and constructing very sophisticated computer systems. One particularly useful abstraction uses hardware, system software, and application software as successive layers on which useful computer systems can be built.

At the center of all computer systems is hardware, i.e., the portion of a computer system that one can see and touch. Hardware is divided into three principal components:

- *Processors.* Processors perform the arithmetic and logical operations, much like the arithmetic operations available in a hand-held calculator. Processors also handle conditional behavior, executing one or another set of operations depending on the outcome of some decision.
- *Memory (short term and long term).* Memory hardware is like the memory function of a calculator, since it can be used to save and later retrieve information. Such information may be lost in a calculator when the power is turned off, as it is in the short-term memory of

most computer systems. Computer systems also need a long-term memory that doesn't forget when the power is lost.

- *Communication (user-machine and machine-machine).* For computers to be useful, they must be able to communicate with people, and so display screens and keyboards are critical components of computer systems. For maximum usefulness, computers must be able to communicate with other computers, and so computers are often equipped with modems or other network connections that can transmit and receive data to and from other computers.

Readers familiar with personal computers are likely to have encountered the technical names or brand names for these three types of hardware. Personal computers might use an Intel 80486 processor as the processor hardware, dynamic random access memory (DRAM) as the short-term memory hardware, and disks as the long-term memory hardware. The user-computer communication hardware for personal computers is the keyboard, mouse, and video screen, while examples of machine-machine communication hardware are telephone modems and networks such as Appletalk and Ethernet.

While the characteristics of the underlying hardware are what ultimately determine the computational power of a computer system, direct manipulation of hardware would be cumbersome, difficult, and error-prone, a lesson learned in the earliest days of computing when programmers literally connected wires to program a computer. Thus computer scientists and engineers construct a layer of "system software" around the hardware.

System software hides the details of machine operation from the user, while providing services that most users will require, services such as displaying information on a screen, reading and writing information to and from a disk drive, and so on. System software is commonly divided into three components:

- *Operating system*—software that controls the hardware and orchestrates how other programs work together. The operating system may also include network software that allows computers to communicate with one another.
- *Tools*—the software (e.g., compilers, debuggers, linkers, database management systems) that allows programmers to write programs that will perform a specific task. Compilers and linkers translate "high-level" languages into machine language, i.e., the ones and zeros that govern machine operation at the lowest level. Debuggers help programmers to find errors in their work. Database management systems store, organize, and retrieve data conveniently.
- *User interface*—the software that enables the user to interact with the machine.

Once again, personal computer users are likely already familiar with the brand names of these pieces of system software. MS-DOS in IBM Personal Computers and the System in Macintoshes are examples of operating system software; Novell is the brand name of one type of widely used networking software; Dbase IV is a popular database management system; Microsoft Basic and Borland C are examples of compiler system software; and Microsoft Windows and the Macintosh Desktop are examples of user interface system software.

While the services provided by system software are usually required by all users, they are not in themselves sufficient to provide computing that is useful in solving the problems of the end user, such as the secretary or the accountant or the pilot. Software designed to solve specific problems is called applications software; examples include word processors, spreadsheets, climate models, automatic teller machines, electronic mail, airline reservation systems, engineering structural analysis programs, real-time aircraft control systems, and so on. Such software makes use of the services provided by system software as fundamental building blocks linked together in such a way that useful computing can be done. Examples of applications software include WordPerfect and Word (word-processing applications) and Lotus 1-2-3 and Excel (spreadsheet applications), and there are as many varieties of applications software as there are different computing problems to solve.

The frequent use of system software services by all varieties of applications software underscores an important economic point: providing these services in system software means that developers of applications software need not spend their time and effort in developing these services, but rather can concentrate on programming that is specifically related to solving the problem of interest to the end user. The result is that it becomes much easier to develop applications, leading to more and higher-quality computing applications than might otherwise be expected.

This description of layering a computer system into hardware, system software, and applications software is a simple example of abstraction.[5] (Box 6.3 explains why abstraction is so powerful a tool for the computer scientist and engineer.) But it suffices to illustrate one very important use of abstraction in computer systems: each layer provides the capability to specify that certain tasks be carried out without specifying *how* they should be carried out. In general, computing artifacts embody many different abstractions that capture many different levels of detail.

A good abstraction is one that captures the important features of an artifact and allows the user to ignore the irrelevant ones. (The features decided to be important collectively constitute the interface

BOX 6.3 WHY IS ABSTRACTION POSSIBLE?

Although abstractions are pervasive in the construction of modern computer systems, it is not obvious that abstractions should work. How is it that the abstractions between the user and the machine do not reduce the available computational capability?

Results from the theory of computation provide the answer. In 1936, the British mathematician Alan Turing explored the computational capabilities of an abstract computing device, now known as the Turing machine. This work laid the scientific foundations for the age of computing and led directly to modern computers with internally stored programs. Turing showed that there exist universal computing machines that can simulate the performance of *any* computing machine, provided the universal machine has a description of the instruction set of the machine being simulated and has sufficient memory.

This crucial insight underlies much of computing, and is fundamental to:

- the interchangeability of hardware and software, since a universal machine with a very small instruction set (i.e., very little hardware) can simulate, with adequate programming, a computer with a much larger instruction set.
- the existence of high-level computer languages. Such languages are much more easily understood by human beings than are the "machine language" of ones and zeroes that machines can execute. Programs written in these languages can be translated into the equivalent machine language programs, yielding the desired result without sacrificing expressive power.
- the ubiquity of computer "bugs," i.e., mistakes in programs. Bugs are possible because a computer that is universal can perform any possible computation, the wrong ones as well as the right ones. But the difference between a program that does the right thing and one that does the wrong thing may be as small as one bit; a small mistake in a program can lead the computer to execute a computation very different from the one intended.

In general, the different layers of the computing system tame the raw universal power of the hardware by hiding (but not eliminating) the sensitivity of the underlying machine to the exact formulation of its program. Successive layers make the computer much easier to use and permit the use of higher-level languages in which the desired computation can still be described very precisely but with less chance of error.

of the artifact to the outside world.) By hiding details, an abstraction can make working with an artifact easier and less subject to error. But hiding details is not cost free—in a particular programming problem, access to a hidden detail might in fact be quite useful to the person who will use that abstraction. Thus deciding how to construct an abstraction (i.e., deciding what is important and irrelevant) is one of the most challenging intellectual issues in CS&E.

A simple example of this issue is the writing of a system program that displays data on a screen. The program may allow the user to specify the size of the screen, or it may assume that "one screen size fits all." In the first case, the user is given control over the screen size, at the cost of having to remember to specify it every time this program is used. In the second case, the user does not need to choose the screen size, but also loses the flexibility to use the program on screens of different size.

A second challenging issue is how to manage all of the details that are hidden. The fact that they are hidden beneath the interface does not mean that they are irrelevant, but only that the computer scientist or engineer must design and implement approaches to handle these details "automatically," i.e., without external specification. Decisions about how best to handle these details are subject to numerous trade-offs. For example, a designer of computer systems may face the question of whether to implement a certain function in hardware or in software. By implementing the function in hardware, the designer gains speed of execution, but at the cost of making the function very difficult to change. A function implemented in software executes more slowly, but is much easier to change.

Abstractions enable computer scientists and engineers to deal with large differences of scale. At the highest level of abstraction, a person editing a document in a word-processing program needs only to mark the start and the end of the block of text and then press the DEL key to delete the block of text. But these few keystrokes can initiate the execution of thousands or tens of thousands of basic instructions in the machine's hardware. Only by inserting abstractions intermediate between the user's keystrokes and the basic machine instructions can those keystrokes be predictably translated into the correct set of instructions. Thus the programmer who writes the word-processing program will provide one abstraction (in this case called a subroutine) that will delete the block from the screen, a second to reformat and redisplay the remaining text, and a third to save the changed document to disk. Within each of these abstractions will be lower-level abstractions that perform smaller tasks; each succes-

sive lower-level abstraction will control the execution of ever smaller numbers of basic machine instructions.

Abstractions are also central to dealing with problems of different size (e.g., searching a database with a thousand or a billion records) or hardware of different capability (e.g., a computer that performs a million or 10 billion calculations per second). Ideally, the user ought to be presented with the same high-level abstraction in each of these cases, while the differences in problem size or hardware capability ought to be handled at lower levels of abstraction. In other words, the user ought not be obligated to change his or her approach simply because the problem changes in size or the hardware becomes more capable.

However, in practice today, users must often pay logical and conceptual attention to differences in hardware capability and problem size. Querying a database of a billion records requires a strategy different from one for querying a database of a thousand records, since narrowing the search is much more difficult in the larger case; writing a program to run on a computer that performs 10 billion calculations per second most likely requires an approach different from one for a program written to run on a computer that performs one million calculations per second, since the former is likely to be a parallel computer and the latter a serial computer.

Bridging the gap between today's practice and the ideal—making the "ought-to-be's" come true—is the goal of much CS&E research today.

SELECTED ACCOMPLISHMENTS

This section is intended to be partly tutorial (describing some of the intellectual issues in the field) and partly historical (describing accomplishments that have had some impact on computing practice). The committee also wishes to bring to the reader's attention a 1989 report of the NSF advisory committee for computer research, *Computer Science: Achievements and Opportunities*, that provides a good discussion of the accomplishments of CS&E from the perspective of the field's own internal logic and intellectual discipline.[6] Further, the committee stresses that this sampling of intellectual accomplishments does not differentiate between those made by academia and those made by industry; as the discussion in Chapters 1 and 2 suggests, the determination of "credit" for any given accomplishment would be a difficult task indeed.

Systems and Architectures

Computing-system architects and building architects play similar roles: both design structures that satisfy human needs and that can be constructed economically from available materials. Buildings have many sizes, shapes, styles, and purposes; similarly, computing systems range from the single microelectronic chips that animate calculators, fuel-injection controls, cardiac pacemakers, and telephone-answering machines, to the geographically distributed networks of thousands of computers. Just as buildings are constructed from a variety of materials, so also do computing systems incorporate many different manufacturing processes and technologies, from microelectronic chips, circuit boards, and magnetic disks to the software that tailors the machine to an application or for general programming use. Standardization yields economies both in buildings and in computers; for example, pre-cut 8-foot studs and pre-made windows are commonly used in residential houses, whereas 8-bit bytes, commodity processor and memory chips, and standardized programming notations and system conventions may be used in many different models of computing systems.

For both computing systems and buildings, the conceptual distance from the available materials and technologies to the requirements established by society and the marketplace is so great that a diversity of designs might satisfy a given requirement, and the task of producing any complete design is extremely complex. How, starting with a "blank slate" of silicon and access to computer-aided-design and programming tools, does someone create a system to play chess, to process radio-telescope signals into images, or to handle financial transactions in a bank? The complexity is managed by applying the abstraction principles described above in "Abstractions in Computer Systems."

The foundation of nearly all computing systems today is microelectronics. Even within the design of a microelectronic chip, the task is organized around such specialties as circuit design, logic design, system organization, testing, and verification, and each of these specialties must deal with physical design and layout together with logical behavior. Design techniques similar to those employed for chips are applied at higher levels to assemble aggregates of chips on circuit boards, and to provide interfaces to disks, displays, communication networks, and other electronic or electromechanical systems.

