COMPUTER SCIENCE

Achievements and Opportunities

Report of the NSF Advisory Committee for Computer Research
John E. Hopcroft and Kenneth W. Kennedy, Chairs

SIAM Reports on Issues in the Mathematical Sciences

COMPUTER SCIENCE: ACHIEVEMENTS AND OPPORTUNITIES

SIAM

Computer Science: Achievements and Opportunities

Report of the NSF Advisory Committee for Computer Research
John E. Hopcroft and Kenneth W. Kennedy, Chairs

This volume is dedicated to Kent K. Curtis,
whose leadership and encouragement led to the
intellectual development of computer research.

Published by the
Society for Industrial and Applied Mathematics
Philadelphia
1989

Cover illustration: The 2500 MFlops/2500 Mips CM-2 is the newest member of the Connection Machine^{R} family of data parallel computers. The 5-foot cube houses the system's 64,000 processors. The Data Vault^{TM} storage system uses dozens of small disk drives to provide very rapid access to 10 Gbytes of data. Connection Machine systems are currently installed in major corporations, government laboratories, and universities. Photograph used with permission of Thinking Machines Corporation.

For additional copies write
Society for Industrial and Applied Mathematics
3600 University City Science Center
Philadelphia, Pennsylvania 19104-2688

Contents

Preface

This report was commissioned by the Advisory Committee to the National Science Foundation Division of Computer Research, which I chaired from 1985 to 1987 and John Hopcroft chaired from 1987 to 1989. The Advisory Committee was concerned about the low Ph.D. production in computer science—less than 300 per year before 1985. For the discipline to achieve a healthy steady state, it would need to significantly expand the number of high-quality computer science departments in the United States. Such an expansion would require, at the minimum, a doubling of the annual Ph.D. production to 600. Such an increase is not without cost. It was clear to us that we would be unable to sustain this increase in Ph.D. production without a corresponding increase in funding from government agencies. To achieve the required resources, computer science would need whole-hearted support from scientists in other disciplines. We perceived that two factors were making it difficult to garner that support: a lack of understanding of the intellectual contributions of computer science by other scientists and the absence of a clear-cut plan for future research in the discipline. The two reports contained in this volume represent an attempt to overcome these factors.

First, we set out to document the intellectual importance of computer science research. To do that, the Advisory Committee asked John Hopcroft to develop a report summarizing some of the most important contributions of computing research over the past three decades. Hopcroft enlisted the assistance of distinguished computer scientists all over the world and the result is the "Scientific Contributions of Computer Science," often referred to as the "Hopcroft Report," which is found in the first section of this volume.

Second, the Advisory Committee decided to identify the important problem areas where increased support for computing research might bear fruit. The result was the "Initiatives Report," the second section in this book. In it, the Advisory Committee recommended three areas— software engineering, parallel processing and robotics and automation—as ripe for significant progress. In addition, they identified some critical resources needed to make these initiatives successful. The reader should bear in mind that these recommendations were made in 1986-87 and do not reflect changes in priorities that have arisen since that time.

When the committee began its work, it had responsibility for all areas of computer research except computer engineering (which includes all aspects of microelectronics except VLSI theory). Therefore, the report ex-

cludes computer engineering from its purview. This explains the absence of microelectronics from the discussion of contributions and initiatives.

Both of these reports were unanimously accepted by the Advisory Committee for Computer Research and passed on to the National Science Foundation in 1987. Some of the recommendations have been adopted and steady progress toward the goals has been achieved. I hope the publication of this volume will help continue the progress which is needed to make computer science a healthy, stable academic discipline.

On behalf of the Advisory Committee, I would like to thank all the people who contributed to making this report possible, along with the National Science Foundation and SIAM for all their assistance in producing and publishing this work. Finally, I would like to acknowledge the contribution of the late Kent Curtis, Director of the Division of Computer Research during most of my tenure on the Advisory Committee. He encouraged us to write this report and ensured that it would have great impact at the National Science Foundation. This book is dedicated to his memory.

Kenneth W. Kennedy
Rice University

Contributors

Forest Baskett
David Clark
Stephen A. Cook
Clarence Ellis
Edward Feigenbaum
Michael J. Fischer
Charles W. Gear
David Gries
A. Nico Habermann
C.A.R. Hoare
John E. Hopcroft
Kenneth W. Kennedy
S. Rao Kosaraju
Butler Lampson
Lawrence H. Landweber
Wendy G. Lehnert
Barbara Liskov
Nancy A. Lynch
John McCarthy
Drew McDermott
Arno Penzias
Fernando C.N. Pereira
John T. Pinkston, III
Tomaso Poggio
Michael Rabin
Edward Riesenfeld
Fred B. Schneider
Jacob T. Schwartz
Dana S. Scott
Charles L. Seitz
Burton J. Smith
Lawrence Snyder
Andries Van Dam

Acknowledgements

Donna Patteson was instrumental in bringing the section "Scientific Contributions of Computer Science" to completion—organizing and editing the section, maintaining contact with the many authors throughout the various stages of the report, and ensuring that the activity did not lose its momentum. Leah Stratmann prepared the report for publication by managing the translation to a consistent format (assisted by the TEX formatting expertise of William LeFebvre) and dealing with the SIAM office. Support for publication was provided by the National Science Foundation and SIAM. John Hopcroft and I gratefully acknowledge all of the contributions, without which this undertaking would have been much more difficult.

Scientific Contributions of Computer Science

Prepared by

Forest Baskett
David Clark
Stephen A. Cook
Edward Feigenbaum
David Gries
A. Nico Habermann
C.A.R. Hoare
John E. Hopcroft
Kenneth W. Kennedy
Barbara Liskov
Drew McDermott
Fernando C.N. Pereira
Tomaso Poggio
Michael Rabin
Richard F. Riesenfeld
Fred B. Schneider
Jacob T. Schwartz
Dana S. Scott
Charles L. Seitz
Burton Smith

Introduction

The impact of computers and computing on our culture is recognized by everyone; what may not be generally apparent is that computer science; as a discipline, is emerging as one of the most intellectually challenging areas of both technical and theoretical study. In the past, scientists made major inroads toward gaining a greater understanding of such issues as the nature of matter, the origin of the universe, and the laws of motion; in the future, the focus of research will shift increasingly toward the exploration of knowledge, reasoning, and intelligence. In the past, tools were developed that increased human physical capacities, improving our physical strength, discrimination, and agility; in the future, tools will be developed that will enable us to multiply our intellectual abilities. Research in the sciences brought about advances in materials and design that made high-speed computers possible; in turn, the use of computers is now completely changing how science is done.

In narrow terms, computer science is the study of symbolic representations and the manipulation of these representations. However, we do not manipulate symbols devoid of meaning. The symbols being manipulated embody knowledge of physical and abstract objects such as atomic particles, biochemical compounds, human behavior, and economic trends. Thus, in broad terms, computer science is concerned with representing and manipulating knowledge. It is apparent that its impacts are far-reaching and that the possibilities engendered by it can be used to expand the knowledge base of many disciplines. All disciplines can incorporate the advances made by computer science and use them to learn more about the world, to solve present problems, and to make predictions about the future. Computer science has gone and will continue to go far beyond mere computation; it will contribute to our understanding of language, abstraction, knowledge representation, problem solving, and reasoning and play a role across a wide spectrum of intellectual endeavors.

The impact of computer science has been particularly strong on the physical and mathematical sciences. The development of large-scale scientific computing has opened up new dimensions of scientific inquiry that previously were inaccessible.

Symbolic manipulation systems have made possible calculations that were once beyond the reach of researchers for reasons of sheer size. Numerical routines now permit the solution of large systems of equations describing mechanical systems, fluid flow, and other processes that otherwise

would have remained intractable. In addition, numerous computer applications have been developed that enable computers to be used to search for oil, to build more fuel-efficient aircraft, and to solve environmental problems, among other things.

Besides advancing science, computer science plays an important role in technology. We are witnessing unexpected and unprecedented technological developments in chip design. In computer architecture, there is a need to conceive and analyze completely new computer structures made possible by new technology. Computer-readable electronic media promise to change the very notion of scholarly material as researchers execute programs to extract, correlate, and otherwise manipulate the contents of books, data bases, and digitally stored objects. These unprecedented opportunities to access knowledge will require significant research into knowledge structures and accessing techniques. For the time being, the technological possibilities are outrunning our imagination to use them; we need new ideas, concepts, and theories. These developments make it imperative that computer science develop quickly in order to guide and exploit the potential opened up by these technological advances.

The purpose of this report is to bring the intellectual achievements of computer science to the attention of the scientific community and thereby illuminate its future potential. Computer science is far more than a support tool; it is more than data processing or scientific computation. It is an intellectual discipline in its own right, and the deeper the understanding of it, the more significant role it can play for other disciplines. Moreover, the newness of the field and the abstract nature of the concepts with which it deals—language, meaning, computation, and complexity—all contribute to a lack of awareness of the revolution in thinking that is quietly taking place. It is important that our science establishment understand these aspects of computer science.

In a report such as this, brevity and selection are very important. No attempt has been made to cover all aspects of the subject. Rather, the intent is to provide a range of topics that indicates the breadth and depth of the field, to demonstrate the impact of computer science on mankind, and to show the necessity of developing the science of computing if the nation is to demonstrate intellectual, scientific, and technological leadership in the next century. Chosen as areas for inclusion were foundations, languages, computer systems, artificial intelligence, and technologies.

Foundations

Stephen A. Cook John E. Hopcroft
Michael Rabin

Research in foundations is concerned with the intrinsic nature of computation and with the formal models underlying computing technology. In the first area, research has given us a framework in which we can formally study the intrinsic complexity of natural computational problems. The theory has yielded the tool of NP-completeness, which has been used to establish the intractability of literally thousands of natural computational problems, ranging from those encountered in daily life such as optimal routing and packing to esoteric mathematical ones. It has also given us a methodology for designing efficient algorithms. Some fundamental problems such as linear programming, matching, and network flow have been solved to satisfaction, whereas great progress has been made on others, such as primality testing and graph isomorphism. From research in formal models, we have identified and determined the capability of fundamental models of computation, and we have developed tools for using these models effectively. For example, we now have theories for programming languages and programming language translators. Research in the foundations of computer science has directly resulted in more applied areas, such as very large scale integration (VLSI) design, theory of data bases, distributed computing, and cryptography.

There are dramatic examples of theoretical work in the 1950s and 1960s that made a substantial and permanent impact. Work in linguistics and programming languages merged, providing the basis for the formal description of programming languages. The theory that was developed allows automation of a programming language translator, so that today it is a trivial task. Whereas the first Fortran compiler required about fifty programmer-years of effort, today an undergraduate student writes a compiler for a language that is far more sophisticated as a term project. In fact, there now exist compiler compilers that automatically generate some parts of compilers from programming language descriptions. Because of the basic research in formal language theory, we can today economically tailor languages to specific

applications. The same formal development has led to methods for natural-language recognition, for filament grammars for certain biological systems, and for constructive solid geometries used in computer-aided design (CAD) systems for describing solid objects and parts. Early work on developing a calculus for describing activity in neural nets led to the notion of *regular expressions* for describing sets of strings of digits or characters. Subsequent work on modeling telephone switching networks led to a model called the *finite state automaton*. These two models proved to be equivalent. Today the model is indispensable in pattern recognition algorithms used to search bibliographic references, modify text, search data bases, and design command structures in operating systems and in compilers.

In this chapter we have selected three topics of current research interest to present in more depth: complexity theory, parallel computing, and randomized algorithms. These three topics promise to play important roles in the next decade in improving performance of computing beyond the limits of deterministic sequential computation.

Complexity Theory

At the heart of foundations research is the intrinsic *computational complexity* of problems. There are both *impossibility proofs*, which provide minimum requirements for resources such as time and space needed to solve a problem, and *possibility proofs*, which provide algorithms that attempt to meet those minimum requirements. These two kinds of results supply both lower bounds and upper bounds on computational resources. However, the lower bounds are much harder to prove, because all possible algorithms for a given problem must be considered. The idea of proving a problem NP hard (a notion explained subsequently) was developed to circumvent this difficulty.

Basic to complexity theory is the notion of a *polynomial time algorithm.* For example, if the time required is bounded by a fixed power of the length n of the input, say n^2 or n^5, then the algorithm is polynomial time. In practice, polynomial time algorithms are feasible for reasonably sized inputs, whereas those that run in exponential time, say 2^n, are not. The class P consists of all problems with polynomial time algorithms.

A nondeterministic algorithm is one with the ability to select or guess a sequence of choices that will lead to a correct outcome, avoiding the necessity of systematically exploring all possibilities. The class of problems solvable by so-called *nondeterministic polynomial time algorithms* is called NP. Many real-world problems not known to be in P are in the class NP. A

problem in NP is said to be *NP complete* if every problem in NP is efficiently reducible to it. Thus, if some NP complete problem turns out to be in P, then the classes P and NP would coincide. The great success of this notion stems from the fact that many combinatorial problems concerning packing, routing, and scheduling, as well as a wide variety of problems in virtually every field of science, have been shown to be NP complete. If someone found a polynomial time algorithm for any one of these NP complete problems, this would lead to polynomial time algorithms for all of them. Whether P = NP is the main open question in complexity theory and a major open question in mathematics generally.

In the last ten years computational complexity theory has also provided the basis for new foundations for *cryptography theory*. A cryptographic encoding scheme requires a decoding function that is easy to compute given the key but too difficult to compute without the key. Thus, computational lower bounds, which are negative results from the point of view of algorithm designers, become necessary to ensure the security of cryptosystems (see subsequent section on randomization in computing). The cryptographic implications go far beyond national security needs. Modern computer networks and electronic transactions in the commercial sector have created a need for a method of transmitting data that is secure from electronic eavesdropping and tampering and for electronic signatures that will stand up in legal proceedings.

Lower bounds in complexity theory are important, but far more energy has been directed to upper bounds, that is, toward developing general techniques for designing *efficient algorithms*. The result is a new generation of algorithms for combinatorial and number theoretical problems, retrieving information, text processing, computer-aided design/computer-aided manufacturing (CAD/CAM), graphics, and VLSI design. Taken together, these techniques have had enormous impact on our ability to apply computers to problems.

Parallel Algorithms

There are two quite different kinds of parallel algorithms: distributed and tightly coupled. A distributed algorithm applies to a group of computers, possibly in different locations. Each computer has its own input data that must be combined with other inputs, usually as part of an ongoing process. For example, a distributed algorithm may be used to implement an airline reservation system. A tightly coupled parallel algorithm, on the other

hand, typically applies to a single computer with many processors all working together to solve one problem. The development of protocols or rules to coordinate the activity of different computers is a fundamental research problem for distributed algorithms. However, in this section we concentrate on discussing tightly coupled algorithms and assume for simplicity that the processors are synchronized and execute instructions simultaneously.

The recent dramatic decrease in the cost and size of hardware makes very natural the idea of building a highly parallel computer with a great many processors. Parallel computers with more than 64,000 simple processors already exist. This development raises fundamental questions: Which problems can substantially benefit from a high degree of parallelism? How can one design parallel algorithms to reap the benefit? Are there problems that cannot significantly benefit and are in some sense inherently sequential?

Which problems can substantially benefit from parallelism? A useful way to formalize "substantial" is to require that the running time of a parallel algorithm be bounded by a polynomial in the *logarithm* of the input size n, for example, $\log n$ or $(\log n)^2$. (Notice that such algorithms would take many fewer steps in general than the input size n.) The class of all problems solvable by parallel algorithms that are fast in the sense just described and that use a feasible (polynomially bounded) number of processors is called NC. This class NC is a precisely defined mathematical class that has been studied extensively and contains many real-life problems.

Notice that a sequential computer with one processor can simulate a parallel computer with p processors in p steps (assuming no overhead or arbitrary-size steps). Thus, it is clear that every problem in NC is in P (i.e., can be solved by a sequential algorithm in polynomial time). Hence, one good way of making precise our first question is to ask, "Which problems in P are in NC?" We give some examples later.

Among the simplest examples of NC problems are the four arithmetic operations—addition, subtraction, multiplication, and division— applied to integers. Here the input size n is taken to be the total number of input digits. In fact, there are parallel algorithms for all four operations that require order $\log n$ time and, at most, n^2 processors. Similar statements can be made for iterated addition and multiplication and for sorting numbers or words. These problems can be shown to be in NC using methods that have been around since the early days of computers, and many of these methods are standard in the switching circuits of today's computers.

