Computer Science Research 1996

The Many Faces of Computer Science

H.J. Sips

1. Introduction

What constitutes the core of computer science? The answer will vary depending on who you ask. It will likely range from a branch of mathematics to an engineering discipline of constructing hardware and software systems. The fact is that techniques and systems from computer science have penetrated very deeply into other disciplines and often stimulated the development of new methods of research. Computers are probably wider used as part of a research method in other sciences than mathematics.

Computer science can be defined as the theoretical, constructive, and experimental science of information processing systems. It is a relatively new science, which has grown from a small core to a very important discipline for society in a time span of only four decades. Central in this development is the digital computer, which by its virtue of almost universal applicability as information processing medium, has placed itself amidst developments in many organizations.

With the enormous increase in the use of computers came the need to put some order in the developments and to create a solid theoretical and methodological basis on which new systems and applications can be developed. In this, computer science relies on the empirical corpus that has grown in four decades in constructing and using information processing systems [1].

In this article, we will place developments in computer science in perspec-

tive and in relation to theory, construction, and experiment as the basic constituents of computer science methodology. Of course, there can be no in-depth treatment or discussion on detailed subjects. Instead, some major developments and trends will be discussed.

2. MATHEMATICS AND COMPUTER SCIENCE

In its kind, computer science is a bit of a strange science: it does not follow the traditional separation between disciplines studying artificial objects, such as mathematics, logic, and theology, and those concerned with observable objects or phenomena, such as physics or biology. In fact, computer science deals with objects from both worlds: it shares its interest in formalisms, symbolic structures and their properties with mathematics. On the other hand, it has much in common with constructive sciences such as electrical engineering when it comes to the design and realization of hardware and software systems.

Its common interest with mathematics in artificial objects is the cause that many computer science faculties have their roots in the mathematics department or be still part of them.

In this respect, the distinction between theoretical computer science and mathematics is often not very clear. This is in contrast with other disciplines such as physics, where we have theoretical physics to explain the nature of physical phenomena (often in highly mathematical terms) and mathematical physics, which is a supporting discipline for theory and experiments in physics.

2.1. Theoretical computer science versus mathematical computer science

Could the same distinction be made in computer science? Would we be able to discriminate between theoretical computer science as revealing the nature of information processing systems and mathematical computer science which is to be supportive to all branches of computer science? Let us try

to make such a distinction as an experiment of thought. Complexity theory is clearly very much related to the nature of computing itself. Hence we would have no problem in classifying this field as to belong to theoretical computer science. But what about for instance Petri nets (see also figure 1) and its theory? Petri nets lend themselves equally well to describing all kinds of dynamic phenomena outside the domain of computer systems and is as such more a general mathematical modeling technique than just revealing the nature of computing. Because Petri nets are frequently applied in computer science, such a subject would then accordingly be classified as belonging to mathematical computer science. On the other hand, the Petri net model could also be considered as a model of computation serving as a semantic model for certain programming systems. From these examples it is clear that to make such a separation is far from trivial.