Microelectronics technology has blurred the boundary between hardware and software. Programs written in specialized notations are commonly built into chips. Systems designed for a single pur-

pose may, for example, incorporate application software that is compiled into such circuit structures as programmed logic arrays or into microprograms that reside in read-only memories. Computing systems also employ built-in programs to provide for maintenance, initialization, and start-up.

General-purpose computing systems require additional layers of software to provide the standardized system services needed to execute a variety of applications. The operating system allocates resources to run multiple programs, handles input and output devices, maintains the file system, and supports interprocess and network communications. Application programs may also require packages that provide standard interfaces for windows, graphics, databases, or computation, whether local or remote.

The remarkable advances in the performance, performance-cost ratio, and programmability of computing systems have been the combined result of achievements of CS&E within each of these layers. Although these advances have built on improvements in the base technologies, computing scientists and engineers have exploited these improvements extremely rapidly. The following subsections describe achievements within these layers: microelectronics; the organization of processors, memories, and communication; operating systems; computer networks; and database systems.

Microelectronics

The history of microelectronics is one of increasing the density of circuitry on a chip; the smaller the transistors and wires within a chip, the faster they can work, and the less energy they require. The process by which a microelectronic chip is fabricated is independent of the particular circuitry that will ultimately reside on the chip. The chip designer creates information-processing, memory, and communication structures within the chip by defining geometric patterns on a set of 10 to 15 photomasks; these patterns define the transistors and wires that will be embedded in the chip. Some of these wires lead to connections external to the chip that allow it to be hooked up to other chips and components.

In the 1960s, early integrated circuits (i.e., small-scale- and medium-scale-integration (SSI and MSI) chips) implemented electronic gates, adders, selectors, decoders, register memories, and other modules that had previously required circuit boards of discrete transistors and other electronic components. Photomasks for SSI and MSI chips were simple enough that they could be cut by hand from large sheets of plastic film before being photographically reduced.

The large-scale-integration (LSI) chips of the 1970s contained thousands to tens of thousands of components and included the dynamic random-access memory (DRAM) chips that eventually displaced the magnetic-core memory, and the first single-chip processors. These LSI chips were too complex for their photomasks to be created without computer-aided-design (CAD) tools. However, the CAD tools of that era served principally as drawing aids, a computerized extension of the drafting board. They did not otherwise assist designers in managing the complexity of their creations.

By the late 1970s, it was widely recognized that continued advances in microelectronics depended at least as much on streamlining the design processes and on new system ideas as on improving the fabrication processes. The time required to design then-sophisticated chips had already grown from weeks to years. Few people were prepared to address the functions (other than memory) that more complex chips would serve. One of the great achievements of CS&E was to show how to design the now-current generation of very-large-scale-integration (VLSI) chips.

Many advances in chip design resulted from the application of lessons learned from managing the complexity of large programs. Structured design disciplines based on a layout-cell hierarchy encouraged designers to compose layouts flexibly and systematically, to focus on improvements at the high levels of algorithms and overall organization, and to abstain from certain low-level optimizations (analogous to "goto" statements in programming) that are known to be more troublesome than useful.

These structured design approaches achieved significant successes even when applied using only computer-aided layout tools; these achievements might be compared with writing structured programs in assembly language. The next step was to incorporate these disciplines into high-level tools called silicon compilers. A silicon compiler includes a "front end" that translates a behavioral description for a chip into an intermediate structural representation, and a "back end" that translates the intermediate representation into layout that is tailored to the geometrical and electrical design rules of a particular fabrication process. Designs produced in this way can accordingly be re-created easily for different fabrication processes. Today, many chips are designed entirely by silicon compilation; nearly all complex chips employ these procedural approaches for creating layouts of some of their major cells.

Of course, chip designs may contain errors. To reduce the cost and effort required to debug chip designs, tools for simulation and verification are essential to the chip designer. The development of

algorithms and programs for multilevel simulation, including where necessary simulation down to the transistor-switch level, is the principal reason that today's million-transistor chips more often than not function correctly the first time they are fabricated. Symbolic verification that a chip's behavior will conform to its specification is not yet universal, but these techniques have been employed, for example, to demonstrate that a floating-point element within a chip will behave identically to software routines previously used for the same functions, and to verify the logical design of processors of moderate complexity.

The net result of these advances in design disciplines and tools is that today's state-of-the-art chips, although they approach being 100 times as complex as those of a decade ago, require a comparable design effort. Whereas a decade ago VLSI-design courses were offered in only a few universities, today they are offered in approximately 200 colleges and universities and have supplied the increasing demand for chip designers. Students in these courses use modern design and simulation tools to produce projects in a single term that are substantially more complex than state-of-the-art chips of a decade ago.

Processor and Memory Design

In recent years, it has become clear on theoretical grounds that for computations performed in VLSI chips and other high-performance digital technologies that press against the physical limits of intrachip communication, it is less costly to perform many operations at once (in parallel or concurrently) than it is to perform an operation correspondingly faster; see Box 6.4. For example, performing a given operation 10 times faster on a chip of a given family of designs would require a chip 100 times larger. However, by replicating the original chip only 10 times and connecting the chips in parallel, the same speedup factor of 10 could be obtained.[7] Much of high-speed computer architecture (both on the market now and to be available in the future) can be understood today as an endeavor to increase performance with parallelism and concurrency, which results in only a proportional increase in area, and to avoid brute-force attacks on serial speed, which leads into a realm of diminishing returns in performance for chip area.

Based on this complexity model and theory, algorithms that approach making optimal use of limited communication resources have been devised for such common operations as arithmetic, sorting, Fourier transforms, matrix computations, and digital signal processing. Analyses

BOX 6.4 LIMITS ON CHIP DESIGN

Early complexity theory originally emphasized the number of operations and the storage required to solve a given problem, since these parameters were the primary cost drivers for solving a problem. But most of a microelectronic chip is devoted to wire-based intrachip communications between its various elements. A specific complexity theory was therefore developed for very-large-scale-integration (VLSI) circuitry that accounted for the cost and time of communication in parallel, concurrent, and distributed algorithms running on a VLSI chip.

VLSI complexity theory relates the chip's area A and the time T it requires to perform a given operation. Optimal chip designs were shown to have the property that for any chip, AT^2 is equal to a constant that depends only on the base technology and the nature of the operation. For example, for multiplication of n-bit binary integers, optimal chip designs have the property that AT^2 is greater than or equal to kn^2, where the coefficient k incorporates characteristics of the target technology and problem. Thus, within the class of optimal designs, it is possible to trade off area (e.g., a larger chip) and time (e.g., faster operation).

of such algorithms account for the patterns of data movement, often in regular topologies such as hypercubes, meshes, and trees. These algorithms have been applied both to the internal design of VLSI chips and to the design of special-purpose systems such as digital signal processors, and to the programming of parallel and concurrent computers. The analyses of communication topologies have, in addition, been important to the design of the communication networks used in programmable parallel and concurrent computers.

VLSI technologies led to a renaissance in many computer architectures during the 1980s. An example is the emergence of reduced-instruction-set computers (RISCs), which provide a good illustration of why single-chip processor performance has advanced more rapidly than might have been expected simply from technology gains. As we have seen, VLSI technology favors concurrency. RISCs exploit concurrency by employing a pipeline structure that can overlap the execution of several instructions. Dependencies between sequential instructions can defeat this approach, but RISCs employ compilers to analyze dependencies and to generate object code (i.e., machine instructions) to schedule the pipeline in advance.

High-speed processors also place severe demands on the memory system. Again, the solutions include parallelism and concurrency: wider data paths to convey more bits in parallel, separate memories for instructions and data, and multiple banks of memories. These innovations were first applied to the supercomputers of the 1960s, and they are applicable to single-chip design issues as well. Fast memories require more area per bit than do slow memories; it is accordingly not economical to implement the primary memory of a computing system entirely from fast memories. A more sensible organization employs a small, fast memory called a cache to store frequently accessed items. If the processor makes reference mostly to items that are stored in the cache, memory operations can be speeded up considerably. Of course, the cache memory must have a way of determining what items will be most frequently accessed. Analysis of cache sizes, organizations, and replacement strategies has been an active and productive area of research and development. Just as advances in processor design have depended on compiler technology, so also is cache design starting to benefit from compiler analysis and optimization of the patterns of memory accesses.

Operating Systems

One of the important, practical ideas that led to modern operating systems was timesharing, invented in the early 1960s. Operating systems are part of the system software that provides frequently needed functions to the user. The earliest operating systems limited the computer to running one user program at a time: these systems could not start the next program until the preceding one had terminated. A timesharing operating system interleaves execution over short intervals between several executable programs available in the machine. This interleaving is equivalent to executing a number of programs concurrently on a number of somewhat slower machines. Before the invention of timesharing, interactive tasks required a computer dedicated to that task, whereas with timesharing, a large number of users can tap into a single machine that provides bursts of computation on demand.

The introduction of hardware address-translation, memory-protection, and system-call mechanisms under the control of a multiprogramming operating system made timesharing a good way to increase the efficiency of a machine's use by allowing multiple processes to reside in memory at the same time, and provided processes with virtual memory. A "process" is an instance of a program during its execution, including both the program's instructions and data. Processes are the basic schedulable units of activity or execution for

nearly everything an operating system does. Execution of a user program, updating a graphics window, transferring data from or to an open file, and controlling the operation of an input-output device can be regarded as processes that execute concurrently. "Virtual memory" refers to the ability of a process to refer to a larger range of memory addresses than may be present in primary memory at any one time.

In addition to their demand for execution cycles, processes require memory resources. The physical memory of a computer is limited and contains layers of increasing size and access time: Machine registers are few but very fast, primary memory is intermediate, and disks store large volumes of data but with large access times. The greatest disparity in access time, hence, the most critical choice for memory allocation, occurs between the disks and the primary memory. Should programs explicitly manage the memory actually available by bringing data and subroutines from disk on demand, or can the operating system perform these functions? Theory and practice have shown that programs do not need to be complicated with the extra task of memory management; indeed, operating systems that manage memory for many processes at once can do a better job overall than is possible when applications programs handle memory management individually.

Paging is one technique operating systems use to provide virtual memory. In common with other caching strategies, paging is based on the likelihood that the instruction and data words accessed are the same as or located close to previously accessed words. A paging strategy partitions memory into frames of, for example, four kilobytes each. A frame can hold a page of information that can be moved between disk and primary memory. Put simply, the operating system's goal is to make it likely that the pages involved in active computations are available in the primary memory, and that inactive pages remain on disk. Accessing a page that is not in the primary memory is called a page fault. The operating system must then intervene to move the needed page from disk to primary memory.

What happens when the primary memory is full with active pages, and another page must be loaded? One algorithm is to replace the least recently used (LRU) page, or any page not used recently. The operating system decides on the replacement based only on its record of the page-access behavior of the processes whose pages it manages, and not on information about their internal structure. The LRU algorithm generally performs well in an environment of unrelated concurrent processes for the same reason that paging works: programs tend to access memory locations that are local to recently accessed locations, a characteristic referred to as locality of reference.