More recently, methods have been developed that are not yet widely used and that represent a potentially large application area for parallel computers.

These methods come from a general theorem that shows how to solve many algebraic problems quickly in parallel. In fact, any polynomial function of degree d that can be computed sequentially in C steps can be computed in parallel in order of $(\log d)(\log C + \log d)$ steps using a reasonable number of processors. A polynomial function is a function of several variables that can be computed by a fixed sequence of additions, subtractions, and multiplications. A good example is the determinant of an n by n matrix, which is a polynomial of degree n in n^2 variables. A sequential method is known for evaluating the determinant in a polynomial number of steps using only addition, subtraction, and multiplication as operations. Thus, the theorem shows how to compute the determinant of an n by n matrix in order $(\log n)^2$ steps and puts this problem in NC. The standard sequential method for determinants, based on Gaussian elimination, gives no hint how to do this. Once the determinant is known to be in NC, other techniques can be applied to show that most standard problems of linear algebra, such as solving a system of linear equations, are in NC. Since linear algebra problems are central to a large fraction of scientific computing, the potential applications are enormous.

The preceding linear algebra examples illustrate a theme: fast algorithms suitable for highly parallel computers are often very different from their fast sequential counterparts. This tends to be true, in particular, of *graph algorithms.* For example, consider a bipartite graph, that is, a collection of two kinds of nodes, say, boys and girls. Suppose certain couples, each consisting of one boy and one girl, are designated as satisfying some relation (say, they like each other). In general, one boy can like more than one girl, and a girl can like more than one boy. The *matching problem* is to pair off as many boys and girls as possible into mutually liking couples; such a pairing is called a *matching.* The problem is of central importance in graph theory; it has been studied since late last century, and extensive effort has been put into finding good sequential algorithms for it. All of these algorithms are combinatorial in nature, and the most sophisticated one runs in time $n^{\frac{5}{2}}$, where n is the number of boys and girls.

Recent parallel algorithms devised for the matching problem are quite algebraic in nature and are based on the following fact. Let M be the matrix all of whose entries are 0, except there is a variable X_{ij} in the i^{th} row and j^{th} column if boy i and girl j like each other. Then the rank of the matrix M is the largest number of mutually liking couples that can be formed. At present, the only good way known of finding the rank of such a matrix requires a random number generator. The variables X_{ij} are replaced by small

random integers, and the rank of the resulting matrix is computed using the parallel methods for linear algebra discussed previously. The result will be the true rank with high probability, and the experiment can be repeated any number of times to build confidence in the result.

The preceding algorithm is intended to find the size of the largest matching. To find a largest matching itself requires a more sophisticated random experiment, followed by the inversion of a related matrix. The upshot is that the matching problem can be solved, perhaps not in NC, but at least in RANDOM NC: that is, within the time and processor constraints required by NC but requiring the parallel computer to have a random number generator, while allowing the answer to be slightly uncertain.

Presumably not all problems can benefit substantially from large-scale parallelism. An example is the general linear programming problem. This was recently shown to be solvable in polynomial time, but it is thought not to be in the class NC. In fact, one can prove that linear programming is complete for P, in a sense analogous to NP complete. Thus, if linear programming is in NC, then every problem solvable in polynomial time is in NC, an unlikely outcome. Other problems have been shown complete for P and, hence, are likely not in NC. This knowledge may save wasted energy trying to find super-fast parallel algorithms where none exist.

There are also a number of basic problems in P that no one has been able to put in any of the categories NC, RANDOM NC, or complete for P. Two examples involve operations on integers that are useful in modern cryptographic schemes: given two positive integers a, b, compute their greatest common divisor; given three positive integers a, b, c, compute $a^b \bmod c$. (Here the operation $x \bmod c$ means take the remainder after dividing x by c.)

In general, parallel algorithms are useful not only in programming parallel computers but also in the design of computer circuits. A parallel algorithm intended for a highly parallel computer can be adapted to design a circuit that solves the same problem in time that grows at the same rate as in the original algorithm. Conversely, fast computer circuits can be converted to fast parallel algorithms. For example, the arithmetic operations are usually built in to the hardware of the computer, and on large computers the circuits usually incorporate algorithms that put these operations in NC. Also, we can easily imagine special-purpose VLSI chips for solving systems of linear equations, using the fast methods described previously. Special-purpose chips for cryptography that compute $a^b \bmod c$ have already been

built, and a good parallel algorithm for this problem could have immediate application.

In many ways, the class NC may be more appropriate for circuit design than for parallel program design. In practice computer programs that require more than order n processors may not be very practical, even though the corresponding circuits are practical.. The bald classification of a problem as to whether it is in NC or not is still of fundamental interest, but more generally it is of interest to study the so-called speedup ratio of a problem. This is the ratio of the speedup factor to the number of processors, where the speedup factor is itself the ratio of the sequential execution time to the parallel execution time. The speedup ratio of a problem can never be greater than one, by arguments given earlier, and it is a function of the number of processors. Problems with speedup ratios close to one have parallel algorithms that make efficient use of their processors.

In summary, the design of a parallel algorithm can be fundamentally different from that of the corresponding sequential algorithm, and yet the study of parallel algorithms has already led to simpler sequential algorithms. The mathematics involved is interesting and deep and offers the potential for far-reaching applications as large-scale parallel computers come into their own. The theory has some difficult open mathematical questions. The most basic is whether P = NC; that is, can every problem solvable sequentially in polynomial time be solved in time polynomial in $\log n$ with a feasible number of processors? The answer seems to be no, but like the P = NP question, this question has evaded the best efforts of mathematicians in the theory of computing.

Randomization in Computing

The preceding discussion brings up another fundamental question in the theory of both parallel and sequential algorithms: must random number generators play an essential role? It also illustrates how studying parallel algorithms can lead to improved sequential algorithms, since random algorithms have interesting sequential versions. The past decade saw a dramatic burst of research on algorithms based on the following seemingly paradoxical proposition: when computational problems such as determining whether a given integer is prime or solving certain algebraic equations are beyond the reach of existing algorithms, try to find an efficient practical algorithm that involves use of a sequence of random bits produced, say, by flipping a coin. As is well known, computations employing random sequences were

used in the classical Monte Carlo Method for solving differential and other equations. To solve a Laplace or a diffusion equation, a discrete stochastic process governed by an approximate form of that equation is formulated. The process is then simulated on a computer, and an approximation to the solution is obtained.

What distinguishes the new applications of randomness in computing is that they address discrete problems and produce exact yes or no answers to questions such as whether an integer is a prime or produce an exact discrete structure such as a matching in a graph. The new algorithms are not intended to emulate a continuous process.

Suppose we have a set of elements, some of which are faulty but at least half of which are good. Suppose further that there is a fast computation for determining if an element is good but no deterministic fast algorithm for producing any good element. Assuming that we can randomly select elements of the set with nearly equal probability, we can choose elements and test them until a good element is encountered. The expected number of trials is two, independent of the size of the set. This basic idea is used in numerous sophisticated variations in randomized algorithms.

Let us consider the problem of determining whether a large integer n, say of a size about 10^{300}, is *prime*. The problem of testing large integers for primality is of great importance in modern cryptography, yet despite recent advances, no practical deterministic method applicable to large numbers is known. Based on Fermat's Small Theorem, a property $W_n(x)$ of integers $1 \leq x < n$ is defined that satisfies the following conditions. If n is prime, then no integer $x < n$ satisfies $W_n(x)$, but if n is composite, then at least $3 \cdot \frac{n}{4}$ of the $x < n$ satisfy $W_n(x)$. Furthermore, to check whether a given $x < n$ satisfies $W_n(x)$ requires $\log n$ arithmetical operations. Given an integer n, we now test it for primality by randomly and independently choosing fifty integers $x_1, x_2, \ldots, x_{50}$, all in the range $1 \leq x < n$. We check whether $W_n(x_1), \ldots, W_n(x_{50})$ are true. If for any i, $W_n(x_i)$ is true, we output "n is composite"; if all $W_n(x_i)$ are false then we output "n is prime". It is readily seen that if n is actually prime the algorithm will always correctly declare it to be so. But if n is composite, it may happen that the algorithm will erroneously declare it prime. However, for any given n, and any one application of the algorithm, the probability of such an error occurring is provably smaller than 2^{-100}. Thus, in this mode of applying randomness, we tolerate a negligible possibility of error in return for a vast improvement in efficiency of the algorithm.

As another example, consider the problem of finding an *irreducible polynomial* (i.e., one that does not factor into a product of polynomials) of degree k with coefficients in a finite field Z_p of residues modulo a prime p. This is, even in the case Z_2, a problem of practical importance for the theory of *error correcting codes* and for the design of important computer circuits. It turns out that it is easy to test whether a given polynomial is irreducible. Also, approximately p^k/k out of the totality of p^k polynomials of degree k are irreducible. Thus, a straightforward application of the idea of randomly choosing a polynomial of degree k and checking it for irreducibility will produce an irreducible polynomial in an expected number of k trials. This example presents an algorithm that always produces a correct answer, although we do not know in advance exactly how many iterations will be required. Again, it provides a solution in a situation where no efficient deterministic algorithm is known.

Randomization also has important applications in *parallel* and *distributed computing.* One architecture for a parallel computer places $N = 2^n$ processors at the vertices of the n-dimensional hypercube. Connections between the processors are wires along the edges of the hypercube, so that each processor is directly connected to n other processors. The whole structure is, of course, wired together in ordinary 3-space. Messages between processors are routed along edges through intermediate nodes. Thus, in the case of $n = 3$, the eight vertices are represented by $000, \ldots, 111$; and 000 is directly wired to 100, 010, and 001. To route a message from 000 to 111, we send it to 100 then to 110, and finally to 111. Thus, for a parallel computer with N processors, a single message from any node to any other node will require, at most, $\log_2 N$ time units for transmissions.

In every phase of the computation, all n processors may wish to send messages to other processors. Even if all the destinations are pairwise different, it may well happen that, when routing in the preceding manner, some of the links and intermediate nodes will become bottlenecks where an excessive number of messages queue up. The solution to this difficulty is to have each processor choose randomly and independently an intermediate destination. The totality of messages is then simultaneously routed in deterministic mode to their intermediate destination and from there to their final destinations. It can now be proved that the probability that any queue of messages, or any delay, will be greater than $c \cdot \log n$ is smaller than n^{-c}. Thus, through the use of randomization we ensure congestion-free routing with overwhelming probability.

An interesting area of the application of randomness is to modern *cryptography.* A well-known public-key cryptosystem, the RSA system, employs a number n, called a *key*, that is the product of two large primes $n = p \cdot q$. (The key n is published, but p and q are kept secret.) If a method were found to factor large numbers, then the RSA system could be broken; however, it is conceivable that the RSA system can be compromised even without an algorithm for factoring numbers. By use of randomization, an encryption algorithm employing a public-key $n = p \cdot q$ of the preceding type can be constructed, which *provably* is as intractable to breaking as the problem of factorization. As a matter of fact, to date the best known algorithms for *factoring numbers*, though falling short of being practical for really large numbers, employ randomization. Randomized algorithms also play a role in the *evaluation* and *testing* of proposed cryptosystems. A well-known proposal for a public-key encryption system based on the knapsack problem was broken by a randomized algorithm.

As to other applications of randomized algorithms, let us only mention the areas of *synchronization* and *fault-tolerant control* in distributed computing, *protection of transactions* (such as contract signing and exchange of secret information), and finding parallel versions for algorithms such as *sorting.*

In the past decade there appeared hundreds of articles applying this radically novel approach of randomization to all facets of computer science. Despite all these applications, however, no one has been able to prove that randomization helps in an essential way. The following fundamental question remains open: is there a problem that cannot be solved in polynomial time by any deterministic algorithm but that can be solved in polynomial time by a randomized algorithm? Regardless of the answer to this question, the concept of randomness will continue to play an important role in the design of algorithms.

Computer Languages

David Gries C.A.R. Hoare
Kenneth W. Kennedy Fernando C. N. Pereira
Dana S. Scott

In every field of scientific endeavor, significant advances have been accompanied by the emergence of new notations (and the rules governing them) as outward and visible signs of deeper conceptual understanding. Often the invention of a notation has been the key to better understanding and has resulted not only in new concepts but also in simplification and unification. Good notation rarely has come easily but has been the result of much thought and experiment. One is reminded, for example, of the battle beginning in the sixteenth century between Newton's notation $\dot{y}$ and Leibnitz's notation dy/dx for the derivative, with Leibnitz's emerging as superior in the more general setting.

The advent of the computer some forty to fifty years ago brought about the need for a new kind of notation, for expressing algorithms. Such a notation is called a *programming language*, because one writes a program—a plan to be followed—in it. In the early days, implementation of a language on a computer was a major hurdle, and the existence of an implementation was a sufficient condition for scientific integrity of research in programming language design. Economy of space and time was the chief concern, and because of this, early languages mirrored quite closely the machine language itself. Now, we are much more concerned with other design goals, such as simplicity, economy of notation, generality, expressiveness, and manipulative ability.

The conflict between efficiency of implementation and other design goals remains an intriguing and important part of our field, which differentiates it from the search for notations in other fields. Since the early days, computer scientists have been experimenting with many algorithmic notations, trying to solve the conflict for a particular context or trying to find better ways of defining, analyzing, and comparing notations (which explains to some extent the existence of hundreds of implemented and unimplemented programming

languages). Often, a significant breakthrough has come not from creating a new concept but in finding just the right balance between the desired goals.

The theme of conflict between goals will appear throughout this chapter. Naturally, our discussion must be limited, and we have chosen to provide three views of the field. In the next section, we look at the achievements of a few major, established languages of computer science and mention some less traditional languages of scientific import. Of importance, however, are not the languages themselves but the underlying scientific issues that they are designed to test. Thus, in the following section, we look at issues involved in groups of languages under consideration today: the procedural, parallel, functional, and logic languages. Finally, we review some of the theoretical achievements underpinning the field. The increasingly complex notions involved in programming languages and the unprecedented rigor demanded of implementations have given rise to many new theories of programming languages, which rely heavily on mathematics and logic. And, as often happens, our applications of mathematics have generated new problems that have attracted the attention of mathematicians.

Significant Language Designs and Their Implementations

The early languages: Fortran, Algol, Lisp

We typically take programming languages for granted, forgetting that at one time they did not exist and that their design and implementation were difficult. We forget, for example, that the oldest and best-known language, Fortran, took some fifty programmer-years to design and implement in the middle 1950s. Fortran paved new ground as the first widespread programming notation that was independent of the underlying machine instruction set and the first to permit nested arithmetic expressions and multidimensional arrays. With no previous implementations to learn from and with a requirement of extreme efficiency to encourage acceptance, Fortran was indeed a significant achievement.

Algol 60, developed several years after Fortran under the auspices of the International Federation for Information Processing, introduced more of a sense of simplicity, structure, and conciseness, for it was designed more with manipulation and human understanding in mind. It generalized the principle of nesting of expressions to other constructs, and it introduced the recursive procedure as a means of expressing some algorithms more simply and mathematically—before it was known how to implement recursion. Later,

research showed that Algol 60 could be implemented quite efficiently, so that the recursive procedure came for free. The designers of Algol 60 managed to advance our knowledge in many ways. Algol 60 sparked much research activity in implementation and optimization, and many of the techniques discovered then are still in use today.

Lisp, also developed in about 1960, was designed for computing with symbolic expressions (rather than numbers). Pure Lisp, modeled after Kleene's lambda calculus and having the list of values as its major data type, had no assignment but only the (recursive) function definition and function application; it was simply a formal notation for doing certain kinds of mathematics. Today, Lisp and its newer variants are the lingua franca of the Artificial Intelligence (AI) community.

The efficient implementation of Lisp required new techniques and data structures to be developed, including extensive use of indirect addressing, dynamic allocation of computer memory, and sharing of substructures of data. Remarkably, these techniques were hidden from view and did not appear in the notation. However, efficiency still remained a key problem, and the only way found to overcome it was to incorporate into pure Lisp procedural concepts such as the assignment and loop. This made the notions of implementation evident in the notation itself, compromising manipulative ability and ease of understanding.