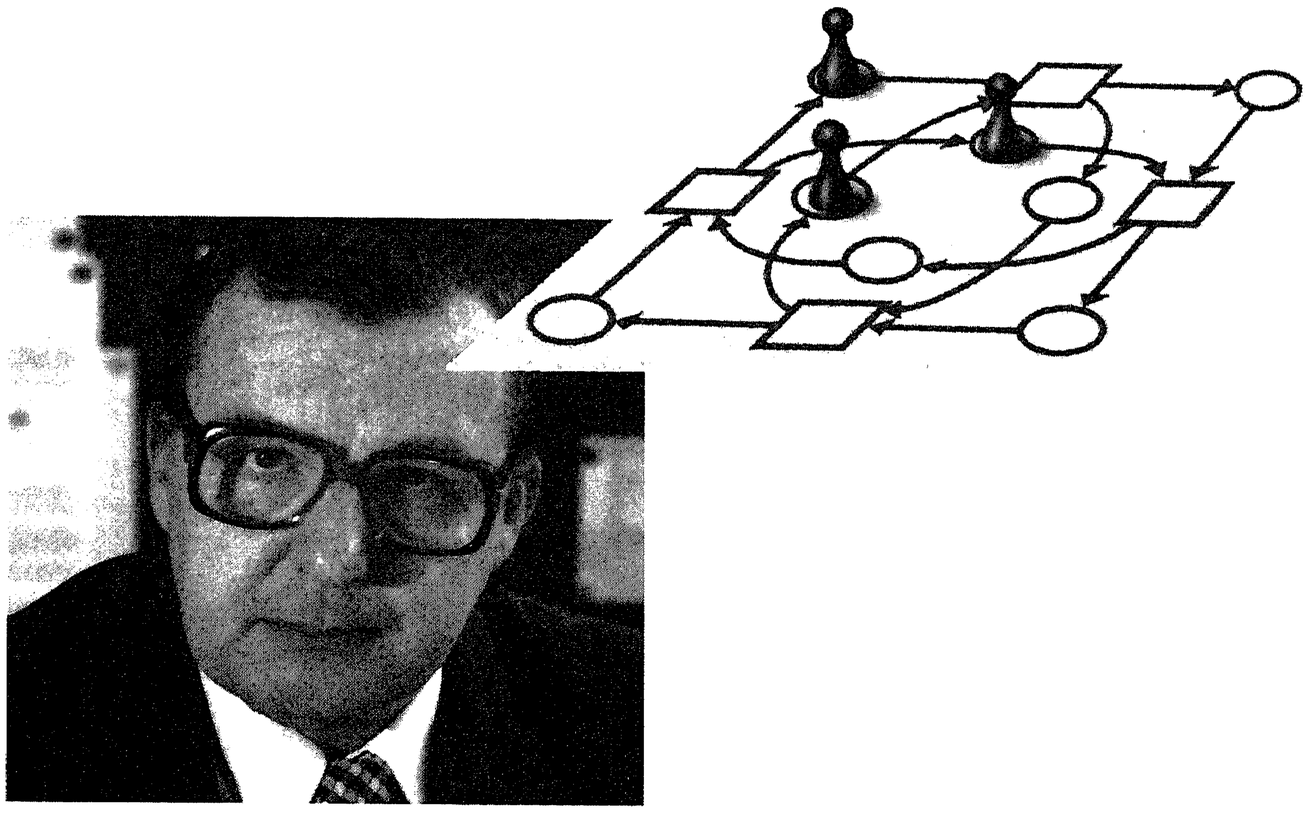

Figure 1. The German computer scientist Carl Adam Petri developed in the 1960s a general method—called Petri nets—for modelling distributed systems and processes (Photo: S. Münch, GMD).

2.2. Model and reality

Why then try to make such a distinction in the first place? The main reason is the problem we have in computer science in the distinction between model and reality [3]. It is often advocated that the application of more formal techniques in computer science will allow us to design systems at a higher level of abstraction, enable proofs of correctness, and lead to robust systems. While in principle this is true, it can only be done effectively if model and reality coincide to a considerable extent. But what is reality in computer science? It is the way we design and build systems. However, we have much freedom in doing that. There is no such thing as a reality 'out there' such as in other sciences against which a model can be validated. This also implies that we can make reality look like the model we have. Some researchers even think that the model is the basis and that reality should shape itself like that. However, current experience is that if we do this, there is a price to pay: some applications cannot be realized efficiently anymore in terms of resource usage and/or time requirements.

Another problem is that in many cases there is not really a precise notion available of the objects we use in reality. We use concepts like 'processes' and talk about 'distributed systems', but generally define them in a rather vague way. For some concepts, like processes, theories do exist, but again these are models and not reality.

2.3. Formalisms

Apart from the progress made, results obtained in the theory of computer science yet have not had a significant impact on computing practice. One of the reasons is the existing gap between model and reality as already explained. A model is necessarily an abstraction of reality. It is in the way abstractions are chosen, where things usually go wrong. Too often abstractions are made on the basis of the resulting mathematical elegance. It suits the mathematician; nicely manipulatable objects result. However, many essential features are abstracted away, leading to formalisms which cannot really be applied in practical cases. Nice examples are theories of communicating processes. The first theories only allowed synchronous communication between processes. This could not hold: asynchronous communication is very essential in many real-life systems and must be part of any theory of processes.

Another reason for the low impact of formal techniques is the highly developed (mathematical) skills that are needed to use them. Most software developers designing actual systems are not acquainted with formal techniques and reasoning. One cannot expect to educate enough people to master these methods and obtain the necessary mathematical skills. The only way would be to bring these techniques down to a form understandable to the average system designer and supported by user-friendly tools.

To overcome the current problems, research in theory should be more directed towards diminishing the gap between model and reality and less towards the (mathematical) art of modelling.

3. CONSTRUCTING COMPUTER SYSTEMS

The constructive part of computer science deals with methods and tools to construct hardware and software systems. The hardware side is concerned with the construction of memories, CPU's, and interconnects. This field is conceptually relatively mature in the sense that we know how to construct computer systems. (See also figure 2.) The progress in terms of capacity and speed is currently merely of a technological nature.

This does not mean that no progress has been made. The production of hardware components has become a highly industrialized process. The enormous investments required to develop a new generation has forced a certain standardization of hardware components. These developments make the construction of computer systems from basic components relatively easy.