Processes may need to communicate with each other. That need existed 30 years ago when time-sharing was applied to single machines but has become increasingly necessary now that processes that are parts of the same computation are commonly distributed across networks of computers.

Applications that employ communicating concurrent processes may require certain actions to be performed by no more than one process at a time, or to be done in a particular order. Suppose, for example, that processes *X* and *Y* share the use of a file that contains a record of your bank balance, and that both processes attempt to record deposits. Processes *X* and *Y* can each have read the initial balance, added their own deposit, and written the new balance back into the file.

If the processes are interleaved so that the two read operations precede the two write operations, one of the deposits will be lost when one process overwrites the result of the other. What is necessary to prevent this error is to lock the file against other accesses when the first process requests the balance, and to unlock the file after its deposit transaction is completed. Problems with the order of execution may arise in situations where one process cannot proceed because another process has not acted. For instance, if process *X* reserves airline seats and process *Y* records cancellations, *X* must wait for *Y* whenever all seats for a flight are booked.

Computer scientists and engineers invented synchronization mechanisms to deal with these problems long before they became common in practice. The first such mechanism employed operations that are analogous to the way semaphores protect railroad sections against conflicting train traffic. A critical section of program that must not be executed by more than one process can be entered only when the semaphore is green, and entering has the side-effect of turning the semaphore red. The semaphore is set back to green when the process that entered has completed the execution of the critical code.

When the semaphore has been set back to green, which of several waiting processes will be allowed to enter the critical section? A priority queue that records the processes waiting for the critical section can ensure fairness, so that all waiting processes will eventually be served. Once these properties of logical correctness and fairness are assured, what is the performance of these solutions? Queuing models have been devised that lead to solutions that are optimal depending on the particular characteristics of the concurrent processes.

A nasty problem not solved by synchronization mechanisms is deadlock. A deadlock arises due to mutual dependencies between processes, such as process *X* using resource *A* and requesting the use of additional resource *B* without giving up *A*, and process *Y* using resource *B* and requesting the use of *A* without giving up *B*. When these processes fail to make progress, all services dependent on them cease to function, eventually halting the entire system. (This is one reason that the distributed systems used by banks and airlines sometimes cease to function.) Proving that a system of concurrent processes will operate correctly and will not deadlock is generally harder than proving the correctness of a single sequential program. Reasoning about concurrent processes has been aided by temporal logic, a first-order predicate logic with the addition of temporal notions such as "forever" and "eventually."

In common with other system-building disciplines, the design of operating systems illustrates how progress has benefitted from theoretical work, has been stimulated by practical ideas, and has been responsive to changes in technologies and needs.

Data Communications and Networking

Computer technology is becoming increasingly decentralized; the personal computers on office desks are an example of this trend. Similarly, centralized corporate control is passing down the hierarchy, and in many cases the hierarchy itself is flattening. To provide connectivity among these distributed parts, computer networks have proliferated at an amazing speed in business, government, and academia. In fact, perhaps the only remaining aspect of information technology that remains in the hands of central management is the network itself.

The first successful large-scale computer network was the experimental ARPANET, first deployed in 1969 to provide terminal-to-computer and computer-to-computer connectivity to major CS&E departments in the nation. ARPANET was based on packet-switching technology (Box 6.5). The network itself consisted of many switches, each connected to at least one and usually several others so that there were multiple paths between any two. The communication lines connecting the switches were not used exclusively for any particular end-to-end communication. Rather, each line was used by a particular message only when one of its packets happened to be transmitted over that line.

Packet switching provided high-speed, cost-effective connectivity between attached devices across long distances by dynamically sharing the expensive bandwidth of high-speed lines among many users. A major benefit of packet switching is fault tolerance—when a node or line fails, other nodes can simply bypass it.

BOX 6.5 ABOUT PACKET SWITCHING

Packet switching, which has been made possible by high-speed computers, involves two separate concepts. The first concept is that a message can be broken into discrete parts, or packets, at the originating station, sent in pieces through a network of switches, and reassembled at its destination. "Packetizing" a message breaks a big message into many small ones, which in turn makes it easier to process.

The second concept is demand multiplexing, a way to keep a single (expensive) circuit in relatively continuous use. This notion is not new in principle: transoceanic voice communications have for many years been based on the routing of different calls through the same circuit on a time-shared basis. In the ARPANET packet-switching context, each switch routes incoming packets to adjacent switches based on the packet's ultimate destination and based on line congestion. The goal is to minimize congestion and the number of hops a message must travel.

Thus the path of an individual packet through a packet-switched network is determined dynamically, and the path of one packet may or may not duplicate the path of another packet, even though each is part of the same message. When a switch receives a packet, it decides where next to send the packet depending on its ultimate destination and on the available capacity of communications lines between itself and the switches to which it is connected.

ARPANET had substantial impact on architectures for commercial networks such as SNA and DECnet. It also spurred a great deal of research on data communications and resulted in the TCP/IP network protocol, now widely used to connect heterogeneous computer networks.

While nationwide packet networks such as the ARPANET met the needs of the 1970s, a new requirement arose from the proliferation of personal computers in the 1980s. Personal computers presented a rapidly rising demand for connectivity among themselves, peripherals, and servers in a local office environment. The CS&E research community anticipated this need in the 1970s and pioneered the hardware and software technologies for local area networks (LANs). Current LAN technologies grew out of small research investments at Xerox (Ethernet) and Cambridge (token ring). The growth of LANs paralleled that of personal computers in the 1980s. LANs provided

relatively inexpensive connectivity in an office environment, and did so at multi-megabit-per-second speeds.

Today, LANs are being interconnected to form larger networks. Meanwhile, the bandwidths of wide area networks (WANs) have evolved from 64 kilobits per second to the widespread availability of economically tariffed T1 service at 1.5 megabits per second (mbps). Thus WAN speeds are approaching the speeds of many LANs deployed today, though LANs are getting much faster, too. Recent needs for faster metropolitan area network (MAN) speeds have been met with the acceptance of the Fiber Distributed Data Interface (FDDI) standard for MANs. This is a 100-mbps offering based on fiber-optic media.

LANs, MANs, and WANs were initially designed to meet data communications problems based on copper technology. The emergence of fiber-optic technology and its consequent 1000-fold increase in bandwidth completely changes the technology trade-offs and requires reconceptualization of network designs. This subject is the focus of effort currently under way in the National Research and Education Network component of the High Performance Computing and Communications Program, discussed in Chapter 1.

Database Systems

Modern database technology has been shaped from top to bottom by CS&E research. Computer science researchers in the 1960s created the relational data model to represent data in a simple way. Computer engineers worked through the 1970s on techniques to implement this model. By the mid-1980s these ideas were understood well enough to be standardized by the International Organization for Standardization (ISO) as the SQL language. Today, the SQL language has become the lingua franca of the database business, for reasons described below.

Computerized databases began in the late 1950s, each built as a special application. By the late 1960s, the network data model had emerged as a generalization of the way these systems stored and accessed data. The network model is a low-level approach that represents and manipulates data one record at a time.

However, despite its undeniable utility at the time, the network data model was troubling, because it lacked a firm mathematical foundation. Indeed, database systems at the time were ad hoc and seemed to behave in random ways, especially in unusual cases. Database researchers needed a good theory with which to predict the behavior of databases and a data manipulation language with clear properties, reasoning that these tools would make it easier to program database

applications that could be insulated from changes to the database as the database evolved over the decades.

The relational data model grew out of this research effort. It is a high-level, set-oriented, automatic approach to representing data. The data are structured in tabular form, making them easier to visualize and display. Database queries and manipulations are based on pure mathematics (sets and relations) and the operators on them. As a result, the model is simpler to use, and gives applications greater independence from changes in technology and changes in the database schema. Early studies showed it to improve programmer productivity by large factors.

The relational model was initially rejected by database implementors and users because it was so inefficient. All agreed that it boosted productivity, but there was skepticism that the nonprocedural language could ever be efficiently implemented.

Research computer engineers in academia and industry took on the challenge of efficiently implementing the relational model. The goal was to provide the simplicity and nonprocedural access of the relational model and performance competitive with the best network database systems. This effort required the invention of many new concepts and algorithms.

This research took about ten years. In the end, the relational system's performance began to meet and even exceed the performance of systems using the network data model. With this feasibility demonstration, the slow transition from network to relational systems began. First, computer vendors began offering database systems based on the research prototypes. Then the SQL international standard was approved, and customers began to use the relational approach. Today, 25 years after the first research papers appeared, the transition to SQL databases is in full swing.

Without the early seed work by computer scientists, the relational model would not have been invented. Without the early seed work by computer engineers, the implementation feasibility would not have been demonstrated. And without these two advances, we would likely still be programming databases in low-level programming languages.

Based on the relational model, the database community spent much of the 1980s investigating distributed and heterogeneous database systems. Those ideas are now well understood, and the ISO is about to approve a standard way for databases to interoperate—the Remote Database Access standard. This standard promises to allow diverse computers to interchange information and to allow database queries and transactions to span multiple database systems; see Box 6.6 for more discussion.

BOX 6.6 RELIABLE TRANSACTION SYSTEMS

Some of the CS&E work on transaction systems addresses issues that arise in databases, distributed computing, and operating systems.

Consider the problem of a distributed computation running on many different computers and databases. If one computer or process fails, what are the others to do? For example, one bank may attempt an electronic transfer of money to another bank. If the receiving bank's computer fails while the transfer from the sending bank is in progress, then the money may disappear, because the sending bank will have debited the right account but the receiving bank will not have been able to credit the right account (since it never received the message).

Transaction concepts and techniques are designed to simplify the design and implementation of operating systems for such distributed applications. The transaction concept requires that a change in the state of a system involved in a transaction should have the ACID properties: the change should be all or nothing (*A*tomic), result in a correct state transformation (*C*onsistent), be free of concurrency anomalies (*I*solated), and generate transaction outputs that are not lost in case of system failure (*D*urable). If anything goes wrong before the entire transaction is complete, the use of transaction techniques will ensure either that all changes are undone (so that the databases of sender and receiver are restored to the state that existed before the transaction began) or that changes to databases will endure and also that all output messages will be delivered.

Most current database research activity focuses on adding more semantics to the relational model and to the transaction model. This work, operating under the banner of object-oriented databases, is still in its early stages. In addition, consistent with the application-oriented trend of computer science in general, there is considerable interest in databases designed specifically for certain problems: geographic databases, text databases, image databases, and scientific databases.

Programming Languages, Compilers, and Software Engineering

Programming Languages

Advances in science often lead to the invention of notations to express new concepts or to simplify and unify old ones. The advent of computers spurred the invention of programming languages, a

BOX 6.7 PROGRAMMING LANGUAGES OF SIGNIFICANCE

C allows the programmer to control details of machine operation in a manner that depends only on general features common to most machines and not on the idiosyncracies of individual machines.

Lisp concentrates on the manipulation of symbolic—as distinct from arithmetic—information. Lisp is used widely for symbolic computation, the representation of cognitive processes, and in systems that can reason about and manipulate computer programs.