Later designs: Pascal, Ada

Work in programming language design ranges from the simple language developed by a single person to the large, complex language developed by a committee, group, or company. Both extremes have their place and can generate significant advances. Here, we describe two such extremes, Pascal and Ada.

Pascal was developed in the late 1960s by a single person, in academia, with the goal to generalize Algol 60 but to keep the language as simple and transparent as possible. It was a tremendous engineering achievement, and today, without a push from any manufacturer or large group, it has become the major tool for teaching programming in high schools and colleges in the United States. A major thrust of Pascal was the extension of the notion of type: it introduced the subtype, the enumeration type, and the record (akin to the cross product of sets in mathematics) as major data-structuring tools. But a prime reason for its success was the simplicity of its design.

The language Ada was developed under the auspices of the United States Department of Defense (DoD) with input from many sources. Ada illustrates how difficult and time-consuming language design can be—not because of political or economic problems but because of scientific and technical considerations.

From 1973 to 1974, a cost study showed that the DoD was spending $3 billion a year on computer software, over half of it for software on *embedded* computers, computers that form part of a larger system such as a radar system. The software was written in many different languages, most of them ten to fifteen years old, and it was felt that a single, new, well-designed language could have a significant impact, especially if it contained more modern concepts on parallelism, tasking, types, and program structuring.

In 1975, requirements for a language were published. Proposals for the language design were solicited and screened; four contracts were let; programming language experts reviewed the results of the preliminary designs and chose two designs to be further developed; and after further work, one was chosen as the language Ada. The language was submitted to the American National Standards Institute (ANSI) for consideration, and after further extensive modification, it was finally accepted by ANSI in 1983, eight years after the process had begun.

Care and thoroughness were used throughout the development of Ada. Much time was spent dealing with technical design requirements before any designs were proposed, and the requirements were modified four times over a three-year period. Throughout the process, computer scientists and others in computing were asked for comments, which were extensively studied. In a sense, Ada was something of a group effort. Few thought, however, that the development of Ada would take so long. In 1975, one computer scientist was scoffed at for saying that it would be ten years before a usable compiler for the language was finished; ten years later, he was vindicated.

From the DoD's standpoint, Ada is indeed successful. More and more companies and programmers are using it, and through Ada they are learning newer concepts of languages and programming. Ada would not have been possible ten years earlier, because the field simply had not advanced enough at that point. Nevertheless, many computer scientists are unsatisfied with Ada. They believe that reliability requires simple notational tools, and they find Ada far too complex and cumbersome. Thus, the search for better notations continues.

Less traditional languages: BNF, troff, TEX, Postscript

Besides the more traditional kinds of programming languages just discussed, there are other kinds of computing languages, whose sentences look less like programs, but nevertheless fall in our field of language design. We discuss several such languages here. All of them represent significant achievements, and the later ones, those dealing with editing and typesetting by computer, have had a marked effect on the productivity of thousands of people.

During the design of Algol 60, the language BNF, or Backus-Naur Form, was developed to describe the syntax of Algol. BNF and its variants quickly became standard tools for describing the syntax of languages and opened up the rich field of formal languages, discussed in a later section. This quite theoretical field has been of major practical import, for it has led to the mechanical generation of parsers or recognizers for a language from its syntactic description, thus speeding up implementation of a language considerably. Here, we see how the concerns of the programming language community led to the development of a whole new field and its application in other areas (such as linguistics).

The languages troff and TEX, developed by computer scientists to automate the formatting of documents, have changed the world of typesetting. In fact, TEX is now used by the American Mathematical Society for setting its journals, and authors can contribute electronic versions of their documents ready for printing, thus bypassing the traditional typesetting and proofreading. Troff and TEX are indeed programming languages: a document consists of a sequence of commands and text, with the commands describing how the text is to be typeset. Employing many concepts developed in the field of programming language design, these languages have spurred a good deal of research into the structuring of documents. These languages are a clear indication that computer scientists can apply their programming language talents in neighboring areas to achieve tremendous increases in productivity and economic benefit.

Further progress has come in the form of so-called WYSIWYG (What You See Is What You Get) editors in which the document in its final form is displayed on the screen and edited, so that the user need not deal with the internal form of the document as a set of commands and text. WYSIWYG editors were developed and implemented some ten years ago in computer science laboratories, but suitable hardware was not available; it required the advent of the inexpensive, fast personal computer, with suitable display facilities, to make them as popular as they are today.

As another achievement in document processing, we note that the relatively new language Postscript, designed purely for computer and not for human use, is revolutionizing how typesetting programs interact with laser printers. Computer science has many scientific achievements, such as Postscript, that have tremendous import in the way computers are used but that are hidden from the view of the layperson.

Advances in Language Design Concepts

Widespread notoriety and application are not the only touchstones of scientific progress, and scientific improvements are found in many research languages developed over the years. In this section, we organize our discussion of scientific achievements around ideas rather than individual languages, touching on concepts in procedural, parallel, functional, and logic languages.

Procedural languages

The need for efficiency has sparked a great deal of research into the automatic analysis of programs in languages such as Fortran, Pascal, and Ada, with the goal being a theory of program optimization. Some of this work is purely theoretical in nature; some of it is theoretical but with very practical results. Register optimization, recursion elimination, analysis of redundant operations, identification of variables that are not used before their values are redefined, and interprocedural analysis are examples of the kinds of optimizations developed by computer scientists that have been incorporated in commercial compilers.

Research in language implementation and optimization has also influenced new machine architectures. An important example is provided by research on reduced instruction set computer (RISC) systems, in which higher performance is achieved in the machine architecture by relying on the compiler to perform some of the tasks normally done in hardware. These ideas, developed by the 801 project at IBM Research, the RISC project at Berkeley, and the MIPS project at Stanford, have been incorporated into the design of more than ten commercial computers.

The structuring of large programs or systems (sometimes over 1 million lines) is of tremendous practical import, and for years researchers have been investigating the structural and notational issues involved in such systems. Some key ideas for providing structure, beyond the conventional subroutine,

have involved the notion of *type*, and in the rest of this section we discuss advances concerning this notion.

The type of a variable defines the set of values with which the variable can be associated, and this in turn defines the set of operations that can be applied to it. For purposes of efficiency as well as early detection of errors, Algol 60 (and later Pascal) employed *strong typing*: the type of a variable depended only on the syntax of the program and not on its execution. Lisp, on the other hand, for reasons of expressiveness and flexibility, used *weak typing*: the type of a variable could change during execution. Here, again, we see the conflict between expressiveness and manipulative ability, as opposed to efficiency, a conflict that in this case has not yet been resolved.

One attempt to provide a compromise between strong and weak typing is *polymorphism*: type remains a syntactic property, but the type of some variables (for example, a procedure parameter) varies in a disciplined manner. For example, it should be reasonable to write a single procedure to sort an array of elements of *any* type as long as that type has an ordering $<$ on it. Incorporating the concept of polymorphism cleanly into a language has been difficult, partly because the notion of type was not well defined. The languages ML and Ada allow forms of polymorphism, as do most other newer languages.

A second way to make a language more powerful is to enrich its set of types. For example, Pascal introduced the *record* as a finite collection of named objects, akin to the cross-product of sets in mathematics. Also, the language APL (ca. 1968) was built completely around the multidimensional array and provided extremely concise notation for operations on it. Brevity was a chief concern, and one could write a task in one APL line that might take fifty in another language. Unfortunately, these one-liners were too concise, and rarely were programs modified because they could not be read. APL showed that brevity by itself was not enough; attention to human understanding and to rules for manipulation are needed. Further, because of the dependence on the array, APL programs were typically inefficient.

The language SETL (ca. 1972) introduced the set and the sequence (i.e., the tuple of any length and of values of any type) as primitive types. SETL is extremely expressive, and a program that can make effective use of these types can be developed in perhaps one tenth of the time required in a more conventional procedural language. The extreme expressiveness in SETL, however, made its implementation inefficient. This conflict has generated a great deal of research on implementing such general types and on analyzing

a program to detect whether more restrictive and efficient implementations can be used for it.

The most flexible scheme is to allow the programmer to define new types and their implementations. This notion is intertwined with *data abstraction*: a programmer abstracts away from the implementation and simply defines the set of values of interest and the operations to be performed on them. A program is then written and understood in terms of variables of the newly defined type, and not in terms of the implementation of the type. In fact, the programmer is prohibited from referring to details of the implementation. This allows the level of understanding to be raised as high as the problem domain itself, if types supporting the ideas and notation of the problem domain can be developed. This idea was embodied in the language Simula 67 (in about 1967), but the recognition that a mathematical entity called a *type* was involved became apparent only during the 1970s.

During the 1970s, researchers began experimenting with structuring mechanisms built around the notion of types. Some strongly typed languages with such mechanisms are CLU (ca. 1974), Modula (early 1980s), and Ada, with its *generic package*. Other languages, called *object-oriented languages*, have been developed, of which Smalltalk is a prime example. Object-oriented languages have generally been weakly typed, allowing greater freedom of expression but a corresponding decrease in execution efficiency and early detection of errors. With object-oriented languages, the notion of type inheritance has been of interest and has received formal study.

A final technique developed was *type inference*. Typically, in a strongly typed language the programmer is required to declare explicitly the type of each variable used in a program. The language ML pioneered the idea that strong typing could be achieved without all explicit declarations if the language processor could *infer* the type of each variable based on its use.

The notion of type has played a central role in all the work on structuring, and this has required study of the type itself, beginning with the notion of a type as a set of values. The development of theories of type, type inheritance, and type inference has involved algebra, category theory, logic, and philosophy and has attracted the attention of scientists in these other fields.

Parallel languages

The notion of parallelism, or concurrency, originated in operating systems. In the 1970s, it became the purview of programming language design, for

one could not study various ideas concerning parallelism, such as synchronization and message passing, without notations for expressing them. Thus, studies of parallelism were begun by computer scientists before distributed programming and parallel-machine architectures had become economically important.

The von Neumann computer itself handled parallelism using the *interrupt*: at any time, an event such as the end of an input/output operation could cause execution to be interrupted and begun at a different place. To provide any hope of understanding parallelism, this completely undisciplined interrupt had to be hidden from the programmer's view, and various notational mechanisms were introduced for this purpose. The *semaphore* was one such early mechanism. Also, the *monitor* was introduced as a structuring mechanism to encapsulate all operations whose parallel operation might interfere (because they operated on shared data) into a single program part.

Two general models of parallelism have been studied: the shared-memory model, in which different processes share and operate on the same variables, and the message-passing model, in which each process has its own variables and communicates with other processes only by message passing. Notations and theories for each model have been introduced, and for each model various methods for proving programs correct have been put forth.

Among notable examples of languages that cater to various forms of parallelism are Concurrent Pascal, Modula-2, Ada, and CSP. CSP, which has become the lingua of researchers in the theory of concurrency, deserves special mention. It is an example of a serious notation that needed no implementation, for its prime purpose at the time was the study of issues rather than the implementation of programs. Some time after its appearance in 1978, however, CSP became a major influence in the architectural design of a machine composed of hundreds or thousands of communicating computers.

Today we have massively parallel computers, as well as vector machines like the Cray and architectures like the Floating Point array processor. Taking advantage of the parallelism in the different architectures requires new languages. Another approach is to develop new methods for extracting parallelism from a program written in an existing language. Study of this approach has resulted in many advances in analyzing the structure of programs, which have been incorporated into prototype systems and thereafter into a number of commercial compilers.

Functional programming

A major criticism about procedural languages is the obstacle they present to manipulating and analyzing programs. Procedural languages typically do not allow the simple replacement of one program segment by another or the building of programs on a higher level; their mathematical laws are too complex.

In 1978, the language FP (for Functional Programming) was introduced as an alternative to the procedural language. Like pure Lisp, an FP program consists mostly of function definitions and applications. But it also can include various *combining forms* used to build and manipulate higher level programs. Most important, FP is based on an algebra of programs that gives rules for manipulating programs written in terms of these combining forms. Thus, one can manipulate and solve programmatic equations in unknowns using algebraic laws, just as one does with arithmetic expressions.

This idea of a programming language based purely on mathematical reasoning is a great idea, but efficient implementation has continued to be a problem and has therefore been studied intensively. Other functional languages, such as SASL and Miranda, have been developed; research has been performed on the theory of functional languages; and research into the design of parallel architectures to support efficient implementation is underway. The idea has become intimately connected with the concept of *data-flow languages*, whose development began in the early 1970s.

Functional programming has given us a new way of viewing algorithms; it has provided us with new scientific tools for writing, manipulating, and analyzing programs that are just beginning to be explored.

Logic programming

A program specification can be thought of as a sentence of some predicate logic: it describes the relation among the input variables that must hold initially and a corresponding relation of the output variables that hold upon termination. Programming would indeed be simplified if such a specification could be used directly as program: give the computer a specification and input values, and it generates the corresponding output values! However, this is equivalent to automatic theorem proving, for a proof of the existence of output values that satisfy the specification has to be generated. Theorem proving is known to be extremely inefficient, and the scheme seems infeasible.

Two breakthroughs made logic programming feasible. First, the logic used was restricted to a form known as *Horn clauses*, which reduced significantly the amount of searching needed in generating a proof. Second, the technique called *unification*, which had been in use in research in theorem proving, was found to be useful. The first demonstration of logic programming was with the language Prolog in the 1970s. Today we find logic programming systems on our personal computers, and the idea of representing knowledge bases as logic programs was a key part of the Japanese fifth-generation computer project.

However, efficiency continues to be a problem. Different forms of the same specification can require drastically different times for proof generation, and the writer of the specification has to know enough about the underlying theorem prover to use the most efficient form. Sometimes, the writer must place hints to the theorem prover directly in the specification, compromising the simplicity of the scheme. The need for efficiency—while retaining the simplicity of logic programming—continues to stimulate further research. Ongoing work includes attempting to eliminate the need for the operational requirements in the specification, developing new theories that reduce execution time, developing computer architectures expressly for executing logic programs, and exploiting the parallelism in existing architectures. The field has grown to cover wide theoretical and practical ground, with important overlaps with AI, theoretical computer science, computer architecture, data-base theory, linguistics, and logic.

Programming Language Theory

The need for understanding, analyzing, and comparing languages and programs, together with the computer's requirement for precision and rigor, has spawned several supporting theoretical subfields that have grown into significant research areas in their own right. These subfields have a rich, deep, and significant structure. In this section, we are limited to discussing three theoretical areas: formal language theory, denotational semantics, and axiomatic semantics.

Formal language theory

In the 1960s and 1970s, intense study of the syntax of formal languages resulted in many theoretical achievements. But the field was influenced from

the beginning by the practical need to write *parsers* for programming languages, to be used in the translation of programs into the machine language of a computer. This influence led to the development of programs that mechanically produce a parser for a language from its syntactic description. Today, many programming language processors are written using such parser generators. Further, these parser generators are used not only for traditional programming languages but for other notations as well. The same formal development has led to methods for natural language recognition, grammars for certain biological systems, and constructive solid geometries used in CAD systems, to name a few. This work has also helped in the automatic production of "syntax-directed editors", which allow a program to be edited in terms of its syntactic structure and not only as a linear sequence of characters.

Denotational semantics

One aim of our programming language theory is to create a rigorous context for explaining the *precise semantics* of the full range of programming concepts. Moreover, this foundation must support the development of practical tools for reasoning about programs. The first definitions of the semantics of a programming language were *operational* in nature—they were in terms of how some (abstract) machine would execute a program. Although intuitively satisfactory, operational definitions were not suitably amenable to mathematical analysis, and various questions one might pose about a language simply could not be answered.

Significant progress was made in semantics with the development of *denotational semantics* in the early 1970s. A denotational definition of a language is a function that for each input state of a program yields its output state. This approach has several advantages, most of which stem from the fact that a denotational definition is amenable to mathematical analysis.

For example, by restricting attention to programs whose denotational definitions are continuous (a class of programs that include those used in practice), we can analyze the semantics of a programming language in terms of fixed points of continuous functions. Thus, the meaning of a loop or recursive procedure is defined as the least fixed point of a certain function, and the existence of the fixed point can be proved. As another example, denotational semantics can be used to prove the soundness of an axiomatic proof rule (see the next subsection) in a rather natural way; using a purely operational definition of semantics, this would be far more difficult.

The development of denotational semantics resulted from the introduction of *domains* and *domain equations*, which generalized earlier ideas of *algebraic semantics*, in particular by allowing domain constructions to include mathematical spaces of procedure denotations and other operators. In the past fifteen years there has been intense development of denotational semantics, and denotational semantics has begun to influence the design of new languages. Examples include Standard ML, HOPE, CLEAR and SASL.

Axiomatic semantics

A specification of a sequential program S is often given in terms of a relation P that its input variables must satisfy and a corresponding relation R that its output variables must satisfy. The notation $\{P\}\ S\ \{R\}$ has the meaning: execution of program S begun in a state in which P is true is guaranteed to terminate in a state in which R is true. In the late 1960s, the problems of understanding programs gave rise to the idea of defining a language not in terms of how S is executed but in mathematical terms by which statements $\{P\}\ S\ \{R\}$ could be proved. In such an axiomatic definition, each statement of a language is defined by an inference rule $H\ /\ \{P\}\ S\ \{R\}$, where the hypotheses H state the conditions under which $\{P\}\ S\ \{R\}$ can be inferred. This axiomatic approach, a direct application of formal logic, has been a major influence ever since. For example, the axiomatic approach has influenced language design, for it was evident that the simplicity of an axiomatic definition was often reflected in the simplicity of a language as a whole.

Program logics, as well as rules for meaning-preserving *transformations* of programs, raise the fundamental logical problems of soundness and completeness, and their investigation has drawn upon many techniques of logic. However, they have also suggested new logical problems. For example, the notion of *relative completeness* arose in the course of proving the completeness of a programming logic. Novel logics for specifying program behavior, such as *dynamic logic*, have been developed. In the realm of parallel programming, *temporal logics* and *trace logics* have been developed and analyzed to deal with the problems of nondeterminancy and infinite execution sequences. Thus, it can be seen that mathematical logic has been the inspiration of a great deal of work in programming language theory. This work in turn has caught the attention of logicians and is proving a fruitful source of interesting problems for them.

Investigations of program development have also led to the use of a program logic as a *calculus* for the derivation of programs, the goal being a collection of heuristics for program development that are based on developing a program and its formal proof of correctness hand-in-hand. The material is working its way into textbooks and influencing the teaching of programming.

The use of the computer requires an unprecedented amount of detail, precision, and rigor. Experiments indicate that this seemingly inherent complexity of programming can be overcome in many cases by the judicious use of formal manipulations according to a program calculus and an underlying predicate logic. In a sense, Hilbert's program of formalizing mathematics and proofs is finding direct application in programming; for some scientists, formal proofs of theorems as sequences of applications of inference rules are an everyday occurrence. The formal derivation of programs is far from pervasive. Nevertheless, the results are impressive. Older algorithms have been described more simply, and the techniques have been used to develop new sequential and parallel algorithms.

Finally, we mention significant advances in the mechanization of conventional and programming logics. The Boyer-Moore theorem prover has been used to prove theorems in many fields. The implemented language Gypsy has been used to prove various properties of programs for communications processing applications. Nuprl, a language for doing constructive mathematics and designed around the notion that the type of a logical proposition is the set of its proofs, presents an interesting alternative to programming. From a constructive proof of a theorem (in a certain form), Nuprl can extract a program, thus reducing program construction to proof construction.

Computer Systems

Forest Baskett
A. Nico Habermann
Fred B. Schneider
David Clark
Barbara Liskov
Burton Smith

Research in computing systems is concerned with developing principles and tools to enhance the accessibility and power of computers. The goal is to bridge the gap between the demand for computing imposed by applications and the supply provided by extant technologies. Research is performed by designing systems, analyzing systems, and—most importantly—abstracting key problems and devising mechanisms to facilitate their solution. Thus, in addition to producing systems, researchers make important contributions by inventing new abstractions and devising ways to manipulate them.

The nature of computing systems research is diverse. As in more traditional sciences, there is an empirical element. Programs do not behave in random ways, and knowledge of how they do behave allows the structures that execute them to be optimized. Unlike the natural sciences, however, the phenomena being studied are of our own making. Changing the phenomena—that is, the hardware or software—is a perfectly acceptable way to avoid a problem. Finally, computing systems research has a significant engineering component. Computing systems tend to be complicated enough that building them is the only way to evaluate certain ideas.

We cannot hope to survey research in computing systems in a short space. The field is enormous, spanning aspects of hardware and software design. However, it is possible to get a taste of computing systems research by looking at a few of its scientific contributions. We do this as follows. The examples selected are representative of the type and style of research in the area, although they reflect our biases.

Operating Systems

Sharing is one way to make an expensive resource, like a computing system, more accessible to a collection of users. To put this principle into practice,

computing systems researchers designed systems programs, called *operating systems.* These allowed a processor and peripheral devices to be shared among a collection of users, giving each user the illusion of having a virtual computer to himself or herself. Achieving this illusion was not a simple task, however. Program execution had to be controlled to prevent different users from interfering with each other and to ensure efficient use of available resources.

The processor was shared by time-multiplexing it among user programs. Task switching was performed by the operating system, which received control in response to hardware-generated interrupts. However, designing interrupt handlers proved to be a difficult task. Predicting in what state an interrupt handler would start execution was impossible, and consequently it was difficult to ensure that the handler would always do the correct thing. Indeed, an interrupt handler might even itself be interrupted!

The insight that processors with interrupts were just a form of asynchronous parallel processes provided a liberating abstraction and an elegant way to think about the problem. A variety of primitives and algorithms were quickly developed to synchronize asynchronous processes. These, in turn, allowed the interrupt handler problem as well as a host of others to be viewed in a single framework, which allowed intrinsic properties of solutions to be separated from implementation details associated with a particular machine. And the framework allowed questions to be formulated that had never even been articulated before. For example, although memory access in early machines appeared as an indivisible operation, computer scientists also investigated models in which it was not atomic. Some of the standard problems—known to be important in implementing operating systems—could be solved in this model; others could not be. The existence of this body of research meant that answers were readily available as (then) unforeseen advances in very large scale integration (VLSI) technology and computer architecture have permitted construction of memory that does not satisfy this atomicity assumption.

Today, the study of asynchronous parallel processes has had impact beyond computing systems research. Within computer science, an active research area in the theory community concerns communication and computation costs inherent to parallel algorithms. Researchers in semantics and cognitive sciences have investigated various models of parallelism. In addition, renewed interest by logicians in temporal and other modal logics is largely a consequence of recognition by computer scientists of the suitability of such logics for proving properties of parallel programs.

Sharing a computer among users involves more than just sharing processor cycles among programs. To maintain the illusion that each user has a private machine, memory usage must also be controlled. This is because programs written in isolation might use identical names to designate (what should be) different memory locations. One solution to this problem, developed by computing systems researchers, was *virtual memory*, an abstraction that, like real memory, maps names to memory locations. Virtual memory is managed by the operating system. Tables are maintained describing the mapping in effect, and information (in units called pages) is moved from relatively slow peripheral storage devices to main memory and back as it is needed by executing programs. As a result, it is possible for each user's virtual machine to behave as if it has a private memory that is larger than the one actually connected to the processor.

Virtual memory makes it possible to write programs without concern for the actual memory size of a computer. It illustrates a pervasive theme in computing systems research: defining and implementing abstractions that package technology for easier use. As another example, file and data-base systems implement abstractions that allow information to be saved and retrieved without concern for how or where the information is stored. In fact, the success of many operating systems, including the popular UNIX system, can be attributed directly to the clean, high-level abstractions they present to their users. These abstractions allow programmers to devote more time to their applications and less time to dealing with idiosyncrasies of the system. Inventing clean, high-level abstractions is difficult, but then, designing tools that will be used for as yet unimagined tasks is always hard.

Virtual memory illustrates another important aspect of computer systems research: controlling resource use to achieve high utilization. To load a new page into main memory, we might have to move back an old page to the peripheral storage device. The obvious question is which page to evict. Empirical studies of program behavior reveal that page references are not entirely random. Programs tend to exhibit locality: a recently referenced page is more likely to be referenced sooner than a page that has not been referenced for some time. Locality can be exploited in designing a page replacement algorithm, allowing a virtual memory system to perform considerably better than if page-out decisions were made randomly. Here, experimental evidence concerning program behavior allowed an aspect of the system to be optimized.

Processor cycles and memory pages are just two examples from a diverse collection of resources that an operating system must manage. To ensure

adequate system performance, access to all of these resources must be controlled. In early systems, scheduling policies drew heavily from previous work in telephony. However, it soon became apparent that new policies were required. Resource requests in computing systems (unlike telephone systems) are characterized by high-variance distributions, as exemplified by the rule that a large percentage of a resource will be required by a small number of the requesters. To ensure high resource utilization with these high-variance distributions, researchers derived new scheduling and allocation methods. Contributions to traditional job-shop scheduling theory were also made. Problems in analyzing computing systems led to advances in traditional operations research methods, including new methods for solving queuing models.

One might think that today's personal computers render much of this work obsolete, since machines are no longer shared by users. Just the contrary is true. First, expensive resources, like high-resolution printers and file storage systems, must still be shared. Second, most personal computers allow a user to run several programs concurrently. Window facilities, for example, allow the user to interact concurrently with a number of programs, thereby supporting a synergism between these tools. Although the processor is not being shared among users, it is being shared among these tasks, and the theory and solutions developed for implementing multiuser time-sharing systems still apply.

Computer Architecture

The needs of some applications are defined in terms of cost and raw computing, communication, and storage capacity. Computer architecture is the area of computing systems research concerned with designing computers to satisfy those needs. The term *architecture* is particularly appropriate because design is done at a fairly high level of abstraction. At any given time, current technology provides building blocks—arithmetic, logic, and storage elements—with certain performance characteristics. The computer architect attempts to combine these building blocks so that they cooperate harmoniously to implement a structure that satisfies performance and cost goals. By intelligent choice of abstractions and/or their implementation, we can exploit the latest technological breakthrough or overcome a technological bottleneck.

One example of an innovation in the implementation of an abstraction is the idea of instruction pipelining. The machine language programmer

is told—that is, provided with the abstraction—that instruction execution occurs as a sequence of five operations:

1. The instruction is fetched from memory.
2. The instruction is decoded.
3. Operands are fetched from memory.
4. The operands are combined as prescribed by the instruction.
5. Results are stored in memory.

The naive implementation of this five-operation cycle would be to process one instruction completely, then the next, and so on. For technological reasons, accessing memory usually involves a significant delay. Measured in units of memory-access delay, the elapsed time between completing each instruction would be at least 3 memory access times—one access time for each of steps 1, 3, and 5. Notice, also, that the memory is left idle during steps 2 and 4. A *pipelined* implementation of instruction execution could complete instructions at close to twice this rate. A pipeline works just like an assembly line and exploits the empirically observed absence of dependencies between adjacent instructions. In this case, while one instruction is being decoded, the next is fetched, and so on. The result is that an instruction can be completed every 3 memory-access delays. Moreover, memory is kept busy all the time.

The idea of pipelining is not restricted to instruction processing. It can be applied at various levels in the design of a computing system. Pipelined arithmetic units enable time-consuming operations, like multiplication, to be accomplished at a high rate. Pipelining can also be applied at the highest level of the system. A class of computers, called *systolic arrays*, is able to achieve extremely high throughput because these computers consist of a collection of functional units connected into one large pipeline that implements the data transformation described by a program.

Caching is another example of how computer architects are able to achieve high performance by conceptual breakthroughs rather than relying on technological ones. High-speed memory tends to be expensive, and all memory has limited bandwidth. Ideally, we desire cheap, high-speed, high-bandwidth memory. The question is how to implement a memory abstraction with these performance characteristics; caching provides the answer. A *cache* is a small,

high-speed memory that serves as a front end to a larger, slow-speed memory. Information is moved into the cache when it is needed and moved out when it is no longer needed. Because programs exhibit locality in memory references, the cost of moving information into a cache can be amortized, and most memory accesses will be to the high-speed cache. Caches that are smaller than 1 percent of the total memory size are sufficient for dramatic performance improvements (i.e., over 95 percent memory accesses can be satisfied by the cache). Thus, a relatively high-speed memory is implemented by using a large amount of slow-speed memory, a small amount of high-speed memory, and mechanisms to transfer information between them.

The issues in implementing a cache are similar to those associated with implementing a virtual memory. It is not rare for both hardware and software to involve the same abstractions. Consequently, both can benefit from new abstractions and innovations in implementing old ones. In fact, the computer architect often must choose between realizing an abstraction directly in hardware and choosing more primitive abstractions to implement, leaving it to systems software to realize that abstraction. This also means, however, that work in computer architecture can be responsible for new research directions in programming languages, compiling, and algorithms. For example, recent investigations into reduced instruction set computers (RISC), which are based on the premise that small and simple instruction sets are best, have stimulated research into compiler techniques.

Most agree that still more computing power will be needed to execute applications programs that unravel new complexities in, for example, particle physics, quantum chemistry, and genetics and to handle engineering applications such as computer-aided design and computer-controlled manufacturing. Future computers will meet these goals not only by innovation in electronic component technology but also by an architectural revolution—altering the instruction processing abstraction from the single-stream (one instruction at a time) Von Neumann model to a model in which multiple instruction streams are processed in parallel. The correct choice of parallel processing abstraction, if indeed there is a correct choice, remains a hotly debated topic. Past work from operating systems gives some insight into parallel processing, but only for instruction execution abstractions that correspond to parallel asynchronous processes. Other computational models have also been proposed. Examples of such abstractions are the data-flow and functional programming models. In both, a set of equations of a particular form, rather than a sequence of instructions, defines a computation.

As a result, the instruction stream is not defined prior to execution but is created as the computation proceeds.

Most of the problems associated with parallel architectures based on parallel processes are associated with the communications abstraction. Some architectures employ a shared memory for communication. However, implementing this abstraction requires an extremely high bandwidth memory because the memory must service many (fast) processors. New caching schemes, in which each processor has its own cache and program variables might reside concurrently in more than one cache, are currently being investigated as a way to solve this bandwidth problem. A second communication abstraction under investigation is the use of message passing. Realizing this abstraction requires communications channels that interconnect processors. For the extremely large numbers of processors contemplated, it is technologically infeasible to connect every pair with a direct link. Yet, restricting the topology of interconnections makes an architecture poorly suited for programs in which there is a clash between the processor interconnection topology and the program's communications topology.

Additional research will be required to resolve these problems. Abstractions will be proposed, and experiments will affirm or deny their viability. The experiments will be costly because designing and engineering a large computing system can require significant time and resources. Moreover, it does not suffice to build just the hardware. For abstractions that differ significantly from those with which we have experience, software must be constructed and programming methods must be developed. And, although computers are universal machines and therefore any new computing system could be simulated, it is still necessary to build the hardware because simulation is just too slow to gain meaningful experience. However, the payoff can be great. All the sciences can benefit from the tools that result.

Networks and Distributed Systems

Not all application requirements can be translated into access, capacity; and cost. Sometimes, an application imposes constraints on the structure of a system, usually in terms of physical placement of system components. For example, an organization might need to share information that is located on computers at various branch locations, requiring that the computers at these locations be linked. Process control applications are another example in which placement of computers is dictated by the application. Here, processors must be located near the sensors and actuators that interact with

the physical process being controlled. Finally, implementing the fault tolerance needed for high availability requires that when components fail, they do so independently. Processors that are physically separated and linked by a communications network exhibit this failure independence.

A distributed system is a collection of computers interconnected using a computer communications network that provides (relatively) narrow-bandwidth, high-latency communications channels. At first, we might think that an understanding of telephony would be sufficient to enable networks to be implemented and that an understanding of parallel asynchronous processes (from operating systems) would be sufficient to enable applications to be designed for distributed systems. Unfortunately, computer communications demands are not well served by traditional telephone switching technology. In addition, the use of narrow-bandwidth, high-latency communications channels changes the coordination problems that can and must be solved, in addition to changing the costs of various solutions. Computing systems research in networks and distributed systems has tackled these problems, in some cases uncovering fundamental limitations and results in fault tolerance and coordination of communicating processes.