Smalltalk was the first "object-oriented" programming language integrated with a graphical environment, and was based on "active" data objects (i.e., objects with some executable code associated with them) coupled together by simple interfaces.

Prolog allows a program to be written by specifying logical relations that have to be satisfied by the solution of the problem. The "logic programming" paradigm is now spreading into database practice.

Visicalc was the first spreadsheet for personal computers. It simulated business ledgers and reports, automatically performing the necessary recalculating when any change was made to a table entry.

ML provides the ability to separate the definitions of data types from the manipulations to be performed on them.

TEX is a high-level typesetting and formatting language, often used for technical manuscripts, and is the basis for electronic submission of papers to journals of the American Mathematical Society.

PostScript is a language developed for use by laser printers rather than people; its purpose is to express the detailed layout of pages of text and pictures, providing the link between typesetting programs and printers of different manufacture.

new kind of notation for expressing algorithms in terms more appropriate to human discourse, while nevertheless meeting the exacting mechanical needs of computers. The development of programming languages introduced new challenges. First, more formality and greater detail were required; because of processing by computer, nothing could be left for interpretation by humans. Second, new concepts

had to be developed to deal with execution of an algorithm; previously, mathematics dealt primarily with a more static world. Third, a conflict between transparency of notation and efficiency of execution was introduced; programs have to be reasonably efficient as well as perspicuous.

Since the dawn of the computer age, many developments in the area of programming language design have emerged, developments that are reflected in a succession of programming languages that make it easier to express algorithms precisely in various problem domains. Many are familiar with mainstream languages descended from Fortran via Algol and Cobol through countless variants such as Basic and Pascal. However, though the bulk of programming today is done in these languages, there are countless alternate styles, as noted in Box 6.7. Without the development of the concepts that have been embedded in these languages as well as the languages themselves, the information age would not be so advanced.

Compilers

Compilers provide the essential link between programming languages (i.e., the readable, higher-level, problem-oriented text written by human beings) to the low-level machine encodings that actually govern the operation of a computer. Advancements in compiler technology have been remarkable in the past few decades. For example, the first Fortran compiler took dozens of person-years to build with what were essentially ad hoc techniques. Today, the same effort is required to develop compilers that handle languages of much greater complexity and sophistication. This vast increase in productivity is due partly to better programming environments and more powerful computers, but most significantly to a sound theoretical understanding of compiler techniques that allow a high degree of automation for much of the compiler writing process. Several waves of theory have contributed to compiler technology: formal languages, semantic analysis, and code optimization and generation.

Formal language theory provided the understanding of grammar upon which the mechanical parsing in every modern compiler is built, and that shaped the very form of programming languages. The theory provides the mathematical basis for engineering the front ends of compilers, where the structure of a program is extracted from a program text. The subject of parsers is now so well understood that their construction has been highly automated. From a precise definition of the syntax of a programming language a parser can be generated in a very short time.

Formal theories of semantic analysis gave accurate descriptions of the meaning of programming languages, which has led to more predictable languages and in some commercial instances to automatic generation of compilers from language specifications. In concert with type theory, formal semantics have enabled compiler writers to implement languages that support these advanced modes of expression in a clean and rigorous manner. One result has been that polymorphic programming systems—programming systems that can support the design of algorithms that work independently of the type of input data—can now be transported from one computing environment to another with a minimum of difficulty.

Techniques for analyzing programs have been developed that enable optimizing compilers to speed up the execution of programs substantially. An important approach is called data-flow analysis, a method for determining where values once computed in the program can possibly be used later in the program. This information is now routinely applied in optimizations such as register allocation, common subexpression elimination, and strength reduction in loops. Register allocation attempts to keep the most frequently used quantities in the most accessible storage. Common subexpression elimination attempts to identify and obviate the need to recalculate equal quantities at different points of a program. Strength reduction in loops amounts to identifying instances of repeated evaluation of polynomial expressions that can instead be updated by finite differences, thus simplifying the computation.

More elaborate optimization is required to compile sequentially written programs to make effective use of parallel hardware architectures. Although good parallelizing Fortran compilers exist for particular computers, this technology is still far from routine.

Software Engineering

Software engineering refers to the construction of software systems. While there is some controversy in CS&E regarding the extent to which the "engineering" in software engineering is truly science-based engineering (as opposed to management), it is clear that large software systems (in excess of several million lines) can and have been built. Without many of the software engineering tools that have been developed in the last 30 years, the construction of large software systems would not be possible. These tools have been developed in response to pragmatic needs, but they may well incorporate some of the products of research. Two generic types of software engineering tools are described below. In addition, software reuse,

an issue of generic concern to software engineers, and real-time computing, a computing application of great engineering importance to society, are discussed.

Project Control Systems The development of large-scale software systems is particularly problematic. A large-scale software system may have millions of lines of code. Writing such a system is obviously a many-person enterprise, and the efforts of all of these programmers need to be coordinated efficiently. Project control systems enable the construction of large systems by automating coordination processes that would otherwise be subject to human error.

For example, large systems are written in modular form, to facilitate debugging. Modules must be relatively small (hundreds of lines of code), so that the programmer can understand the details of the module. Any given module may exist in several different versions, perhaps an early version that is known to work properly but implements only basic functions and a later one in progress that is being enhanced. To assemble the large system (e.g., for system-level testing), it is necessary to gather together the working versions of each module, taking into account the fact that a change in module A may mean that an earlier version of module B must be included. Assembly can be performed manually, but with tens of thousands of modules to be gathered, it is inevitable that mistakes will be made. Automated configuration management systems keep track of a myriad of bookkeeping details, enabling such assembly to proceed with far fewer errors.

Source Code Control Systems The source code of a software system is written by human programmers. When source code becomes voluminous and involves a team effort of many programmers, management is often difficult. Source code control systems automate many management-related tasks. For example, such systems can ensure that only one person is making changes to any given module at any time, and that all changes are recorded (usually through an automated comparison of the new file to the old version) so that there is a trail of implementation responsibility comparable to the change history found on engineering drawings.

"Discovery tools" are an important adjunct to source code control. These tools provide for navigation within a large library of software so that developers and maintainers can efficiently answer questions such as, What modules touch this variable? What does that subroutine do? What is the data layout of that table? Do two selected actions always happen in order? By what routes through the program can control get to this point? Such tools help programmers understand better the code on which they work.

As an aside, it is interesting to note that the algorithm used to compare files in the source code control system of Unix came from research; the same is true for the flow analysis tools used in many discovery tools.

Software Reuse An elusive grail of CS&E has been the reuse of software parts. Just as one constructs electrical appliances or houses largely from off-the-shelf parts, so would one also like to build software in a similar fashion. Because software is the dominant cost of most systems, developing ways to reuse software is a growing concern and is being tackled in a variety of ways.

In some domains, useful libraries of software components have long existed. Particularly well known are the libraries that provide efficient implementations of common mathematical constructs, such as trigonometric functions or matrix operations. Major packages of routines for statistics and other fields of mathematics and engineering are also available. Programming tools, as exemplified by Unix, foster the construction of systems out of many small and independent generic components coupled by a "shell" language.

By generalizing from specific instances, the domain of applicability of code may be widened. For example, the notions of polymorphism and modularity found in Ada and ML (and other languages) make it possible to write portions of programs that are applicable to more kinds of data, thus reducing the need to write highly similar portions of code that differ primarily with respect to the kind of data being processed.

Object-oriented programming provides a different way to structure programs to increase abstraction and reduce the need for rewriting. In object-oriented programming, the basic procedures that operate on an object are encapsulated with that object. For example, an object called POLYGON can have associated with it not only a description of its sides but also a procedure for calculating its area. A second important feature of object-oriented programming is that an object can "inherit" procedures of a higher-level object but can also replace some of them with new procedures. For example, having already defined a POLYGON, one can then define a SQUARE as a special instance of POLYGON but with a more efficient procedure for calculating its area. A program that manipulates polygons or squares would not include code to calculate area, since the object itself would provide that code. Object-oriented programming provides a new flexibility in structuring programs, leading to easier development and maintenance of large programs and programming systems.

Despite such developments, reuse of software is not common, and much more remains to be done in this area.

Real-Time Computing Real-time computing refers to a mode of computing in which computing tasks must be accomplished within certain time limits. For example, computers controlling the operation of an airplane's engines and flaps are computing in real time; failure to meet the relevant computing deadlines could easily lead to disaster.

When a processor for a real-time computer system must handle several tasks simultaneously, how much time to give to each task and in what order they should be performed are crucial considerations for the real-time programmer: sometimes a lower-priority task must be preempted by a higher-priority one, even though both must be completed on time (because the higher-priority task may depend on a result computed by the lower-priority one). Such decisions are simple to make when the number of tasks is small and the amount of available computational power is large relative to the demands of the tasks taken in the aggregate. But as a matter of engineering practicality, the designer of an airplane or a missile does not have unlimited freedom to choose processors that are greatly overmatched to the computing tasks involved.

The traditional method of scheduling concurrent tasks is to lay out an execution time line manually.[8] Although for simple cases it is relatively easy to determine task execution sequences, the resulting program structure is hard to change. For example, the specification of a task may be altered so that it demands more computational resources; accommodating such a change might well require redoing the entire time line. The scheduling must accommodate the differing deadlines, demands on resources, and priorities of the various tasks to ensure that all tasks are completed on time.

However, by taking advantage of certain features of a relatively new programming language, Ada, it is possible, under certain circumstances, to implement conveniently an algorithm ensuring that all tasks will be completed on time without knowing the precise details of the execution sequence. This algorithm—the rate monotonic scheduling algorithm—also enables the convenient accommodation of changes in task specification without re-doing the entire time line, and is starting to come into use in some real-time aerospace applications.

Algorithms and Computational Complexity

As noted above in the section "Computer Science and Engineering," algorithms as an intellectual theme pervade all of CS&E. However, the formal study of algorithms is an important subarea of CS&E research in its own right (but see Box 6.8). The design and analysis

BOX 6.8 A NOTE ON TERMINOLOGY

The term "theory" as used within the CS&E community is often construed quite narrowly, as exemplified by work presented at conferences such as the ACM Symposium on Theory of Computing (STOC) and the IEEE Computer Society's Symposium on Foundations of Computer Science (FOCS).

However, the committee believes that a narrow view of theory is excessively limiting. Indeed, it believes that "theory in CS&E," "theoretical computer science," and "theoretical work in CS&E" should in fact refer to *all non-experimental work in CS&E intended to build mathematical foundations and models for describing, explaining and understanding various aspects of computing.* Such work includes research performed in the FOCS-STOC community, such as the theory of computation, study of formal models of computation, and computational complexity, but is not limited to these areas. It also includes the analysis and synthesis of algorithms, the study of formal languages, the syntax and semantics of programming languages, compiling techniques and code optimization, principles of operating systems, various logics for reasoning about programs and computations, fundamentals of databases, expert systems, knowledge representation, mechanical theorem proving, heuristic search, principles of computer architecture, VLSI design, parallel and distributed computing, and the rich theory of numerical analysis and scientific computation.