The central proposition of modern data networking is that a useful communications abstraction can be implemented using *packet switching* (as opposed to circuit switching). In packet switching, a communications channel is multiplexed serially by sending small *packets*. Each packet contains routing information and a small element of data. Information to be transferred between two sites is decomposed into packets, each of which independently wends its way to the destination, perhaps even taking different routes. The use of packets permits a higher rate of multiplexing, and hence a high degree of sharing of expensive communications bandwidth. Even telephone companies have now embraced packet switching to implement the circuit-switching abstraction they present to their customers.

Packet switching does not itself prevent the offered load from exceeding available bandwidth; nor does it prevent statistical fluctuations in load from causing short-term overloads. These congestion problems have occupied network designers for the last century. Algorithms designed to control congestion are difficult to construct because congestion might arise and disperse faster than the system can respond. A good analogy here is with air-traffic control. Imagine the difficulty of managing the air-traffic control system if controllers could only communicate by putting messages into airplanes and flying them from airport to airport! Actual air-traffic controllers have access

to low-delay communications paths (e.g., telephone and radio) for control; a computer communications network must use its own data paths for control.

In addition, process execution speeds in a distributed system are high, relative to the speed with which processes can change or sense the state of the system. As a result, system processes can observe the same set of events in different orders, unless those events are causally related. Researchers have developed theories to reason about computing systems subject to this phenomenon, resulting in new abstractions about computation, new applications for logics of knowledge and belief, and even new insight into the nature of causality. Those familiar with special relativity are not surprised to find that time-space diagrams have received application in these theories and that anomalies predicted by special relativity are actually observable in distributed systems.

Another major concern to networking researchers is the correct choice for the communications abstraction provided to users of the network. Raw packet delivery is unsatisfactory as an abstraction because the network may lose or reorder packets. The *virtual circuit* abstraction ensures that packets are delivered in the order sent; unfortunately, it is impossible to implement this abstraction without admitting the possibility of unbounded delivery delays. For this reason, both less powerful and more powerful abstractions have been investigated. Devising suitable abstractions is particularly difficult because of the great variation that can be found in the channels composing a network. Channels can vary in bandwidth and end-to-end delay by as much as six orders of magnitude.

One outgrowth of research in networks and distributed systems is an increased understanding of fundamentals for achieving fault tolerance. In most centralized systems, no serious attempt was made to keep the system running if a component failed, and the whole system failed as one. In distributed systems, it became desirable to keep parts of the system running, even if other parts failed. This has resulted in new methods for partitioning function and responsibility and a new set of abstractions for thinking about fault tolerance.

The traditional view of fault tolerance is in terms of stochastic measures, like MTBF (mean time between failures). A more recent view, t-fault tolerance, characterizes fault tolerance in terms of the maximum number t of failures that can occur before the system will violate its specification. Clearly, stochastic measures can be derived from t fault tolerance if given stochastic characterizations for system component failures. The advantage of t fault tolerance is that it permits evaluation of fault tolerance that is

achieved through design—independent of the reliability characteristics of the components used to implement that design. Fault tolerance can now be viewed in a technology-independent manner. This view of fault tolerance has led to some surprising insights. One is that distributing an input or coordinating the actions of the components in a replicated system can be very expensive and sometimes impossible. Results associated with the so-called Byzantine Generals problem give bounds on the costs. The results also give insight into types of failure that less expensive implementations are unable to tolerate. They show that TMR (triple-modular redundancy), a widely used fault tolerance technique in computer engineering, is based on some previously unstated assumptions that are not always valid.

Computer scientists have enjoyed the benefits of computer networking and distributed computation since the 1960s, when the ARPANet was first constructed. Initially developed as an experimental network connecting computer science research facilities, the network is now widely used by computer scientists (and others) for day-to-day communications activities, including electronic mail, remote login, and file transfer and sharing. Electronic mail has revolutionized communications patterns among users and has led to commercial ventures and standards of use in all disciplines. Office automation and exploitation of personal computers are possible only because of our understanding of computer communications networks and distributed systems. And, mundane as they are, these largely clerical uses, made possible by computing systems research, are changing the way business is conducted, financial resources are managed, and people conduct their daily lives.

Concluding Remarks

Computing systems research, like much of computer science, is about properties and implementation of abstractions. The ultimate utility of these abstractions derives from the extent to which they make computing resources available to applications. A reasonably small set of abstractions—concerned with cooperation and sharing resources—finds utility over and over at various levels of a computing system, leading one to believe that these abstractions are, in some sense, fundamental. New breakthroughs in electronic component technology can be exploited and new applications demands sated with minimal disruption to users' views of computing by virtue of these abstractions.

Although there is a large engineering component in computing systems research—one builds systems in order to understand the utility of the ab-

stractions they implement—it would be a mistake to view engineering as the primary research activity. The main contribution of the natural sciences is to observe, predict, and explain phenomena, not the experiments that validate those explanations. Similarly, the main contribution of computing systems research is the abstractions and our understanding of why they work.

Artificial Intelligence

Drew McDermott Tomaso Poggio

Prior to the invention of the computer, information was something that was transmitted and stored, but it could be processed only when examined by a human being. The human was what made it information, by giving it *meaning*. However, the computer has made it possible to build autonomous, formal agents that process and condense information while respecting its meaning. For example, a payroll program performs a long series of uninspired actions that produce the *right answer*. The *objectification* of information made it conceivable that our original picture of the relationship between humans and information processing could be turned around: instead of humans making information processing possible, perhaps information processing makes humans possible. This hypothesis, that the operation of the mind is to be understood in terms of many small acts of computing going on in the brain, has captured the imagination of an entire generation of researchers since it was first proposed by Alan Turing. Not everyone agrees, however, that all mental activity can be explained in terms of computation, but it is obvious by now that large parts of what the brain does (say, in vision and natural-language processing) can be analyzed as symbolic or numerical computing. It seems pointless to draw boundaries around other parts of the mind where computing must not trespass.

Artificial intelligence (AI) can thus be defined as the science that studies mental faculties with computational models. How much of the mind can ultimately be accounted for this way is as yet unknown. It is important to note that *computational model* does not refer exclusively to a Turing machine or a programming language. Parallel computers, analog networks, and cellular automata represent acceptable models of certain computations. A new branch of psychology, called *cognitive science*, came into being in the late 1950s, inspired by the work of Newell and Simon in AI. The work of the cognitive scientists has helped replace behaviorism with less simplistic models of humans. For example, work on visual imagery, which had all but died out, is now alive and well. Philosophy has been influenced by AI as well.

One of the most debated questions in the philosophy of mind is the status of "functionalism," which explains mental states as analogs of the states of computers. Many philosophers believe that this kind of model explains much of psychology; the debate is whether it is compatible with the facts of consciousness and intentionality. This debate could not be held without the production of actual models of mind.

It is not easy to describe AI tidily. At this early stage, it is not clear whether AI is based on a few fundamental principles or is a loose affiliation of several different subfields, each concentrating on a different part of the mind or on different applications. This report discusses some of the most active areas of AI as examples of the kind of work being done.

Knowledge Representation and Reasoning

One candidate for a unifying principle in AI is the idea of *knowledge representation.* Although in some sense any computer program embodies knowledge (if only of what to do next), AI programs are unique in that they often make *inferences* from complex pieces of knowledge expressed in general notations. The knowledge implicit in a procedure is made manifest only by executing that procedure, whereas knowledge represented *declaratively* as a set of facts is explicit from the start and is accessible in more than one way. Indeed, such knowledge can be assembled, analyzed, and corrected before we have decided upon *any* particular way of using it. We can take as an analogy the laws of Newtonian mechanics, which can be expressed in abstract mathematical equations, and then later applied for many different purposes.

An important requirement for a useful representation language is that the meaning of its sentences should depend *compositionally* only on the meanings of the constituent structures of the sentence and not on the meanings of other sentences or on other surrounding context. As a way of ensuring this property, we are naturally led to the use of *logic-based notations*, in particular, various versions of the *first-order predicate calculus.* Starting in the late 1960s, AI researchers came to realize that computers could not achieve sophistication in various reasoning tasks unless they had formal encodings of large quantities of information about their problem domains that could be processed efficiently. This realization soon led to the elaboration of new problems:

1. What kind of facts can be expressed in formal languages?
2. What is the best method to embody knowledge in data structures?

3. What reasoning algorithms can be brought to bear?

4. How is new knowledge acquired?

Many of these problems now have at least partial solutions, which we discuss briefly.

Concerning the most basic question of what can be expressed in formal languages, experience has been encouraging. There are by now several detailed frameworks for representing general facts about time, physics, and the mental states of agents, as well as more specific facts about medicine, business, geology and other subjects (particularly in the territory of expert systems). It remains to be seen, however, whether these pieces can be put together into a whole that covers a large chunk of human knowledge in a unified way.

It is worthwhile studying formal languages in isolation, but for the computer to make use of them, the formal assertions must be both connected to information stored in data bases and themselves embodied in data structures that summarize relations between assertions. There are now several known ways of doing this, depending on the application. Many of them involve translating logical assertions into systems of nodes, with links between them that can be followed by computer programs to perform inferences efficiently. Such *semantic nets* are also very useful for storing information about hierarchical classifications of objects.

The study of reasoning algorithms has a somewhat different flavor. The initial focus of work in this area was on making deduction more efficient. The result was the discovery of elegant algorithms, based on Robinson's *resolution* principle, and employing the *unification* algorithm. Our understanding of how to carry out deduction has been revolutionized by discoveries like this. But for any given application there are many nondeductive components. Hence, there has been a blossoming of several different reasoning algorithms and an undermining of the notion of a general foundational principle for AI. In practice, each reasoning algorithm follows its own domain-dependent strategies and tends to demand somewhat different knowledge representation techniques. On the other hand, since it would be very useful to have a general theory of reasoning, this state of affairs has exerted a pull on AI theory to come up with broader reasoning algorithms.

One result of the study of reasoning in AI has been the invention of *nonmonotonic logics*, or pseudodeductive systems in which conclusions are revocable given more information. There are several paradigms for accomplishing this extension of traditional deduction. Most of the results are in

what is known as a *circumscription* framework. Circumscribing a predicate P in a theory means adding an axiom schema or second-order axiom to the effect that "Any predicate P' that satisfies the same laws as P and is as strong as P is no stronger." This new axiom allows us to conclude not P in more circumstances than we can from laws for P alone. In semantic terms, it rules out all models of the original facts about P except the *minimal models.* Different versions of the circumscription axiom yield different kinds of minimality. Circumscription is nonmonotonic because adding more laws about P and recircumscribing can eliminate conclusions.

Many of the new reasoning patterns discovered by AI researchers have not been reduced to deduction, and it is not clear whether deduction can be extended to capture them; hence, they must be taken on their own terms. One example is work in *qualitative physics.* Quantitative simulation is in the domain of scientific computing, but human engineers can often predict or explain the behavior of a system without needing detailed numbers describing its components. Elegant methods now exist for predicting, as specifically as possible, the behavior of a system starting from a qualitative description of how its parts interact. One can think of these descriptions in a certain sense as *qualitative differential equations*, which specify the directions in which state variables influence each other but without specifying the magnitudes. The prediction algorithms note the directions in which quantities are changing and the interesting thresholds towards which they are heading. If just one quantity can reach its threshold next, that tells the program unambiguously what the next *qualitative state* of the system will be. Qualitative states can be defined technically as regions in state space in which all quantity-influence relations remain the same. In many cases, the behavior of the system is underspecified, and more than one qualitative state is a possible successor to the current one. The system pursues all possibilities. The ultimate result is a finite graph of qualitative states showing all possible behaviors of the system.

An investigation of the AI literature reveals a multiplicity of knowledge representation reasoning methods. It is not yet clear what unifying principles underlie them, if indeed any do.

Machine Learning

If knowledge, its representations, and reasoning algorithms to manipulate it are indeed central to AI, then the problem of machine learning (the automatic acquisition of new facts and reasoning methods) is crucial. Here, too,

various powerful techniques have been discovered but no unifying principles. In fact, the absence of unifying principles may be counted as a major discovery of AI. The idea that all mental activity might be explainable in terms of learning, in an organism that starts as a *tabula rasa*, has been discredited by the discovery that certain apparently plausible unifying mechanisms are in fact meaningless. For hundreds of years, psychologists and philosophers have thought that the basic mechanism of learning was the transference of successful behavior in a situation to novel but similar situations. When we attempts to realize this idea on computers, we discover that there is no such thing as *intrinsic similarity*. Two situations are similar if some algorithm says they are, and any algorithm must neglect some differences; hence, for any two situations some algorithm will say they are similar, and we are left with the problem of devising algorithms for particular domains. It is now clear that an algorithm for, say, learning cognitive maps will have little to do with one for learning language. There is no choice but to study such problems on their own terms. As a result, in learning as elsewhere, we now know a little bit about a profusion of different learning tasks.

Some general principles have emerged, however. We can make a distinction between *internal* and *external* learning. The former is learning consequences of what we already know, as when we improve our skill at applying methods of symbolic integration. External learning is acquisition of genuinely new facts, as when we learn physical laws through observation. The former can profit from the powerful technique known as *explanation-based learning*. This method consists of extracting from a particular problem-solving session a general principle that will allow similar problems to be solved faster later by skipping over intermediate steps.

For external learning, guaranteed explanations are not obtainable. When learning a new law, a learning program must search through the space of possible versions of the law, trying experiments or making observations to rule incorrect versions out. When the language of the law is simple enough that all possible versions can be expressed as a lattice of more general and less general candidates, then we can keep track of exactly which versions are still viable by keeping track of the upper and lower bounds in the lattice within which the correct version lies. As more observations come in, they can be used to narrow the bounds. When applicable, this idea allows a "binary search" through the set of candidate laws.

Computer Vision

Not all subfields of AI are oriented directly around knowledge manipulation. Computer vision, the attempt to understand how information can be extracted from the light bouncing off objects, is a good example. As we will discuss later, vision algorithms must embody a lot of knowledge about optics, but they do not need to represent it declaratively. This distinction has not prevented the field of computer vision from developing some of the most satisfying results in AI. Before computational methods were brought to bear, vision theory had progressed little beyond optics. Electrodes could be stuck into cells in the visual cortex, but their signals were generally a mystery. Since approximately 1970, vision researchers have produced a plethora of detailed models of different aspects of vision. Many workers believe that the job of the visual system is to build a symbolic description of what it is looking at, and the role of computer science is to tell us what a symbolic description is. We may not know where to locate it in the brain, but we know we are looking for it.

Problems in vision are usually classified as part of *low-level* (or "early") *vision* or part of *high-level vision.* Early vision performs the first steps in processing images through the operation of a set of visual modules such as edge detection, motion, shape-from-contours, shape-from-texture, shape-from-shading, binocular stereo, surface reconstruction, and surface color. Its goal is to yield a map of the physical surfaces around the viewer. High-level vision can be identified with the "later" problems of *object recognition* and *shape representation.* Here, questions of knowledge representation will enter in an essential way.

The problem of vision begins with a large array of numbers recording an intensity value for each pixel (picture element) in the image. The precise value at each pixel depends not only on the color and texture of the three-dimensional (3-D) surface that is reflecting the light but also on the orientation and distance of the surface with respect to the viewer; on the intensity, color, and geometry of the illumination; on the shadows cast by other objects; and so on. The goal of early vision is to unscramble the information about the physical properties of the surfaces from the image data. In a sense early vision is the science of inverse optics. In classical optics (or computer graphics) the basic problem is to determine the two-dimensional (2-D) images of 3-D objects, whereas vision (whether biological or artificial) is confronted with the inverse problem of recovering 3-D surface from 2-D images. In color sensing, for instance, the goal of vision is to decode the

measured lights in terms of the reflectance of the surfaces and the spectral power distribution of the illuminant.

The problems of inverse optics are very difficult to solve, despite the apparent ease and reliability with which our visual system gives meaningful descriptions of the world around us. The difficulty is at least twofold. First, the amount of information to be processed is staggering: a high-resolution television frame is equivalent to 1 million pixels, each containing eight bits of information about light intensity, making a total of 8×10^6 bits. The image captured by the human eye is even more densely sampled, since in the human eye there are in excess of 100 million photoreceptors. Real-time visual processing must be able to deal with many such frames *per second.* It is therefore not surprising that even the simplest operations on the flow of images (such as filtering) require billions of multiplications and additions per second. Second and more important, the images are highly ambiguous: despite the huge number of bits in a frame, it turns out that they do not contain *enough* information about the 3-D world. During the imaging step that projects 3-D surfaces into 2-D images, much information is lost. The inverse transformation (from the 2-D image to the 3-D object that produced it) is badly underdetermined.