In the interests of clarity and in deference to traditional usage, the committee has generally used the term "theoretical work" in this report to refer to this broader notion of theory. Further, it believes that the theoretical CS&E community will be enriched by an expansive vision of theory in CS&E.

of algorithms combine intellectual and theoretical challenges with the satisfaction of computational experimentation and practical results.[9]

Algorithms Everywhere

One domain in which better algorithms are enormously important is science and engineering research. Chapter 1 noted that the solution of certain partial differential equations has been speeded up by a factor of about 10^{11} since 1945. A large part of this speedup is due to the development of faster algorithms, as outlined in Table 6.1.

TABLE 6.1 Algorithmic Improvements in Solving Elliptical Partial Differential Equations in Three Dimensions

Method	Year	Run Time[a]	Time[b]
Successive over-relaxation (suboptimal)	1954	$8\ N^5$	250 years
Successive over-relaxation (optimal)	1960	$8\ N^4 \log_2 N$	2.5 years
Cyclic reduction	1970	$8\ N^3 \log_2 N$	22 hours
Multigrid	1978	$60\ N^3$	17 hours

[a] N is the number of grid points in one linear dimension.

[b] Time required for solving an equation with $N = 1000$, assuming a time of 10^{-6} seconds for processing each of the 1000^3 points.

SOURCE: Adapted from John Rice, *Numerical Methods, Software, and Analysis*, McGraw-Hill, 1983, p. 343.

Box 6.9 describes an algorithmic advance that made feasible a computationally complex calculation for a member of the committee.

In many problems faced by industry and commerce, other types of algorithms are often relevant, including linear programming (LP) algorithms and algorithms to solve the traveling salesman problem.

An LP problem is one that requires the maximization or minimization of some output subject to a set of constraints on the inputs. For example, a manufacturing firm makes two different products, widgets and gizmos. The sale of each product results in a different profit; the manufacture of each product requires a different combination of engineering, inspection, and packaging attention as it moves through the shop. The total amount of attention that can be given every day is fixed by union regulation. What is the optimal combination of widgets and gizmos for the firm to manufacture so that it can maximize its profits?

Problems of this general form arise in all walks of life. Airlines use highly sophisticated LP algorithms to schedule equipment and flight crews.[10] Equipment scheduling seeks to optimize the use of available aircraft and maintenance facilities by assigning aircraft to each route to meet the requirements of inspection and maintenance at the right ground facilities, with the needed personnel, and with the fewest delays. Crew scheduling seeks to optimize the use of personnel and layover facilities by matching the available crew qualifications and home bases with equipment demands, while respecting government and other work rules, such as those governing time between flights and travel time between assignments. Airline scheduling problems may involve millions of variables. Until recently, the solutions could only be guessed at.

BOX 6.9 BETTER ALGORITHMS HELP THE EARTH SCIENCES

In the late 1970s, a member of this committee (Jeff Dozier) began to examine the problem of calculation of the surface energy balance over mountainous terrain. Different patches of terrain receive different amounts of sunlight, depending on the position of the sun in the sky and the shading on nearby slopes caused by mountain peaks. The latter calculation is computationally complex, because the calculation at a given point must take into account shading that might arise from all other points in the region of interest.

The initial solution to this problem in 1979 calculated the tangent of the angle from a particular point to all other points in a specified arc. The maximum tangent is the horizon angle. This solution requires a computing time proportional to N^2, since each point in the terrain grid requires comparison to the other $N-1$ points.

In 1980, John Bruno (Computer Science Department, University of California at Santa Barbara) and Dozier began to discuss this problem. Bruno and Peter Downey (Computer Science Department, University of Arizona) formulated a much more elegant solution that eliminated comparison to most of the other points in the grid and therefore ran in a time proportional to N rather than N^2. Computations not previously feasible became feasible.

SOURCE: Jeff Dozier, University of California at Santa Barbara.

The chemical industry also uses sophisticated LP algorithms to compute, for example, the least expensive mix for blending various distillates of crude oil to make the desired products.

Another scheduling problem that arises in many practical applications is best understood in terms of the traveling salesman problem (TSP): given a set of cities and the distances between them, find the shortest tour for a salesman to visit all cites.

The TSP arises in planning deliveries and service calls. Less obviously, it appears in many different contexts in science, engineering, and manufacturing. For example, in crystallography with high-energy and high-density x rays, the irradiation time is small compared to the time needed for the motors to change the orientation of the sample and/or the detector's position. For an experiment with some 14,000 readings, minimizing the repositioning time can cut the duration of a week-long experiment by 30 percent to 50 percent. In manufacturing electronic components, the drilling of holes in circuit boards

with an expensive laser drill puts a premium on minimizing the total "travel time" of the drill head (or board) between successive holes. For a complex circuit board, the shortest total route among as many as 17,000 holes may be required. The fabrication of a VLSI circuit may call for solving an analogous problem with over a million sites.

Faster algorithms can reduce significantly the time required to perform tasks on the computer, thus making industry more efficient, and can also increase dramatically the sizes of the problems that can be feasibly solved. An appropriate selection of algorithms can also have a profound effect on the conduct of science, again by reducing the time needed to solve certain problems from utterly unfeasible times to quite feasible times.

The Study of Algorithms

In the 1930s, before the advent of electronic computers, Church, Kleene, and Turing laid the foundations for the theory of computation and algorithms. Their work showed that there was essentially only one basic concept of effective computability—aside from matters of speed and capacity, a problem solvable on one computer is solvable on all.

Particularly relevant to the study of algorithms is the Turing machine, an abstract computational device that Turing used to investigate computability. The Turing machine is also a vehicle through which it can be shown that all computational problems have an intrinsic difficulty that theoretical computer scientists call complexity—an inescapable minimum degree of growth of solution time with problem size.[11] The solution time for a problem with high complexity grows very rapidly with problem size, whereas the solution time for a problem with low complexity grows slowly with problem size. Switching to a faster computer may improve the computation time by a constant factor, but it cannot reduce the rate of increase of computation time with the growth of the problem size.

Suppose we could prove that finding the optimal tour for TSPs with n cities requires on the order of 2^n steps in the worst case.[12] Then no computer could overcome the prohibitive exponential growth rate for the solution of the TSP. A factor-of-100 speedup over a machine that could barely solve 100-city problems would not increase the capacity enough to solve 110-city problems. (This would be true whether the faster computer were a parallel computer with 100 processors or a serial computer that operated 100 times as fast.)

A particular algorithm that solves a given problem may result in solution times that grow with problem size at a faster rate than one

would expect from the intrinsic complexity of the problem. Designing algorithms that run in times that are close to those implied by the intrinsic complexity of a problem becomes an important task, because the need to solve ever larger practical problems grows much faster than our ability to buy larger or faster computers.

Computational Complexity

The study of the complexity of computations, referred to as computational complexity theory, has revealed remarkably rich connections between problems. Good examples of complexity theory arise in the context of linear programming and traveling salesman scheduling problems.

If the solution time for a problem grows exponentially with problem size, then there is no hope of solving large instances of the problem, and it becomes necessary to look for approximate solutions or study more tractable subclasses of this problem. The question of minimal growth, or "lower bounds," requires a deep understanding of the problem, since all possible ways of computing a problem must be considered to determine the fastest one.

The study of LP algorithms provides an interesting illustration of how theory and practice interact. For many years, the simplex algorithm was the main way to solve LP problems. In 1972, it was shown that its worst-case running time grew exponentially with the size of the problem, although the algorithm worked well on practical problems.

In 1979 came the startling announcement of a new polynomial-time algorithm. Although this algorithm did not compete well with the highly developed simplex algorithms, the proof that polynomial time was possible galvanized the research community to activity. The new research yielded "interior-point" algorithms that today compete successfully with the simplex algorithm. A major airline now uses an interior-point algorithm for scheduling equipment and crews.

Although LP problems can be solved in polynomial time, it is widely believed that there is no polynomial-time algorithm for the TSP. However, in spite of at least 20 years of hard effort, there is no proof of the conjecture that the problems in the class denoted by NP are not computable in polynomial time. The class of NP problems, which contains many important combinatorial problems, is widely believed to be not computable in polynomial time. The TSP and hundreds of other problems are known to be "NP-complete," which means that a polynomial-time algorithm for any one of them will guarantee a polynomial-time algorithm for all. For example, from a polynomial-time algorithm for determining whether a set of jigsaw

pieces will tile a rectangle, it is possible to derive a polynomial-time algorithm for the TSP, and vice versa. The question of determining whether NP-complete problems have polynomial-time algorithms, the "P = NP" question, is one of the most notorious and important problems in computer science.

At present, there is no hope of solving exactly 10,000- to 1 million-city problems, which do arise in practical applications. Therefore researchers seek efficient algorithms that find good *approximations* to the optimal tour. The study of approximate methods is an interesting blend of theory and experiment to assess the performance of various approaches and to gain insight toward better algorithms. For some approximate algorithms, there exist theoretical guarantees of running time and accuracy. Trade-offs between time and accuracy have also been demonstrated. Good solutions have been obtained for practical problems with 10,000 to 1 million cities.

Besides its practical relevance, computational complexity provides beautiful examples of how the computing paradigm has permitted computer scientists to give precise formulations to relevant problems that previously could be discussed only in vague, nonscientific terms.

Consider, for example, a problem about the basic nature of mathematics. All our experience suggests that the creative act of finding a proof of a theorem is much harder than just checking a proof for correctness. However, until recently this problem could not be formulated in precise quantitative terms. Complexity theory allowed this problem to be made precise: how much harder is it *computationally* (for an algorithm) to find a proof of a theorem than to check its validity? The correctness of a properly formalized proof can be checked in polynomial time in the length of the proof. However, finding a proof of a given length can be done by "nondeterministically" guessing a proof and then checking in polynomial time whether it is a correct proof. Thus the finding of proofs of a given length is an NP problem. Furthermore, it is not hard to show that it is NP-complete.

Hence a startling conclusion emerges: finding proofs of length n is polynomially equivalent to solving n-city TSPs! Either both can be done in polynomial time (which is strongly doubted) or neither can.

Complexity theory seeks to understand the scope and limitations of computing. In doing so, it addresses questions about the basic nature of mathematics and the power of deductive reasoning.

Artificial Intelligence

Artificial intelligence (AI) is founded on the premise that most mental activity can be explained in terms of computation. AI's scien-

tific goal is to understand the principles and mechanisms that account for intelligent action (whether or not performed by a human being), and its engineering goal is to design artifacts based on these principles and mechanisms.

The premise of AI has a long tradition in Western philosophy. Aristotle and Plato believed that thought, like any other physical phenomenon, could be unraveled using scientific observation and logical inference. Leibniz equated thought with calculation, setting the stage for Boole's treatise on propositional logic, titled "The Laws of Thought." Much later, the advent of computers led Alan Turing to envision a new field, computing machinery and intelligence. The formal discipline of AI was inaugurated in 1956 at a Dartmouth conference by John McCarthy, Marvin Minsky, Allen Newell, and Herbert Simon.

As an empirical science, AI follows the classical hypothesis-and-test research paradigm. The computer is the laboratory where AI experiments are conducted. Design, construction, testing, and measurement of computer programs are the major elements of the process by which AI researchers validate models and mechanisms of intelligent action. The original research paradigm in AI involved the following steps:

- Identify a significant problem that everyone would agree requires "intelligence."
- Identify the information needed to produce a solution (conceptualization).
- Determine an appropriate computer representation for this information (knowledge representation).
- Develop an algorithm that manipulates the representation to solve the problem (e.g., heuristic search, automated reasoning).
- Write a program that implements the algorithm and experiment with it.

A few subareas of AI are examined below from two points of view: their impact on society (including their economic impact) and their influence on scientific thought.

Impact on Society

The impact of AI on society can be measured using two criteria. Do the products that result from AI help society at large? How much of an industry has been created as a consequence of AI? The use of AI technologies in industry has led to significant economic gains in tasks involving analysis (e.g., machine diagnosis), synthesis (e.g., design and configuration), planning, simulation, and scheduling. AI technologies are also helping chemists and biologists study complex phenomena, for example, in searching enormous databases, creating new molecular structures, and decoding DNA sequences. Stockbrokers use software assistants in analysis, diagnosis, and planning at the expert level. Medical doctors can examine more hypothetical cases, thus including not-so-obvious symptoms in their diagnoses. The best-known economic impact comes from the widespread use of expert-system technologies. (Table 6.2 lists some expert systems that have been or are being used in industry.)

But AI offers more to business than just applications of expert systems. Speech-generation products have been in use for 15 years, and speech-analysis products are beginning to reach the market. Image-processing programs are in daily use in the government, health care organizations, manufacturing, banking, and insurance. It is hard to measure the economic impact of image processing, because such systems often provide better performance rather than savings, but image processing is a billion-dollar-per-year industry. Vision and robotics are affecting manufacturing. Planning and scheduling systems are used routinely in industrial and military settings. The impact of AI in manufacturing manifests itself at least in two ways: in the automation of assembly, quality control, and scheduling operations and in the use of AI techniques in engineering design.

TABLE 6.2 Examples of Expert Systems in Use

Company	Expert System	Purpose
Schlumberger	Dipmeter Advisor	Interpret data from oil wells for oil prospecting
Kodak	Injection Molding Advisor	Diagnose faults and suggest repairs for plastic molding mechanisms
Hewlett-Packard	Trouble Shooting Advisor	Diagnose faults in semiconductor manufacturing
Xerox	PRIDE	Design paper-handling systems
Hazeltine	OPGEN	Plan assembly instructions for PC boards

SOURCE: Data from Raj Reddy, "Foundations and Grand Challenges of Artificial Intelligence," *AI Magazine*, Volume 9(4), Winter 1988, pp. 9-21.

Impact on Scientific Thought

Perhaps the most important impact of AI on scientific thought is the realization that computable information can be *symbolic* as well as numeric. The beginning of AI was marked by the introduction of the programming languages IPL and Lisp, whose purpose was the manipulation of symbolic information. Symbolic manipulation became a fertile ground for problems of searching, organizing, and updating databases efficiently and for reasoning from information. Theoretical and practical results in searching, reasoning, and learning algorithms are now part of the core of CS&E. Furthermore AI has had such a deep impact on psychology, linguistics, and philosophy that a new discipline—cognitive science—has emerged. AI has contributed to other branches of CS&E as well. Many concepts in programming languages and environments arose directly from attempts by AI researchers to build AI software. AI languages like Lisp, Prolog, and Scheme are used in many parts of CS&E. The nature of AI forces experimentation more so than in some other branches of CS&E, and the challenges brought forth by the need to experiment have led to the development of important concepts. AI has also dealt with areas of perception and interaction of artificial agents with the physical world. The challenge of how to deal with multimodal information in a consistent, predictable fashion has resulted in new efforts in applied mathematics and control theory, called discrete-event dynamic systems, which combine dynamic systems and temporal logic. Such hybrid systems offer a suitable modeling tool for many difficult physical and economic models. The following subsections look more closely at three subareas of AI: heuristic search, reasoning and knowledge-based systems, and speech.

Heuristic Search In the 1960s and early 1970s, the tasks that were identified as requiring intelligence were mostly of the puzzle-solving variety. They possessed simple specifications but lacked feasible algorithmic solutions. Tasks such as theorem proving and cryptarithmetic puzzles were studied at Carnegie Mellon University. At the Massachusetts Institute of Technology, the problems of understanding simple sentences in English in the world of children's blocks and solving freshman calculus problems were studied. Work at Stanford University focused on automatic speech recognition and the Advice Taker—a system that would take advice specified in a logical language to solve simple problems in planning courses of action. Even though these problems sound deceptively simple in that humans accomplish them routinely, calculating exact solutions for them can be infeasible. Instead, good-enough answers, which are close to but not

optimal, are accepted if they can be calculated within given time constraints. AI tries to find these good-enough answers using heuristic search. The problem is formulated as a search through the space of systematically generated possibilities, and methods are determined to reduce the search to likely candidates.

For example, a checker-playing program would generate possible moves up to a certain play and would select the move using an evaluation function like "choose a move that results in the largest piece gain." The evaluation functions are heuristic: they are rules of thumb that work most but not all of the time. This work in heuristic search borrows from operations research, statistics, and theoretical computer science. The phenomenal progress in computer chess affords a practical demonstration of the power of heuristic search. In 1967, Richard Greenblatt's program defeated a class-C player at a tournament; today, the program Deep Thought has a grandmaster's rating. Advances in search strategies, combined with special-purpose hardware matched to these strategies, have been an important part of this success.

Reasoning and Expert Systems Fascination with search strategies has led to investigations of reasoning as a process. Since reasoning requires knowledge, the issue of representing knowledge has become central to AI. In the early research on reasoning, it soon became apparent that general-purpose methods would not be capable of delivering expert-level performance on problems that required domain-specific knowledge, and, because of this, early research dealt more with domain-specific problems. Such research has led to expert systems (also known as knowledge-based systems) that are based on a simple idea: symbolic reasoning guided by heuristics over declaratively specified knowledge of a domain can result in impressive problem-solving ability. The main research issues in the development of expert systems are the extraction of knowledge about the domain and the criteria used for decision making.

Speech Perceptual tasks, such as speech, are characterized by high data rates, the need for knowledge in interpretation, and the need for real-time processing. Error tolerance is an important issue. The usual methods for symbolic processing do not directly apply here. An important speech-understanding system developed in the 1970s at Carnegie Mellon University was the Harpy system, which is capable of understanding speaker-dependent continuous speech with a 1000-word vocabulary. The early 1980s witnessed a trend toward practical systems with larger vocabularies but with computational and accuracy limitations that made it necessary to pause between words.

The recent speaker-independent speech-recognition system, Sphinx, best illustrates the current state of the art. Sphinx is capable of recognizing continuous speech without training the system for each speaker. Operating in near real time using a 1000-word resource management vocabulary, Sphinx achieves 97 percent word accuracy in speaker-independent mode on certain tasks. The system derives its high performance by careful modeling of speech knowledge, by using an automatic unsupervised learning algorithm, and by fully utilizing a large amount of training data. Difficulties in building practical speech interfaces to computers include cost; real-time response; speaker independence; robustness to variations such as noise, microphone, and speech rate and loudness; and the ability to handle spontaneous speech phenomena such as repetitions and restarts.

While satisfactory solutions to all these problems have not been realized, progress has been substantial in the past ten years.

The Future of AI

The theoretical and experimental bases of AI are still under development. Initial forays on the experimental end have led to the development of a host of general-purpose problem-solving methods (e.g., search techniques, expert systems). Experimental AI can be expected to broaden the scope (scalability and extensibility) of applications in the areas of speech, vision, language, robotics, expert systems, game playing, theorem proving, planning, scheduling, simulation, and learning. Integrated intelligence systems that couple different AI components (e.g., a speech comprehension program to a reasoning program) will become more common. Collectively, these results should lead to significant economic and intellectual benefits to society.

Theoretical AI should lead to the development of a new breed of approximate but nearly optimal algorithms that obtain inputs as an ongoing process during a computation and that cannot precommit to a sequence of steps before embarking on a course of action.[13] And finally, AI will continue its quest for theories about how the human mind processes information made available to it directly via the senses and indirectly via education and training.

Computer Graphics and User Interfaces

Graphics

Computer graphics has its roots in data plotting for science and engineering. In many problems, the significance of a calculation does not rest in the exact numerical results of computation, but rather in qualitative trends and patterns that are best illustrated by a graph. Indeed, pattern recognition is a well-honed skill in which humans still have substantial advantages over computers.

From the earliest days of printed output, computers have thus been printing graphs as well as tables and lists of numbers. However, paper is a static output medium; electronic display screens offer the ability to create dynamic representations of data. Since the human eye-brain system is better adapted to detecting change over time than to inferring it from a sequence of images or even simultaneously viewed images, the ability to display dynamically changing image forms on a screen is as much a step forward over a graphical depiction (or a set of them in sequence) as a static graphic depiction is over a table of numbers.

Dynamic displays are particularly powerful for the interactive user when he or she can influence the parameters of both the model and its visualization(s). Indeed, the shift from the batch computing[14] typical 25 years ago to interactive computing today enables the user to make real-time changes in input data supplied to a program or in the operating parameters governing the operation of a program and to receive much more rapid feedback. Computer graphics has become a standard tool for visualizing large amounts of scientific data, as well as for the design of objects from scratch or from standard building-block components.

WIMP Interfaces

The use of computer graphics is not restricted to design and scientific visualization; it is rapidly becoming the standard method by which human beings communicate with computers. In particular, the use of typed commands to control the operation of a computer is giving way to the use of so-called WIMP interfaces that make use of *W*indows in which different programs may be run, graphical *I*cons on a screen that represent actions that can be taken or files that can be opened, and *M*ice with which the user can *P*oint to icons or select options from a menu.

WIMP graphical user interfaces have three major advantages over the use of command languages. They simplify computer operation for novices, because they provide a constant reminder of what the allowable actions are at any given moment. For most users, they are faster than typing, since it is usually easier to point to an icon or a menu selection than to type it in. Lastly, they are less susceptible to error, because the use of icons and menus limits the range of choices

available to the user (unlike typing, in which the user may type anything).

A Bit of History

The history of interactive computer graphics goes back to the 1950s: pioneers using MIT's Whirlwind computer in the 1950s were the first to make use of an experimental display screen, and these graphics were later elaborated with the TX-2 computer and its extensive collection of input mechanisms: knobs, dials, keyboard, and light pen.

In 1963, Ivan Sutherland developed a system called Sketchpad for the TX-2, which introduced the notion of interactively constructing hierarchies of objects on a virtual piece of paper, any portion of which could be shown at arbitrary magnification on the screen. Users could specify simple constraints on picture elements to have the computer automatically enforce mathematical relationships among lines drawn with a light pen. This "constraint satisfaction" allowed a kind of free-hand sketching, with the computer "straightening out" the approximate drawings. Sketchpad was the opening round in an era of computer-driven displays as tools for specifying geometry for CAD/CAM in the automotive and aerospace industry and flight simulators for pilot training. MIT's Steven Coons and other researchers in academia and industry developed various kinds of spline "patches" to define free-form surfaces for vehicle design.