The natural way to approach this problem is to exploit a priori knowledge about our 3-D world to remove the ambiguities of the inverse mapping. One of the major achievements of computer vision work in the last decade is the demonstration that *generic* natural constraints (the term generic is used here in the same sense as in the mathematical theory of dynamical systems) that is, general assumptions about the physical world that are correct in almost all situations are sufficient to solve the problems of early vision; and very specific, high-level, domain-dependent knowledge is not needed. Two main themes are therefore intertwined at the heart of the main achievement of early vision research: the identification and characterization of generic constraints for each problem and their use in an algorithm to solve the problem.

Some of the most powerful constraints reflect generic properties of 3-D surfaces. One of the best examples is the recovery of structure from motion. Perceptual studies show that a temporal sequence of images of an object in motion yields information about its 3-D structure. Consider for instance a rotating cylinder with a textured surface: its 3-D shape becomes immediately evident as soon as rotation begins. It has been proved that a 3-D shape can be computed from a small number of identified points across a small number of frames — if one assumes that the surface is *rigid.* Vari-

ous theorems characterize almost completely the minimum number of points and frames that are required. *Continuity* of surfaces is another useful assumption: surfaces are typically regions of coherent aggregates of matter, do not consist of scattered points at different spatial locations, and are usually smooth. These constraints are very powerful for solving the correspondence problem in stereo and motion and for reconstructing surfaces from sparse depth points. Of course, constraints of this type are occasionally violated, and in these cases algorithms that strictly enforce them will suffer from "visual illusions."

It is natural to ask whether a general method exists for formalizing constraints in each specific case and translating them into algorithms. An interesting answer to this question has emerged in the last two years. We will describe it from a representative point of view, though by no means the only possible one. This unifying theoretical framework is based on the recognition that most early vision problems are mathematically ill posed problems. A problem is well posed when its solution exists, is unique, and depends continuously on the initial data. Ill posed problems fail to satisfy one or more of these criteria. In vision, edge detection (the detection and localization of sharp intensity changes) is ill posed when considered as a problem of numerical differentiation, because the result does not depend continuously on the data. Another example is the reconstruction of 3-D surfaces from sparse data points, which is ill posed for a different reason: the data alone, without further constraints, allow an infinite number of solutions, so that uniqueness is not guaranteed without further assumptions. The main idea in mathematics for "solving" ill posed problems (i.e., for restoring them to well posed problems) is to restrict the space of admissible solutions by introducing suitable a priori knowledge. In vision, this is identical to exploiting the natural constraints described earlier. Mathematicians have developed several formal techniques for achieving this goal that go under the name of *regularization theory.*

In standard regularization the solution is found as the function that minimizes a certain convex functional. This functional can be regarded as an "energy" or a "cost" that measures how close the solution is to the data and how well it respects the a priori knowledge about its properties. Consider the direct problem of finding y, given z and the mapping A:

$$Az = y$$

The inverse and usually ill-posed problem is to find z from y. Standard regularization suggests transforming the equation into a variational problem

by writing a cost functional consisting of two terms. The first term measures the distance between the data and the desired solution z; the second term measures the cost associated with a functional of the solution $\|Pz\|$ that embeds the a priori information on z. In summary, the problem is reduced to finding z that minimizes the quantity

$$\|Az - y\|^2 + \lambda\|Pz\|$$

where λ, the regularization parameter, controls the compromise between the degree of regularization of the solution and its closeness to the data. Mathematical results characterize various properties of this method such as uniqueness and behavior of the solution. Solutions of this type have been obtained for several early vision problems: edge detection, optical flow, surface reconstruction, spatiotemporal approximation, color, shape from shading, and stereo.

Computer vision has always had a special two-way relationship with brain sciences: suggestions from visual physiology and psychophysics have played a role in many developments of computer vision. For instance, discoveries of neurons that seem to behave as edge detectors in the visual cortex of primates had a significant influence in the development of early computer vision programs. In turn, computational theories of vision are now influencing the psychophysics and the physiology of vision. It is very likely that this trend will grow more important for both fields. Mainly because of the theoretical advances of the last decade, it seems that early vision is now on its way to a systematic solution. Much less has been accomplished in high-level vision, however. At the level of object recognition and scene description, the vision system begins to blend with the rest of the mind, about which elegant unifying theories do not yet exist.

Concluding Remarks

In summary, AI is in a way the branch of computer science that is most nearly a classical empirical science. It studies the world at the computational level, in much the same way that chemistry studies the world at the chemical level. It is not a priori obvious that there *is* a chemical level; in principle, everything is just physics. But in many situations it is possible—and necessary—to ignore the details of elementary-particle interactions and focus on interactions in terms of *molecular bonds*, *valence*, *stoichiometry*, *reaction rates*, and so on. Similarly, in principle, the brain's functioning can

be explained in terms of the behavior of its neurons and their membranes. Attention should be focused not only on these details, but also on the *information* that the neurons are transmitting and the *computations* they are doing. If this is approximately correct, then it may be just as necessary to focus on this higher level in understanding the brain as it is to focus on the chemical level in understanding chemical systems.

The nervous system is not the only place in the universe where nature has exploited computation. Another good candidate is the operation of the cells of organisms. Although in principle the behavior of DNA is describable chemically, the important thing about a particular DNA molecule is the message encoded in its nucleotides; this message is completely *arbitrary* from a chemical point of view. In many cases the only reasonable way to describe the operation of a cell is at a computational level in which genes are thought of as switching each other on and off, so that the set of active genes behaves like the state of a computing device, the next state and the outputs (proteins) being functions of the current inputs and the previous state. The study of such molecular computers—if they really do exist—might or might not be assimilated to AI. Indeed, it is not clear whether in the long run AI will be stable as a single discipline or split up along mental-module boundaries that are yet to be discovered. The point to absorb is that computation appears to exist in nature as well as in artifacts; its study is now emerging as a new empirical science.

Applications of Computer Science

Edward Feigenbaum Richard F. Riesenfeld
Jacob T. Schwartz Charles L. Seitz

As stressed in the preceding chapters, computer science has developed concepts and techniques that begin to compare in depth and virtuosity with the best that much older and better established disciplines (e.g., mathematics) have to offer. Nevertheless, computer science remains, as its name proclaims, an applied science. This is to say that computer science is a discipline in which the influence of both technology and application is particularly strong, in which theory needs to be guided not only by aesthetic concerns but also by vital (though possibly indirect) connections to practice, and in which the ability of practice to fructify theory (a fundamental influence even in mathematics, as emphasized by John von Neumann) is particularly vivid and direct. The present chapter of this report will illustrate this point by tracing the relationship between computer practice and theoretical research in a few salient cases. Our aim is to show both the significant way in which research has been able to contribute to practice and the manner in which practical experiments and problems have brought basic questions to the attention of the research community.

VLSI design

It is commonplace that the steady (yet amazingly rapid) evolution of microelectronics technology is a key driving force of the computer age. Continuing development of the technology of very large scale integration (VLSI) has required not only steady improvement in the physical aspects of integrated circuit manufacture (especially miniaturization, reduction of power consumption, and sophisticated packaging) but also continuous innovation in design and verification methods for these miniaturized but extremely complex systems. Indeed, by the mid-1970s the very large number of circuits that could be integrated onto a single digital integrated circuit ("chip") meant that VLSI technology was as fundamentally limited by design costs as by

problems of semiconductor device fabrication. For this reason, the many techniques for complexity management and design verification developed by computer scientists for large software systems have suggested complexity management tools that are now crucial for continuing improvements in VLSI design.

The most complex chips of a decade ago typically required a year to design, using modest computing aids for graphical assistance in their physical design or layout. Today, even though VLSI chips have become as much as 100 times more complex than those produced only a decade ago, many chips can be designed largely in an automatic way from high-level specifications. Moreover, even where this remains impossible, the design of at least sections of complex chips is supported by software tools that greatly simplify such important design tasks as optimization of switching functions, assembly of logic arrays, circuit placement, power routing, and wire routing. These design tool advances, and the improved productivity that they permit, exploit efficient new design algorithms and couple these algorithms into interactive design environments.

Equally important advances have been made in simulation and verification techniques. Trying, after chip fabrication, to locate design errors that cannot be sensed directly owing to the very small internal signal energies involved is an all but hopeless task. Instead, VLSI chips are now designed using formal construction rules and design disciplines that both minimize errors and exclude constructs that would make it difficult to verify or to simulate chip behavior. In addition, static or "syntactic" checks are applied to VLSI designs to verify compliance with geometrical and electrical design rules and to pinpoint paths critical for timing. It is important to emphasize that the algorithms used for these tasks exploit geometric and combinatorial computation techniques supplied by the *basic research community.* One significant new simulation technique is event-driven switch-level simulation, which, since it is much less computationally demanding than circuit-level transient analysis, allows complete VLSI chips to be simulated. As a result of many individual improvements in VLSI analysis tools, complex chips are expected to function correctly, and usually do, on the first test after being fabricated.

In addition to these vital pragmatic contributions, theoretical research has clarified our understanding of the ultimate factors that constrain VLSI technology because they constrain the power of any computing apparatus of given density and volume. Once appropriate theoretical models characterizing the fundamental physical resources required in VLSI design were

defined, it became possible to establish results of this kind using estimation techniques pioneered in computational complexity theory, the study of the ultimate performance of algorithms. However, significant extensions of these older ideas were required. The complexity theory developed for the study of algorithms executed on sequential computers regards the number of operations and storage space required by a program as fundamental resources but neglects what on a chip is the most expensive part of a computation, namely communication. In VLSI, it is the *area* of the wires on a chip and the *time* and *energy* required to change their state that dominate the cost and performance of computations. All computing technologies that operate close to physical limits will be most seriously limited by communication, and so they must be treated explicitly as "distributed" systems in order to be successfully modeled.

This important architectural conclusion emerged clearly from the first theoretical studies of the limits of VLSI technology, which analyzed the way in which communication between the processing elements on a chip limits the extent to which chip area A and processing time T required for a computation can be simultaneously minimized. Basically, these results state that AT^2 cannot be less than a function determined by the theoretical nature of the problem to be solved and by the technology used. For example, in multiplying two n-bit binary integers, area and time together are bounded with AT^2 growing at least as n^2.

Theoretical insights such as these have allowed designers to devise ideal layouts that attain theoretical performance limits and provide useful guidance to practical design efforts. For example, such AT^2 results explain why it has proved so costly to push high-speed arithmetic design by making computers perform *single* operations as rapidly as possible. They also show why it can be less costly for computers to perform many operations concurrently. The cost of a computation such as a multiplication can be viewed as using ("renting") an area A of silicon for time T, which has a cost proportional to AT. The AT^2 lower bound on computational cost shows that the cost of a single multiplication is expected to vary as T^{-1}. It follows that a multiplication need not cost as much if you are not in a hurry to have its result. Hence, if N multipliers are used in parallel, computations involving many multiplications that can be performed simultaneously (as in computing a Fourier transform) can be performed N times faster at a given cost or in a given time at a cost reduced by the factor N, compared to single multiplications performed at maximum speed. This comparison illustrates the point that computations that can use many "parallel" or "concurrent" threads

of control are fundamentally less costly than computations in which each intermediate result depends on a previous result. Many of the computing problems discussed in this report, as well as many other scientific and engineering computations, allow such parallel execution and can be expected to exploit parallel machines as these become available.

Thus, VLSI theory has given computer scientists a fundamental reason to develop highly parallel computing systems, and VLSI technology has made such machines practical. In consequence, during the last few years increasingly more and more highly concurrent commercial computers have appeared; they include message-passing "hypercube" multicomputers, systolic arrays, and large fine-grained systems such as the connection machine. It is not surprising that most of these systems originated from university VLSI research, specifically at university computer science departments involved both with technology and with theory. Thus, the contributions of basic computer science (as distinct from physical device research) to real-world VLSI practice have been very substantial. However, in so deeply technological an area as VLSI design it would be a mistake to ignore the major infrastructure development that this advance has required. The ability of universities to prototype chips economically has been vital to their creative involvement in VLSI practice and theory. Currently, this capability is supported principally by the "MOS Implementation Service" (MOSIS) project at the University of Southern California Information Sciences Institute under Defense Advanced Research Projects Agengy and National Science Foundation sponsorship. Designs are sent in standardized interchange formats by electronic mail to MOSIS, where hundreds of designs from tens of organizations are automatically batched together for realization on the semiconductor wafers that are the basic handling unit of microelectronic fabrication.

Computer-Aided Graphics and Geometric Design

Construction of an animated graphic image (in an array of 1024 by 1024 pixels) requires the computation of 1 million items of information updated at least 15 times per second to create an impression of smooth motion; thus, computer graphics has always had a large appetite for raw computer power. For this reason, the VLSI developments summarized in the preceding section can be expected to extend substantially the availability and quality of computer graphics over the next few years, ultimately approximating resolutions (in arrays of 2000 by 2000 pixels) at which computer-generated images will be hard to distinguish from photographs. However, the data structures

used to define and display graphics images can be very complex, and access to them is required to be performed in real time. Accordingly, graphics programs are often large and complex, making their realization dependent on the most powerful available programming languages and methodologies. Moreover, since image generation can often be performed in parallel (e.g., for the 1 million pixels of a 1024 by 1024 image), graphics programmers have been actively involved in parallel and distributed programming, and graphics applications help spur the current high level of interest in parallel machine architectures.

Very practical problems of computer graphics have raised subtle issues for the algorithm designer. For example, to construct the image of an artificial graphic scene containing multiple objects, we must distinguish those object surfaces that lie behind and are hidden by other objects from those surfaces that are visible from a given viewpoint and need to appear in the image being constructed. This is the *hidden surface problem*, a question entirely unanticipated before being encountered by early computer graphics experimenters. Its study has raised very interesting problems and has led to significant theoretical developments. For example, the hidden surface problem sometimes must be solved for complex artificial scenes containing hundreds or even thousands of separate items. Imagine a scene containing many spheres with known centers and diameters, many of the them entirely hidden behind combinations of spheres. How rapidly can these hidden objects be identified and eliminated from the construction of the scene? Computational geometers have found more and more sophisticated and efficient techniques for handling potentially important graphics-related problems of this kind.

Graphics is also necessary for the closely related subject of *computer-aided geometric design*, which provides tools that the engineer and industrial designer can use to define the shapes of products that are to be cut out of metal, molded, or assembled. This practical activity leads at once to further challenging theoretical problems. For example, how best can *free-form* surfaces be represented within a computer? This problem, motivated by the need to describe the exterior surfaces of aircraft, has been under active investigation since the early 1960s. A general class of surfaces (subsequently known as Coons patches), which interpolates between four arbitrary boundary curves that can be specified with considerable generality, was defined. These "patches" became the object of considerable study and formal analysis.

The polynomial approximation ideas implicit in Coons's work relate closely to *spline approximation methods*, initially developed by numerical analysts in the one-dimensional case but recently extended to two (and more) dimensions, partly for use in geometric modeling and computer graphics. *Multivariate* spline theory has become an important specialized area of numerical analysis, with strong geometric overtones. Not only have theoretical studies of multivariate splines been influenced by needs and insights arising in geometric modeling, but geometric modeling programs also have furnished useful tools for the spline theorist. The mathematical constructs used to define multivariate splines can be abstract and complex, leading in directions that are difficult to visualize without sophisticated graphic tools, which assist significantly in the comprehension of the behavior of functions, geometric structures, and mechanisms. Thus, the requirements of geometric modeling have moved spline theory in new directions, which graphic modeling tools have made easier to explore.