In the 1960s, Douglas Engelbart at SRI pursued the notion of using computer displays to "augment human intellect." His group built the first hypermedia systems and office automation tools, including word processors, outline processors, systems for constructing and browsing hypermedia, software for tele-collaboration, as well as various input devices, including the mouse. In the early 1970s, the Xerox Palo Alto Research Center (PARC) pioneered bitmap raster graphics workstations with graphics in the user interface, both for office automation tools such as word processors and illustration programs and for software development (e.g., Smalltalk). These developments, coupled with the reduction in price of display subsystems from $100,000 in the mid-1960s to $10,000 in the early 1970s to about $1,000 today, are the underpinning for today's graphical user interfaces.

The Macintosh personal computer and Windows (for IBM-compatible personal computers) have brought the experience of SRI and Xerox to the popular marketplace, dramatically lowering knowledge barriers to computer usage, and the commercial success of these products

testifies to their significance and impact. As users have required more and more interactive capability with their computers, the popularity of graphical user interfaces and graphical displays has grown by leaps and bounds. Now, millions of users have graphical interfaces on their desks in the office or at home, and it is often possible for users to operate new applications with little or no reference to a manual. Even young children are comfortable using paint programs and illustration programs, not to mention video games and multimedia electronic books and encyclopedias. With these developments, graphics at long last has begun to fulfill in earnest the promise of the Sketchpad days as a standard means of human-computer communication.

Scientific and Engineering Visualization

The science and engineering community has also made good use of graphics technology. An NSF-sponsored report[15] in 1987 emphasized the need of computational science and engineering for graphics to help make sense of "firehoses of data" generated by supercomputers and high-powered workstations. In such data-rich situations, graphics are not merely desirable—they are essential if users are to design objects or perceive subtle patterns and trends. Coupled to interactive computing that enables the user to use the visualization of intermediate stages in the computation to direct further exploration, intuition and insight about phenomena and objects can be developed that would not otherwise be possible.

Scientific visualizations make increasing use of three-dimensional graphics. Depth perception is greatly enhanced by real three-dimensional stereo images that are possible only when the visual inputs to each eye are slightly different, and so some special input mechanism (e.g., glasses) is necessary to present each eye with the information necessary for the brain to construct a three-dimensional image. Technology such as stereo head-mounted displays with miniature display screens for each eye can control all of the visual input received by the wearer and is the basis for "virtual-reality" presentations that completely immerse the user in a synthetic world.[16] Coupled with the ability to "walk through" such a presentation, suitable immersive representations can give the user an even better and more visceral understanding of the spatial relationships involved than is possible even with a non-immersive stereo presentation, provided that the user can control the vantage point from which the scene is viewed.

Graphics are also beginning to assist scientists in developing pro-

grams to analyze their data. Traditionally, scientists had to choose between two styles of software use. They could write their own programs from scratch, one line at a time, until the entire program was written, or they could use a prewritten "canned" application. Writing from scratch gives the scientist tight control over what the program does but is time consuming and prone to error. Using a canned application is faster but lacks flexibility.

However, in recent years, a style of "visual programming" has begun to emerge that is intermediate between these two choices. The key construct within visual programming is the explicit specification of data and control flow between preexisting processing modules; such flows are implicitly specified when a program is written in the conventional manner. Using a screen and a mouse, the scientist specifies data and control flows between different modules, and the system automatically assembles the corresponding program from these modules. The scientist has the freedom of specifying the flows as appropriate to the problem at hand and also has some freedom to customize each module to a limited extent by adjusting a few parameters (typically using a WIMP interface to do so). The result is that scientists can rapidly assemble systems that will perform a given task, much as a music buff can assemble stereo components into a working sound system without being much of an electrical engineer. Further, while this style of programming is currently limited to high-end scientific users, it is likely that visual programming will become increasingly important in the commercial world as well. Microsoft's Visual Basic is a good example of such an application.

Engineering visualizations are at the heart of computer-aided design today. Computer-aided design now includes design of mechanical parts, architectural or engineering structures, and even molecules. Large structures such as airplanes or buildings can be assembled "virtually," i.e., as information artifacts stored in a computer. Parts, substructures, and entire structures can be designed and fitted together entirely on a display and are usually fed to on-line simulations that can analyze and virtually "test" these artifacts. The result is a dramatic shortening of the time required from concept to product.

Touch, Sound, Gestures

Since vision is the communications channel with the highest bandwidth for information transmission, it is not surprising that a great deal of work has been done in graphics. But human beings also make use of sound, touch, and gestures to communicate information. Computer scientists and engineers have thus developed a variety of

devices that allow users to communicate with computers using these channels as well: head or eye trackers, force and tactile feedback devices, gesture recognition via wands and data gloves provided with many-degree-of-freedom sensors, and so on. These devices are especially useful in virtual-reality environments.

Tactile feedback is illustrated well in the problem from biochemistry of understanding molecule "docking." How a complex organic molecule physically "fits" into another is often a key to understanding its biological function. It is possible, although demanding, to calculate the molecular forces that determine docking positions and orientations. But as an aid to developing the biochemist's intuition and feel for docking behavior, computer scientists and engineers have developed ways to depict molecules visually. By controlling their orientation and position through the use of a specially designed joystick capable of responding with graduated forces, the user can orient and move the molecule along the path of least resistance as he or she tries to fit the molecule into the receptor site.

Sound will become a more important medium for interaction with computers in the future. Audio feedback is widely used in keyboards today to inform the user that a key has actually been pressed. Some computer systems have voice synthesis systems that make information accessible to users without demanding their visual attention. Such output would be particularly useful when the demands of a task (e.g., driving) would prohibit the diversion of visual attention; thus an intelligent on-board navigator for automobiles will almost surely provide directions (e.g., "turn left on Willow Street") by voice. Speech input is a more challenging problem discussed in Chapter 3.

Finally, human beings use gestures to communicate meaning and manipulate objects. The Dataglove is a glove with sensors that can indicate the extent to which the fingers of a hand are bent. Appropriate processing of this information allows the computer to construct a representation of the hand's configuration at any given moment. Coordinated with a virtual reality presented via stereo goggles, the Dataglove allows the user to perform certain manipulations of a synthetic object with his or her hand as though it were real; however, the glove does not provide the tactile feedback that real objects provide.

Intellectual Challenges

Computer graphics and interactive computing pose many intellectual challenges for CS&E. In the earliest days of computing, most of the computational power in a computer system could be devoted

to executing the algorithm to solve a problem of interest, and developers of computer systems paid a great deal of attention to making the best use of available computer resources. But with interactive computing that places the user at the center of the process and ever cheaper computational power, a larger and larger fraction of the computational power will be devoted to making the specific application as easy and comfortable to use as possible for the user. Developers of computer systems will pay much more attention to making the best use of human cognitive resources such as attention span and comprehension time.

This trend is reflected in the development of computer hardware. Real-time user interaction requires that the time between user input (e.g., pointing to a screen icon) and computer response (updating of the displayed scene) be very short, on the order of a few tenths of a second. This limit places a premium on very fast processors and also drives the development of special-purpose hardware (e.g., graphics accelerators) that can relieve the central processing unit of some of the responsibility for servicing the user interface.

The importance of special-purpose hardware is complemented by an increasing emphasis on investigating better representations of data and information. While a "better" algorithm to solve a problem is generally one that runs faster, a "better" representation of information in the context of user interaction is one that provides insights not previously accessible.

For example, consider the difference between looking at an orthographic engineering diagram of a metal bracket and holding a metal bracket in one's hand. An orthographic diagram provides front, top, and side views of an object. In some sense, the orthographic diagram is "equivalent" to the metal bracket in hand, since the engineering diagram can be used to manufacture the bracket. But very few individuals can look at an arbitrary object in an engineering diagram and comprehend just what the object is. A much better sense for the object is obtained by manipulating it, looking at it from different angles, turning it in hand. Today's computers provide the next best thing: a visual image that can be viewed from different perspectives under user control. Using such a display provides a better intuition for time-varying behavior than does viewing a sequence of static images, because the viewer need not cope with the cognitively demanding process of identifying correspondences between the images but rather can follow the correspondences from one moment to the next. The best of today's CAD systems provide images on screen that can be rotated, exploded, collapsed, and otherwise manipulated on command.

The techniques of computer graphics offer a host of different choices to represent information, as demonstrated by modern molecular graphics. An important aspect of modern biochemistry is to understand how complex molecules fit together. Among other things, such understanding involves knowledge of the outside contours of several molecules in their "fitted-together" configuration, as well as knowledge of how surfaces from different molecules fit together "on the inside." To take a simple example, imagine two cubical dice that are brought together so that they are face to face. It would be important to know which surfaces were exposed to the outside (e.g., the 1, 2, 3, 4, 5 of die 1 and the 1, 2, 4, 5, 6 of die 2) and which surfaces were touching (the 6 of die 1 and the 3 of die 2).

Visualizations of molecule fittings could use so-called space-filling models, the equivalent of models made of clay that were fitted together. But the surfaces in contact would then be invisible from the outside, and since the precise geometry of the proper fit is determined in a very complex manner, it is not as simple, as in the case of the two dice, to determine what surfaces from each molecule are in contact.

But an alternative representation—the dot-surface representation—provides a notable alternative. The dot-surface representation replaces the space-filling model with a shell representation. However, the shell's surface is not solid; rather it is composed of many dots. These dots are spaced far enough apart that it is possible to see past them, but close enough that the eye can perceive the surface of which they are a part. Using these dot shells instead of space-filling models to represent molecules enables the user to see both the outside contours (as before) but also "through" them to see formerly invisible surfaces in contact with each other.

In many situations, the availability of different representations provides an unparalleled degree of flexibility in depicting information. Computer graphics-based representations can be created, modified, and manipulated with such speed that the utility of different representations can be rapidly determined, and the user can choose which to use under any particular set of circumstances. These representations are limited only by the imagination of their creator, since any representation that can be imagined can be implemented. Thus computer graphics can be said to have spawned a new set of concepts for depicting information that were not practical prior to the computer.

Finally, finding ways to provide perceptual coupling between computer-generated image and user is important. Evolution has provided human beings with rich and coordinated senses of sight, sound,

and touch; immersive virtual-reality presentations, data gloves, audio output, and force feedback are the first steps toward the full exploitation of human senses.

SYNERGY LEADING TO INNOVATIONS AND RAPID PROGRESS

As anyone who has recently purchased a personal computer knows all too well, computer technology advances at an extraordinarily rapid rate. To a large degree, the rapidity of technological change arises from a high degree of interconnectedness among the various subdisciplines of CS&E. The resulting synergy is intellectually powerful and results in many practical benefits as well.

Synergy arises in many contexts. Box 3.1 offered two examples of the impact of synergy between subdisciplines of CS&E. A third example, with more obvious and tangible economic impact, is that of workstations. It is common today to see a multimillion-dollar mainframe computer replaced by workstations that offer much better cost-performance ratios. The growth in the use of workstations is fueled by advances in many areas: networking (that enables workstations to be interconnected and thus enables users to share resources), reduced-instruction-set computers (that speed up computations by large factors), portable operating systems (that can run on computers made by different vendors), and a better theoretical understanding of how compilers can best use hardware resources. No single one of these areas is solely responsible for growth in the workstation industry, but taken together they have produced a multibillion-dollar industry.

The benefits of synergy are not limited to innovation. Synergy also speeds up developments in the field. Consider, for example, the development of the 80X86 processor chip family by the Intel Corporation. (The 80X86 processor is the heart of IBM-compatible personal computers.) In designing the 80286 processor chip, Intel engineers had to use seven different brands of computers depending on the step of the design process. This was before portable operating systems were available, and so a designer had to learn seven different command languages and seven different text editors to complete a microprocessor.

However, shortly after the 80286 was released, portable operating systems came into use at Intel. This meant that designers of the later processor chips had to learn only one command language and only one text editor, and this contributed to the shortening of the time between the 80486 and the 80586 compared to the time between

the 8086 and the 80286.[17] Other reasons for the reduced development time include a graphical user interface to the design software (making the designer more productive), better compilers and algorithmic improvements to the design software (making the design software faster), and simply faster computers (so that the designer can perform even unchanged tasks in less time). And in a fast-moving technology like electronics, shorter design time corresponds directly to better performance and lower cost. Hence the designers of the Intel 80486 and 80586 (as well as their customers) have and will benefit from advances in hardware technology, operating systems, compilers, user interfaces, design software, and theory of algorithms.

INTELLECTUAL AND STRUCTURAL CHARACTERISTICS OF CS&E AS A DISCIPLINE

CS&E is an entirely new entity, neither simply science nor simply engineering. Its historical roots derive from mathematics and electrical engineering, and this parentage gives CS&E its distinctive flavor.

Perhaps most important in creating this flavor is the close relationship between the intellectual content of CS&E and technology. Technological developments have a major influence in defining what the interesting problems of the field are; problems in CS&E often become interesting because the technology necessary to implement solutions becomes available. Put another way, new generations of technology open up and make interesting a variety of new and challenging questions. This intimate link to technology creates a deep connection between the research and educational activities of academic CS&E and a multibillion-dollar computer industry, with intellectual progress strongly influenced by and at times strongly influencing commercial development.

Intellectually, CS&E includes programming, which allows a programmer's thoughts to spring to life on the screen. As a result, computing technology can create virtual realities—universes unconstrained by the laws of physics—and this ability to take part in creating new worlds is part of the excitement of programming. In other words, the objects of study in CS&E (ways to process and represent information) are created by those who study those objects—whereas other scientists study what nature gives them (atoms, cells, stars, planets). This new form of expression also offers opportunities for important intellectual contributions to the formal study of information.

Theory in CS&E often develops after years of practice, with experiments largely establishing the feasibility of new systems. By contrast, experiments conducted by physicists and biologists play an im-

portant role in proving or disproving their theories. The experimental construction of algorithms as programs is more likely to uncover practical issues that were ignored in the mathematical proofs rather than flaws in the proofs themselves. Feedback from experimentation can then lead to the creation of more realistic models that in turn give rise to more relevant theories. In other words, CS&E experiments often demonstrate the irrelevance and inadequacy of previous theoretical work, rather than proving that theoretical work wrong.

The commercial and industrial side of computing is also unusual. Whereas most industries focus on producing tangible goods, an increasingly large part of computing involves software and information. Software and information are unlike other products in that they are primarily an intangible intellectual output, more akin to a useful report than a useful widget. By far the largest cost of software and information involves research, development, documentation, testing, and maintenance (primarily intellectual activities) rather than manufacturing (typically, copying disks and manuals). As a result, capital requirements for software production are relatively much lower than those for widget production, while personnel requirements are relatively much higher.

When the recognition of CS&E's intellectual heritage is combined with understanding of the rapid pace of change, the youth of the field, the impact of a large industry, the magical implications of programming, and the mathematical implications of algorithms, the exciting and enticing nature of CS&E is revealed.

NOTES

1. The notion of CS&E as a discipline based on theory, abstraction, and design is described in Peter Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, Joe Turner, and Paul R. Young, "Computing as a Discipline," *Communications of the ACM*, Volume 32(1), January 1989, pp. 9-23.

2. Personal communication, Donald Knuth, March 10, 1992 letter.

3. Frederick Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1975, pp. 7-8.

4. "Computer Displays," Ivan Sutherland, *Scientific American*, June 1970, p. 57.

5. This division isn't even necessarily unique, since abstractions in CS&E can be created with a great deal of flexibility. A grammar checker builds on a word-processing program, and thus the word processor plays a "system software" role for the grammar checker. But the word processor builds upon the operating system, so that the word processor is applications software from the perspective of the operating system.

6. John E. Hopcroft and Kenneth W. Kennedy, eds., *Computer Science Achievements and Opportunities*, Society for Industrial and Applied Mathematics, Philadelphia, 1989.

7. In practice, the trade-off is even more favorable than this discussion implies. In the former case, the area of the chip would be more than 100 times larger, due to second-order effects that are not accounted for in the VLSI complexity model. In the latter case, it could be possible to reduce the increase in total chip area to less than a factor of ten by designing its functional elements to perform multiple operations in a "pipeline" arrangement. (A pipeline architecture is analogous to a bucket brigade; when it is finished with an elementary operation, an element passes its result to the next element in sequence, thereby allowing the entire array of elements to be kept busy nearly all of the time.)

8. The discussion that follows is taken largely from Lui Sha and John B. Goodenough, *Real-Time Scheduling Theory and Ada*, Technical Report CMU/SEI-89-TR-14, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1989.

9. The reader may find it amusing to multiply two arbitrary 2×2 matrices with a minimal number of multiplications. The standard method of multiplying these matrices requires four additions and eight ordinary multiplications to compute the matrix product. It turns out that it is possible to compute the matrix product with only seven scalar multiplications though at the expense of additional additions (and subtractions). The algorithm for multiplying 2×2 matrices is the basis for a more general algorithm for multiplying matrices of any size that requires considerably fewer arithmetic operations than would be expected from the standard definition of matrix multiplication.

The details of the seven-multiplication algorithm are described on p. 216.

10. Actually, LP is applicable to a much broader class of problems.

11. A problem's complexity depends on the model of computation used to derive it. An area of active investigation today within theoretical computer science concerns the use of different models that differ in the fidelity with which they match actual computers. For example, some theoretical computer scientists consider so-called random access machine models, on the grounds that these models can access any memory location in a constant number of steps, while a Turing machine can access a given memory location only by stepping through all preceding locations.

12. The function 2^n is called "exponential" in n. The function n^2 is polynomial in n, as would be n raised to any exponent. An algorithm that executes in a time proportional to a function that is polynomial in n, where n is the size of the problem, is said to take "polynomial time" and may be feasible for large n, whereas an exponential algorithm is not. The reason is that for sufficiently large values of n, an exponential function will increase much faster than any polynomial function.

13. These algorithms would be nearly optimal in the sense that a given algorithm working on problems of a particular nature would consume no more than a certain amount of computational resources while generating solutions that are guaranteed to be very close to optimal with very high probability.

14. Batch computing refers to a mode of computing in which the user submits a program to a computer, waits for a while, and then obtains the results. If the user wishes to correct an error or change a parameter in the program, he or she resubmits the program and repeats the cycle.

15. Bruce H. McCormick, Thomas A. Defanti, and Maxine D. Brown, eds., *Visualization in Scientific Computing*, ACM Press, New York, July 1987.

16. Virtual reality, the object of both serious intellectual work and far-fetched hyperbole, will be the subject of a forthcoming NRC project to be conducted jointly by the NRC's Computer Science and Telecommunications Board and the Committee on Human Factors. As a mode of information representation, virtual reality is still in its infancy compared to modes such as ordinary stereo or two-dimensional graphics.

17. The time between the 8086 and the 80286 was five years, while the time between the 80486 and the 80586 is expected to be three years (the 80586 will be released in

1992). See John L. Hennesy and David A. Patterson, *Computer Architectures: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, California, 1990, inside front cover.

Solution to the problem posed in Note 9 above.
Let the elements of matrix A be denoted by

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

and similarly for B. If we define

$$x_1 = (a_{11} + a_{22}) \cdot (b_{11} + b_{22}) \qquad x_5 = (a_{11} + a_{12}) \cdot b_{22}$$
$$x_2 = (a_{21} + a_{22}) \cdot b_{11} \qquad x_6 = (a_{21} - a_{11}) \cdot (b_{11} + b_{12})$$
$$x_3 = a_{11} \cdot (b_{12} - b_{22}) \qquad x_7 = (a_{12} - a_{22}) \cdot (b_{21} + b_{22})$$
$$x_4 = a_{22} \cdot (b_{21} - b_{11})$$

Then the entries of C are given by

$$c_{11} = x_1 + x_4 - x_5 + x_7 \qquad c_{12} = x_3 + x_5$$
$$c_{21} = x_2 + x_4 \qquad c_{22} = x_1 + x_3 - x_2 + x_6$$

The significance of this algorithm is not its utility for 2×2 matrices per se, but rather that it is the basic building block for an algorithm to multiply $N \times N$ matrices. Such a matrix can be treated simply as a 2×2 matrix where the elements are themselves matrices of size $N/2 \times N/2$. The result is that instead of requiring on the order of N^3 multiplications and additions (as would be expected from the definition of matrix multiplication), the matrix multiplication requires on the order of $N^{2.81}$ arithmetic operations. While such an algorithm involves some additional overhead, it turns out that for sufficiently large values of N, it will run faster than one based on the definition for matrix multiplication.

SOURCE: This method was first published by V. Strassen, "Gaussian Elimination Is Not Optimal," *Numerische Mathematik*, Volume 13, 1969, pp. 354-356.

7

Institutional Infrastructure of Academic CS&E

The term "institutional infrastructure" is used here to refer to the institutions that have some important bearing on academic CS&E. Thus institutional infrastructure includes major funding agencies that support research, the universities that house academic CS&E, and the various professional organizations that provide vehicles for dissemination of research and other support to the discipline.

FEDERAL AGENCIES FUNDING COMPUTER SCIENCE AND ENGINEERING

An overview of federal support for CS&E was provided in Chapter 1. A more detailed description of each major research-supporting agency is provided below. (Figures cited are presented in constant 1992 dollars and are subject to the caveats specified in Note 18, Chapter 1.)

Department of Defense

The modern military is highly dependent on computers in almost every aspect of its responsibilities, including weapons acquisition, command and control, communications, intelligence, weapons control, and administration.

Among federal agencies, the Department of Defense is the largest single funder of CS&E research; historically a little over one-third of

COMPUTING THE FUTURE

A BROADER AGENDA FOR COMPUTER SCIENCE AND ENGINEERING

Juris Hartmanis and Herbert Lin, *Editors*

Committee to Assess the Scope and Direction of
Computer Science and Technology
Computer Science and Telecommunications Board
Commission on Physical Sciences, Mathematics, and
Applications
National Research Council

"What Is Comp. Sci. & Engg?"

NATIONAL ACADEMY PRESS
Washington, D.C.