An important technique for modeling complex bodies is to construct them as *Boolean combinations* of basic geometric forms, allowing the free addition and subtraction of forms such as solid spheres, cylinders, and polyhedra. Extensions of these techniques, however, to handle surfaces of freer forms, such as 2-D spline surfaces, pose deep problems in computer algebra, numerical analysis, and data structure design. In particular, if careful attention is not focused on questions of computational robustness, computations can easily become unstable. Even small numerical errors in the calculation of surface intersections can cause the outside of a 3-D figure to be confused with its inside, leading to wildly erroneous values for the volume of the figure. Other less catastrophic numerical errors can lead to miscalculation of hidden surfaces, resulting in production of images with artifacts or even paradoxical, bizarre graphic images.

Other computer graphics problems arise from the necessity to transform the results of geometric computations performed for continuous bodies into discrete pixel-intensity arrays ready for display. If this is not done properly, odd-looking sampling artifacts (the so-called staircase effect or jaggies) can appear. Originally it was naively assumed that acceptable pictures could be generated if one applied computing power sufficient to calculate an intensity value at the center point of every pixel in a sufficiently fine pixel array. Experience proving this to be false triggered many theoretical and empirical studies of the *anti-aliasing* problem. This development has improved considerably our understanding of the lighting models required to produce realistic-looking synthetic images and, more generally, of the psychology of

image perception. Exposure to intriguing visual phenomena appearing unexpectedly in the course of extensive graphic experimentation was essential to this progress.

Graphic modeling and display technology interacts in essential ways with other applied research areas reviewed in this report. For example, appropriate graphic display is vital to understanding the results of large scientific computation. In raw numerical form, the result of simulating the flow of air around an aircraft hull is an enormous and bewildering array of numbers; reduced to graphic form, the structure and salient features of the flow (and even obvious program errors) become much clearer. Robotic applications also furnish many other illustrations. The practical use of the geometric motion-planning techniques reviewed subsequently requires use of geometric modeling systems to prepare the data that motion planning routines require. As these and many other examples show, research in graphics display and modeling has significantly increased the effectiveness of other computer research and applications efforts.

Scientific Computing

Scientific computing draws out the consequences of basic physical laws by embodying them in large-scale numerical *computer simulations*, often requiring massive computing power. The practical importance is very great, since numerical simulation is now routinely used in such crucial engineering tasks as studying the stresses in reactor containment vessels, the deformation of metal under high-pressure molding, and the way in which atmospheric motions and sea currents interact to shape the world's climate. These are problems that are difficult or impossible to investigate empirically. Success in the endeavor requires the right combination of appropriately simplified mathematical models, numerical procedures, and fast and flexible software. Since assembling all this for application to large problems is normally too much for the individual researcher, scientific computing projects have often required collective effort by computer scientists, mathematicians, scientists, and engineers. The efficacy of these efforts has been much enhanced by the development of first-class software libraries and by a well-established tradition of software sharing, which can stand as a model for other areas of computing.

Consideration of just one fundamental computational procedure, namely the solution of *systems of linear equations*, can serve to illustrate some of the ways in which computer science research has contributed to the more

applied activity of scientific computing. Many physical systems are linear, in the sense that their response to the sum of two external influences is the sum of the separate responses that could be generated by presenting each of the two influences separately. Moreover, the analysis even of nonlinear systems, which do not behave in so simple a fashion, tends to begin with analysis of linear approximations, because the sufficiently small deviations of nonlinear systems form an initial state of equilibrium that will almost always be linear. Consequently, once the physical equations for a system have been discretized to prepare them for computational treatment, it is often desirable to solve some system of linear equations that can be written in matrix notation simply as $Ax = b$. However, since the size of the matrix A defining such a system can be very large, the ordinary, largely qualitative, mathematical analysis (by linear algebra) of the properties of such systems does not provide adequate guidance for their numerical solutions. It was necessary to develop techniques that solve such systems by exploiting the special structural properties of important special cases. Such structures arise for various reasons, such as the locality of the physical force systems that the equations represent, from conditions of invariance, or from geometric considerations. The field of numerical linear analysis on which we touch here is vast, and it is impossible for us to refer to more than a tiny sample of the results obtained; nevertheless these very few examples serve to typify the essential interaction of this field with computer science research.

A major task of scientific computing is to solve *partial differential equations*, linear and nonlinear. There are two basic ways to make such a problem discrete. One is to cover the domain in which the equation is to be solved by a finite mesh of *points* and replace derivatives by differences; this is the so-called *finite difference method.* The other is to choose a finite number of *functions* and approximate the desired solution by a combination of those trial functions. In the *finite element method* these approximating functions are polynomials of low degree. This technique now has become very popular, especially for large structural problems in engineering. For *fluid flow problems*, other methods are active competitors: finite difference methods, so-called spectral methods (discussed in more detail later), vortex methods, and recently developed cellular automata techniques, which exploit large-scale parallelism in particularly effective ways.

Each of the trial functions used in a finite element solution of a numerical problem is nonzero over some limited domain within the overall region in which the solution to a given numerical problem is sought. The pattern in which these domains subdivide a region is normally one in which each sub-

region contacts only a few other subregions. For this reason, the matrices A that appear in the linear system $Ax = b$ to which a finite element procedure gives rise are generally sparse (each row of A contains very few nonzero elements). Systems with this property can often be solved most effectively by eliminating successive variables in an order chosen to introduce as few nonzero coefficients as possible in the resulting series of linear systems of progressively smaller size. Effective *elimination orders* of this kind can be found by analyzing the graph defined by the pattern of nonzero entries in the original matrix A. The efficient graph-analysis algorithms devised for this purpose exploit sophisticated graph algorithms developed during more than a decade of work on graph-related computational procedures.

In the analysis of a large, complex engineering structure (a ship's hull or the steel frame of a bridge) for finite element analysis, even the initial setup of the large system of linear equations with which numerical analysis will begin is a major task. Its automation requires the development of systematic procedures for decomposing general 3-D regions into an arbitrarily fine mesh of regions of simple shape (say, small triangular pyramids). This also applies to many other numerical techniques for dealing with regions whose boundaries are irregular (or variable, as in tracking the surface of an expanding bubble of steam within water or water within oil). For this reason, the large finite element codes based on current engineering practice need to interface to geometric design systems like those discussed in the preceding subsection. They also need to incorporate sophisticated procedures for automatic triangulation of complex geometric regions. These requirements link engineering practice very directly to some of the most ambitious current research efforts in computational geometry and data structure design.

A second important strategy for dealing with numerical problems that originate in geometric contexts is *recursive subdivision.* In this approach, one divides the region in which the problem is originally stated into a small number of subregions, recursively solves the numerical problem for each of these subregions, and then integrates the resulting partial solutions into an overall solution for the region by solving a set of auxiliary problems to match values along all the boundary cuts separating adjacent pairs of subregions. This computational approach, which has become quite important, has profited from sophisticated theoretical analysis of the combinatorics of planar graphs, which have demonstrated that favorable subdivisions of arbitrary plane patterns can always be found.

Whenever a linear physical process being studied numerically is invariant in space or invariant in time, Fourier analysis applies, making analysis

of the system in terms of its resonant modes possible. This is called spectral analysis, and the so-called fast Fourier transform accelerates all such calculations in a sensational way. Its availability, and progressive refinement through a continuing series of studies that have significantly reduced the cost of Fourier-related computations, has revolutionized many fields from signal processing to radio astronomy.

The new possibility of doing thousands of calculations in parallel is influencing numerical scientific computation profoundly. To exploit this opportunity effectively will require the best talents of numerical analysts, computer scientists interested in parallel algorithm design, and computer hardware architects. Since many of the best current numerical techniques are serial in form and, thus, not obviously suited to the aim of sending whole rows of matrix computations directly through systolic arrays of parallel processors, much penetrating inquiry will be required to handle trade-offs between ingenious processing order and brute-force computation effectively. The advent of parallel machines also makes it necessary to rework the large numerical code libraries on which scientific computation depends. Thus, we stand on the threshold of an exciting period in which the connections between scientific computing and computer science will be reaffirmed and strengthened.

Robotics

Robotics is another field in which sophisticated algorithmic techniques supplied by basic computer science research play a steadily increasing role. *Robots* can be defined as computer-controlled devices that reproduce human sensory, manipulative, and self-transport abilities well enough to perform useful work. Presently, both the sensory capabilities of robots and their ability to deal with unexpected events are quite limited. For this reason, today's robots are effective only in highly structured, largely industrial environments in which the position and path of motion of all the objects are known accurately at all times. Research during the past decade has aimed to relax this restriction, in order to make robots usable in less predictable environments. To achieve this, the robot's user must be able to specify the behavior of the robot in a wide variety of environments in general terms, and then have the computer controlling the robot fill in missing lower-level details automatically.

By simplifying the otherwise onerous task of specifying many detailed robot motions, automation of this basic procedure and others like it can contribute significantly to the efficiency and speed with which industry can

introduce robots into manufacturing. For example, we should be able to specify the product of some assembly process and ask the system to construct a sequence of assembly substeps. At a less demanding level, we would like to ask a robot to plan collision-free motions that involve picking up the individual subparts of an object to be inserted, transporting them to their assembly positions, and inserting them into their proper places. Deep and interesting research issues have been brought to light by work on these practical goals. For example, the geometric part of the second of the two tasks defines the so-called problem of *automatic motion planning*, which has received much attention during the past few years. Studies in this area have shown it to have significant mathematical content: tools drawn from classical geometry, topology, algebraic geometry, and combinatorics have all proved relevant. Particularly close relationships have developed with research in computational geometry. Although space limitations preclude any full account of the extensive work that has been done in this intriguing area, it is worth summarizing some recent investigations of the motion planning problem, with the intent of indicating the surprising depth of the algorithmic ideas that have found application to questions that may at first glance appear shallow and purely pragmatic.

The simplest form of the motion planning problem follows: given a known initial position and desired final position of a rigid robot R moving in an environment full of obstacles, decide whether there exists any continuous obstacle-avoiding motion that can take R from the specified initial position to the specified final position; and, if such motions exist, produce one of them. The intrinsic difficulty of this problem is suggested by the complex series of motions sometimes required to move an inconvenient object, like a table or a long ladder, along a winding corridor or up a narrow stairway. Of the many approaches to this problem that have recently been considered, we will note here only the so-called *retraction technique*, developed during the last three years. The central idea of this method is as follows: suppose there does exist a path P connecting the specified initial and final robot positions. One can take each position $P(t)$ that occurs along the path P, find the obstacle O to which this position is closest, and push the robot R away from the closed obstacle O, for example by translating R in the direction opposite to its lines of closest approach to O without allowing it to rotate. Pushing can continue until the robot reaches a position equidistant from at least two separate obstacles. It is not hard to guess that, by applying this operation in a uniform way to every one of the positions $P(t)$ that occur along the path P, we can transform P into a path that (aside from initial and final

phases of motion directly away from and toward the obstacles closest to the specified initial and final positions I, F) consists entirely of positions that are *simultaneously closest to at least two obstacles.* In more intuitive terms, these are paths that try to stay as far as possible from the obstacles by always *remaining midway between the two closest obstacles.*

This approach to motion planning generates some of the most efficient planning procedures known. When developed into a detailed algorithm, it makes use of important ideas that arose independently and earlier in pure computational geometry, namely the notion of *Voronoi diagram.* In its original 2-D plane incarnation, this diagram is defined as the partition of the plane into a set of regions $N_1, \ldots, N_k$, where N_j is simply the set of all points p closer to a particular point p_j than to any other point in a given set $\{p_1, \ldots, p_k\}$ of points. In the more general situation arising in motion planning by the retraction approach that we have outlined, we instead divide the (multi dimensional) space S of all positions of a moving robot R into the set of regions $N_1, \ldots, N_k$, where N_j is defined to be the set of all positions p in which R is closer to the obstacle O_j than to any other obstacle.

The computational cost of a path-finding procedure based on this approach will clearly depend on the *size* of the Voronoi configuration defined by a set of k geometrically simple obstacles. Study of this geometric question has led to a whole series of geometric studies and problems. In this way, the practical, ultimately industrial, problems of robotics have suggested new theoretical issues to geometers, partially repaying them for the important contributions they have already been able to make.

Expert Systems

Historically, one of the most important motivations for AI research has been the creation of programs that solve problems of considerable intellectual difficulty at high levels of competence. Partly this is an engineering activity, involved with the construction of intelligent artifacts; partly, like all good engineering activities at an early stage in the scientific development of an area, it has given rise to scientific endeavors of importance.

Starting in 1965 with the creation of the DENDRAL program for the elucidation of organic chemical structures, a class of programs called *expert systems* arose. These computer programs consist of two parts. One is the *knowledge base*, a collection of data structures in which the task-specific knowledge of the domain of discourse is represented and stored. The knowledge base contains not only the facts relating to the domain but also the

heuristics of expert-level performance (the experiential, judgmental knowledge that reflects "the art of good guessing" for problem solving in the domain). The other component is the *inference engine*, the collection of reasoning methods used to construct lines of reasoning leading to the solution of problems, formation of hypotheses, satisfaction of goals, and so on. These reasoning methods are drawn from symbolic logic or from work on problem-solving methods done by AI researchers.

At present, the modeling of expertise (i.e., the building of an expert system) is primarily an activity of capturing and representing the knowledge that human experts have, and only secondarily is it involved with capturing the reasoning methods that experts use. Priority has been given to the problem of automatic acquisition of the domain-specific knowledge. As usually done in expert system construction, this is a cumbersome, expensive, time-consuming process. Scientific research over the past decade has yielded results in this form of machine learning, particularly a variety of induction methods that are driven by examples, by user interactions, and by basic principles of the task domain. All are grounded in the principle that automatic knowledge acquisition is itself to be viewed as a knowledge-based task. The next phase of development will tackle the many difficult problems of inference.

Expert system applications are already having considerable economic payoff, and their importance will grow considerably in the years ahead. We need the inexpensive replication of an otherwise scarce resource: human expertise. But more important, an expert system can process lines of reasoning and use a body of knowledge about an area far more systematically than people can. In such areas as diagnosis, we can expect that expert systems will improve substantially the quality of medical care by giving primary care physicians quick and reliable access to tertiary care expertise. We can also expect expert systems in other scientific fields to propose hitherto undiscovered hypotheses and theories and to propose important novel designs and solutions. As the sophistication of these systems grows, they will become more and more tools of research.

Initiatives Report

Prepared by

Kenneth W. Kennedy
Clarence A. Ellis
John E. Hopcroft
Burton T. Smith

Introduction

As a part of the assessment of Computer Science and Engineering Research and its needs for the immediate and long-term future, the Computer Research Advisory Committee has been considering potential initiatives and research opportunities that should be given special attention over the next five to ten years. In addition to identifying the areas, our report will spend some time addressing the problem of how best to foster the kinds of advances needed in these areas.

The committee is agreed on three areas that should be the subject of special initiatives: software engineering, parallelism, and automation. Each of these initiatives is involved to some extent with productivity—productivity in software development, productivity of machine execution and productivity in manufacturing. The following sections explore the fundamental issues to be addressed in these research initiatives.

Software Engineering

Software engineering is concerned with the conception, specification, design, analysis, implementation, and maintenance of large complex software systems. There are well-documented problems of size and complexity that form the core of the software engineering problem domain. In particular, as the size of software systems grows very large, it becomes virtually impossible with current technology to attain acceptable levels of reliability, performance, correctness, cost, and so on. The size of software systems which we attempt to develop is continually increasing; likewise, the urgency of need for solid research targeted at the above problems is continually increasing. Therefore we advocate expanded research into foundations and methods for development of large complex software systems. An example emphasizing the previously stated problems is the recent public debate over software engineering and the SDI research program. Regardless of individuals' stances on this issue, the debate underscored the extreme importance of research to stimulate marked improvements in the software engineering area. Thus, the committee feels that there is opportunity for dramatic progress in this area.

Therefore we propose a major initiative in software engineering with the goal of researching foundations and methods that will significantly improve software engineering productivity and quality over the next decade. To achieve this goal, we will need to bring the best university and industry research to bear on the problems and make sure that the results of that

research are widely disseminated to stimulate further research, and where appropriate, rapidly moved into production use.

This research should be directed toward developing theories, techniques, and environments that directly apply to very large systems. There is a need for theories that take into account in a unified manner the various properties (reliability, performance, correctness, etc.) of systems and the various techniques (structured analysis, top down design, etc.) of software engineering. Likely existing properties and techniques need to be rethought and amended for consistency, completeness, and especially for scalability to very large systems. Likely new modularizations and layerings that encourage integrated design and implementation are needed.

Techniques that might be developed within this initiative would allow software engineers much needed assistance in designing, implementing, maintaining, and reusing large systems components. A primary element of software development is analysis, which needs uniform techniques so that it can be applied evenly throughout the phases of the software development process. Analysis means the critical examination across a range of properties of essential features of the artifacts produced throughout the software process including requirements, specifications, performance models, code, and simulations.

The process of software engineering should proceed within a unified computationally adequate software environment. An environment must provide a uniform user interface, a single cognitive system model, and a smooth means for a set of tools and techniques to integrate into a development model or theory. Future environments have an opportunity to exploit advanced architectures such as parallel processors and networks, and exploit the power that exists in modern database technology, object oriented paradigms, AI, and other advancing technologies of computer science. A major question here is how these technologies can be specialized (or in many cases enhanced) to fulfill the requirements of a software engineering environment.

The challenge is clear, and the research work to discover solutions to these software engineering problems will require creativity and sensitivity to the social as well as technical aspects of large systems development.

Unfortunately, software engineering research in universities in the past has frequently been hampered by the absence of large-scale implementation projects of the kind needed to provide realistic testbeds for research. On the other hand, research in industry has frequently been too much influenced by the problems of getting a product "out the door" to perform significant research on software engineering theories and technologies. Some of the

research suggested by this initiative will require NSF to fund software development projects at universities, and the construction of large experimental software systems that go well beyond the typical university research structure of a single principal investigator plus graduate students. These grants may remove some of the pressure from the CER program, permitting that program to restrict itself to building infrastructure.

Other structures that may form and should be encouraged include the coupling of university researchers with industry where large-scale software engineering is being performed, and university consortia (see, for example, Arcadia —the Ada Environment Research Consortium). These structures may help to build a repository of experience with implementation of large systems within research institutions and allow experimentation with large systems.

The primary thrust of this initiative is software engineering, a multidisciplinary area. A successful initiative in software engineering will, to an extent, involve research in a variety of areas. We list a small sample here.

- *Design Languages.* Innovations in programming language design will be required to permit the programmer to operate at a high level of abstraction. Progress will be needed in design languages, specification languages, and languages for rapid prototyping.

- *Design and Programming Environments.* Automatic support for every phase of the software design process, from conceptualization to maintenance, will need to be integrated into the programming environment. Tools in the environment will need to cooperate to assist the designer in producing and maintaining a correct, reliable system.

- *Programming Methodology and Reusability.* At present, most software is written from scratch because programs are difficult to read and strongly dependent on the system environment in which they execute. Productivity may be increased substantially if techniques are developed that evolve programming methodology to adapt existing program fragments instead of starting over each time with a clean slate.

- *Data Bases.* Since design environments must store substantial amounts of information about the systems under development, new approaches to data-base management will be required. Promising new ideas in configuration and consistency management will need exploration, while new applications of general queries by tools in the system are being

developed. Improved user and systems interfaces to data bases will be required, in part to integrate algebraic and analytical tools provided by general-purpose programming languages into the data-base environment.

- *Graphics and Human Interface.* The presentation of information about designs and programs represents an important research direction. Systems typically contain an enormous amount of detail; mechanisms for presenting abstract views of designs and programs need to be explored if human programmers are to deal with these complex systems. Research should consider both dynamic (e.g., system animation) and static (e.g., cross-reference table) views.

- *Implementation and Optimization.* Powerful programming languages cannot be successful unless they are implemented efficiently. New techniques for the efficient implementation of very high level languages will be needed to make such languages palatable and bring their expressive power to bear on realistic problems. The impact of the programming environment on implementation strategy will give rise to new research directions.

- *Specification, Verification, and Testing.* Automatic and semi-automatic techniques are needed to help in the production of correct programs and designs. Hence, research in specification of programs should be encouraged and be complemented by research in verification and automatic test generation. Specification language research must consider specifications at the fuzzy incomplete stages to support the design phase and specifications in their most mathematical forms, to support testing and verification. Although verification technology has fallen short of the goal of automatic correctness proofs of realistic programs, it can still be a powerful tool to assist the programmer in reasoning about the program and planning for its development and testing. Test generation strategies need to be improved to assist the programmer in discovering faults in programs written in advanced languages.

- *Debugging and Systems Maintenance.* Once the existence of a fault has been established, techniques are needed to help the program locate its cause. Debugging technology has improved enormously by the advent of "source-level debuggers" in which the compiler symbol table is made available for the debugging process. Further progress is likely if other

information developed by the compiler made available to the debugger. For example, the debugger could make use of static data flow analysis results or of the verification conditions that the verifier could not prove. Graphics and human interface will be an important component of debugging research. Besides correcting bugs, there is a need for system support to make upgrades, alterations, and enhancements to large systems.

- *Systems.* The principal tools of the software engineer, the computer systems he uses for development, will need continual improvement. Workstations will need to become more powerful, possibly by providing specialized support for fundamental tasks in the programming process. Furthermore, new research will be needed into mechanisms for turning the network into an integrated programming system, so that teams of programmers can work together to produce large systems. This will require breakthroughs in distributed systems and communications.

- *Theory.* The entire research undertaking will need a solid underpinning in theoretical concerns. Programming language semantics and semantics of specification languages will play an important role. More and more, programming systems will be generated from abstract specifications. In addition, substantial research into incremental algorithms for managing a system underchange will be needed to support the programming environment. Furthermore, these algorithms will need to be parallel to exploit the emerging class of parallel architectures.

Parallelism

The need for computational capacity continues to outstrip the ability of hardware technology to deliver it. Hence, computer architecture has turned to parallelism as a mechanism for achieving additional performance. Although computer manufacturers have long employed transparent parallelism in their high-performance architectures, we are now seeing architectures in which the parallelism is explicitly exposed to the programmer. Moreover, it is not just the supercomputers and near-supercomputers that are using explicit parallelism; the attractive price-performance ratio of single-chip processors is driving the entire market toward multiprocessor-based designs. An increasing number of vendors will turn to parallelism over the next few years in an attempt to provide either better performance or better performance for the price.

Unfortunately, parallelism does not come without its penalties. Currently, high-performance execution on explicitly parallel architecture is realized by having the programmer organize the computation so that many tasks can be performed concurrently. This amounts to trading programmer time for system performance. Given the already heavy burdens of software development, performance gains will not come easily. Furthermore, parallel computations often increase the communications costs in a computation, consuming a part of the performance gain. Although these costs can be minimized by carefully scheduling where and when each concurrent task is performed, this too consumes precious programmer effort.

The goal of increased computational capacity is so critical that we propose a broad initiative to achieve the promise of parallelism. The goal of this initiative is to make it possible to routinely obtain computational performance of 10 to 100 times what the base technology can deliver. Furthermore, these gains must be achieved without adversely impacting programmer productivity.

To achieve the goals of this initiative, research will be needed in a variety of component technology areas.

- *Architecture.* Research in parallel architecture will need to evaluate a variety of promising structures, ranging in granularity from coarse to fine. Furthermore, as we become more experienced with parallelism, new architectural ideas worthy of exploration will emerge. The critical notion is that a good architecture should be capable of delivering high performance with good programmability. High performance alone is not enough if there exist formidable obstacles to achieving it. Furthermore, architectures must be designed to minimize the impact of higher communication costs that are inherent in parallel computation. As experience is gained with automatic systems for detection of parallelism, new architectures that support the central paradigms of such systems will be needed.

- *Component Design.* Research is needed to improve designer productivity for system components including processors, memory systems, and interconnection networks. Innovation is hampered by the extraordinary effort required to design large, complex VLSI systems. More suitable single-chip processors are required, with special features for managing communication, synchronization, and access to the memory hierarchy. Research on components with multiple processing elements and powerful on-chip communication, such as systolic arrays, will also

be needed.

- *Languages and Language Implementation.* Programmability is the central concern if the promise of parallelism is to be achieved. The programmer must be able to benefit from the advantages of parallel architectures, while maintaining a high degree of productivity through the use of high-level languages and programming systems. Automatic and semi-automatic methods for extracting parallelism from existing languages must be further developed, along with new languages that combine high programmability with naturally parallel structures for which high performance is easy to achieve. In addition, the allocation of parallel computational resources, the analysis of separately translated modules, and improvement in alias resolution represent formidable research problems inhibiting parallelism using any language.

- *Algorithms and Applications.* Research is needed to develop new algorithms and programs that solve problems of fundamental importance and embed the use of parallelism at the deepest levels. Such work ranges from the most theoretical aspects of parallel algorithm design to practical considerations in implementing parallel applications of high importance. Fundamental limits of parallel computation will also need to be explored. A central goal of this research will be to have an impact on parallel computation comparable to the impact that analysis of algorithms has had on computer science generally.

- *Distributed Computing.* Research in distributed computing is concerned with computation using loosely coupled systems of processors, as in local-area or long-distance networks. Often, this loose coupling is a result of the need to coordinate the activities of geographically separated agents. Work in this area has had a different emphasis from research in more tightly coupled parallel computing; it deals with ways of coping with *uncertainty.* In a distributed system, components participating in a computation generally have only limited knowledge of the state of the rest of the system. Uncertainty arises because inputs to the system might arrive from different sources and at unpredictable times, because the timing for different activities may be unpredictable, and because components involved in the computation may fail without warning. Issues of uncertainty have not yet been addressed very much for tightly coupled parallel processing, but they will surely need to be considered. As tightly coupled parallelism research expands to

treat a wider variety of problems, and as techniques from distributed computing become more efficient and better understood, we expect the research in these two areas to converge. Meanwhile, more research is required on architecture, algorithms, languages, and specification and verification techniques for distributed systems.

An important consideration in encouraging research on parallelism in universities is the availability of parallel computation facilities. The NSF and other granting agencies, as a part of this initiative, should take steps to build the research computing infrastructure to include experimental parallel systems. Equipment grant and resource access programs must take steps to ensure that every computer science department has direct or indirect access to a variety of parallel computing systems.

Robotics and Automation

Robotics and automation, a broad area that encompasses research on all aspects of the interface between the electronic and physical worlds, is ripe for significant progress over the next decade. Its importance to society is self-evident: it represents a powerful lever by which human productivity can be amplified many times.

Progress in automation will require progress in many key areas. Some of these are the constituent technologies that support automation. These would include such items as geometrical and physical modeling, computer-vision, and mechanics and control. The rest are the integrating technologies that allow the various elements of automation to be assembled into a total system. These include the problems of consistent representations of physical phenomena, reasoning and analysis tools, and specification mechanisms for physical objects and processes.

The following are some of the challenges facing researchers in the individual constituent technologies that makeup the field of automation. These and related areas will all require a substantial investment of research effort to achieve significant advances.

- *Algorithmic Design.* A vast range of problems are raised by the need to analyze many forms of data and generate adequate response at acceptable speeds. Every subfield, for example, visual sensing, task planning, and modeling, poses major research problems, most of which we have only begun to appreciate. Often, general solutions to these problems will not be computationally feasible, and families of special solutions

covering the most commonly occurring practical cases will have to be found. Combinations of heuristic search techniques with more general algorithms may frequently result in greatly improved performance.

- *Computer Vision.* The most significant need for computer vision is the development of robust basic algorithms that can deal with real environments rather than artificially simple laboratory situations. The development of these algorithms will depend on advances in fundamental issues such as physical light scattering phenomena, signal models, and the representation of objects including topology, geometry, and image projection properties. Of particular importance is the careful decomposition of visual perception into a set of formal, generic problems that can be studied independently of specific applications. It is already clear that competent vision systems will require massive computation at all stages. The identification of general parallel computational models for vision is essential for the development of new architectures to support vision processing. Advances in sensor engineering will also be extremely important. Desirable developments would include significant increases in resolution, dynamic range, as well as specialized eyes for various spatial scales and spectral ranges. The need for new sensor mechanisms to support active object tracking is indicated in many applications.

- *Geometic and Physical Modeling.* The problems of modeling figure heavily into the question of integration, but can also be considered separately. In addition to currently available models for single rigid solids, completely new classes of models are needed. These would cover the interrelationships in assemblies of solids, the modeling of internal forces (e.g., spring compression), and flexible materials (e.g., cloth, wire). Eventually such difficult physical cases as fluids, gels, and inhomogeneous conglomerates (e.g., piles of dustor gravel) will need to be modeled. The state of computer modeling of physical processes is extremely primitive. Accurate, extensible models of physical phenomena, incorporating a knowledge of error, must be developed. These will eventually need to handle the whole of basic physics, including such issues as friction, sliding, rolling, support, impact, oscillation, momentum, and so forth. Required are not only models of the physical process being automated, but also the sensors, tools, and complete environment.

- *Materials Science.* Many new materials will be needed for lightweight, dextrous robots having many degrees of freedom and capable of reliable operation in a variety of environments, some hostile (space, planetary environments, radiation, heat). The need for robots with well understood and modeled flexure as opposed to absolute rigidity should also be studied.

- *Mechanics and Control.* In the mechanics area, the basic need for a strong, lightweight, power-efficient, mobile robot with a number of degrees of freedom, as in animal skeletal systems, has not been met. A variety of designs, including tendon-driven, pneumatic, hydraulic, and electrical have been considered, but none have yet yielded any widely used device other than the standard six degree-of-freedom arm with its crude pincer gripper. Adequate end-of-arm general purpose hands are not yet available, nor are good libraries of interchangeable special purpose hands. A systematic study of common tasks and manipulations needs to be undertaken to provide, for example, a systematic "theory of grasps." In the area of control, preliminary ideas for bottom-level control of force-guided motions have begun to emerge, but only the simplest cases have yet been conceptualized. Techniques for integrating complex sensory information into force-driven control loops need to be developed.

- *Sensor Technologies.* Currently, adequate tactile sensors are neither available commercially or in usable form from other laboratories. Both whole-skin tactile sensing, for safety and advanced control, and tactile sensing of slip and tangential forces would be extremely valuable, but neither is available. Proximity sensing could be a very useful mechanism in control, but interfaces to control systems still need to be developed and standardized. Similarly, interfaces to other specialized sensors, for example, magnetic and capacitative, need to be researched.

- *Integration.* The ability to integrate in one system all the various elements of automation will be critical to the success of any automation project. This integration should begin with the tools used to design the automation systems and should continue through its manufacture, operation, and maintenance. There are three conceptual elements to integrated automation: representation, specification, and reasoning. First, a uniform system of computer *representations* of physical objects, phenomena, and processes is needed. This includes models of

solids, assemblies, tools, sensors, fluid flow, robot tasks, machining operations, and so forth. Second, there must be *specification* mechanisms for users to communicate the objects and actions, as well as the underlying physics, to the system. These will include elements from the areas of language design, computer graphics, CAD, and programming environments. The third conceptual element of integration is *reasoning.* This includes the software tools that, using the representations for physical objects, processes, and phenomena, perform analysis and simulation to predict the behavior of the automated system. It also includes more advanced reasoning techniques to plan system actions to achieve desired goals.

The integrating technologies described will have enormous computational requirements. The massive amounts of sensory data, the need for real-time control, and the computationally challenging nature of the problems will demand the most advanced processing capabilities available. Current work in parallel processing, networking, and supercomputer development will be critical to the operation of automation systems under real-time conditions, as well as to the analysis and planning that precedes it.

Summary and Recommendations

To carry out the goals set forth in this report, we recommend:

1. That a major initiative in software engineering research be initiated, with the goal of a five-fold increase in programmer productivity within a decade.

2. To support the software engineering and the other initiatives, a new program of research support for medium-size, multiple-investigator projects be initiated. Such projects would typically be funded at $200,000 to $800,000 per year.

3. That a major initiative in parallel computation be launched with funding for projects in architecture, software, and algorithms. The goal is to routinely achieve a ten-fold to hundred-fold increase in performance over what the base computing technology can deliver.

4. That the parallelism initiative be supported by a general improvement of computing infrastructure that will make parallel computing systems

available to every computer science and engineering department over the next two years.

5. That a major initiative in robotics and automation research be launched, with the goal of substantial productivity in mechanical tasks within the decade.

6. To support the automation initiative, that steps be taken to ensure that adequate computational facilities are made available to permit realistic experiments with nontrivial tasks.

ISBN 0-8787-236-X