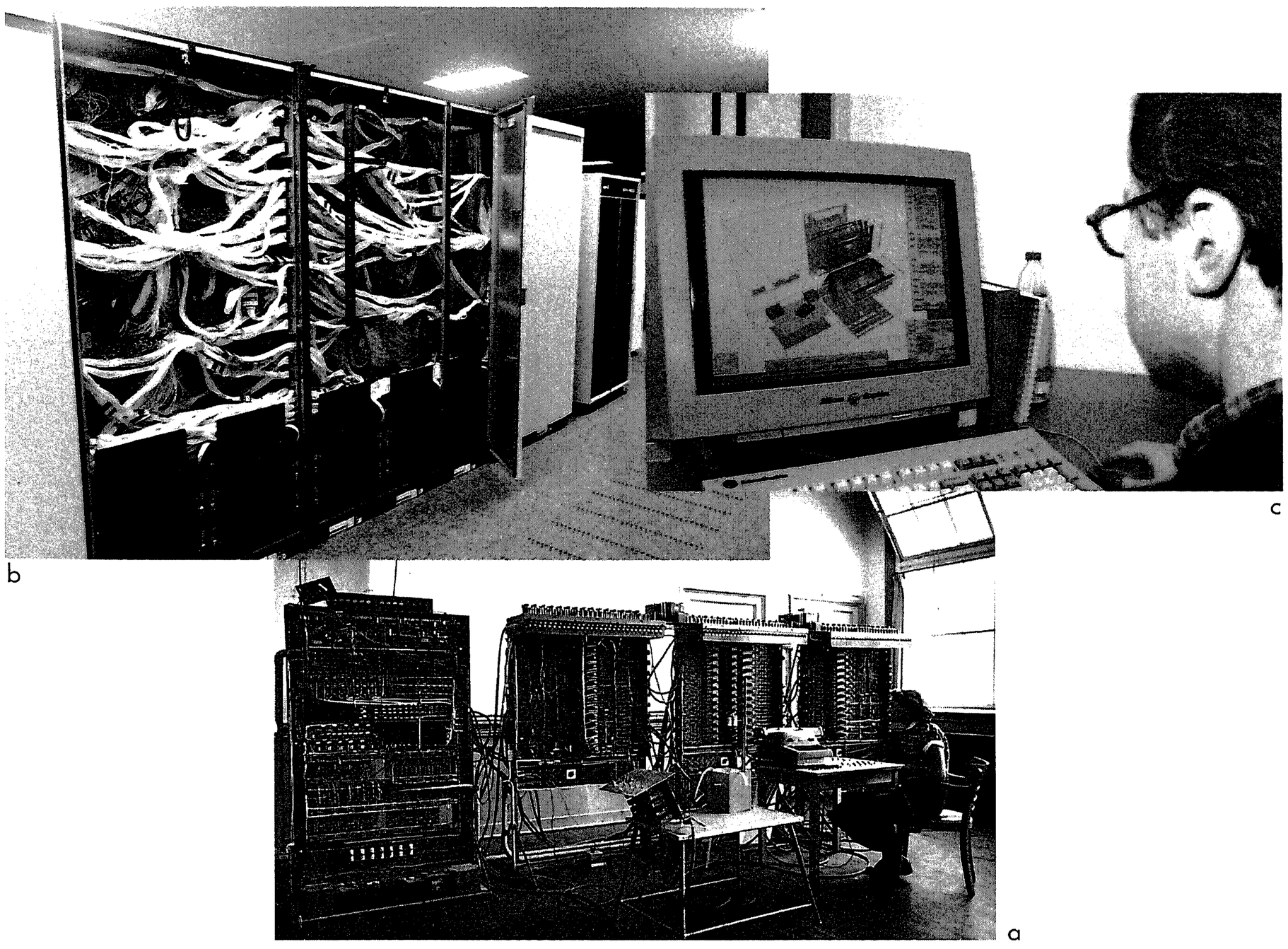

Figure 2. Contrary to software, hardware technology has considerably matured over the years: (a) the ARRA computer developed at CWI (1952), (b) a CDC Cyber 995 mainframe (1980's), and (c) a high performance graphics workstation (1990's).

On the software side, technology is much less mature. The process of software development is still dominated by much detailed hand-crafted work and, even worse, development time is not diminishing at a pace required to deliver in time robust software systems with good performance. Software development time is now becoming the major critical factor in bringing new products to the market.

This problem has been recognized for some years now and is referred to as the software crisis. Basically, two approaches have been proposed to solve the problem. On the one hand, raising the level of abstraction of programming languages and systems would give programmers a more powerful way to express their applications and leave many of the implementation details to smart compilers. The ultimate goal is to be able to automatically

generate code from precise specifications.

Another approach is to reuse code. Much too frequently, programmers implement the same functions and algorithms all over again and do not use programs that already have implemented the required features.

3.1. Programming languages

The two proposed solutions would indeed help to solve the software crisis. However, things have not developed along these lines. There is still a strong base of third generation imperative programming languages, which is not likely to disappear very soon. The original goal of a single powerful programming language for all purposes has not been achieved. On the contrary, powerful programming languages, like ALGOL, have not survived for various reasons. On the other hand, a proliferation of languages has also not occurred. We even see a development towards a smaller set of languages due to the enormous price pressure on software caused by the success of the personal computer. Good quality compilers for programming languages on personal computers can only be provided at low cost when there is a very large user base.

Thus far, higher level languages such as functional languages have not had their expected (by some) breakthrough. Partly this is caused by the lack of commercially available, efficient implementations and partly by the lack of user acceptance of the different model of computation that comes with the use of such languages. Object-oriented features on the other hand seem to find their way into the world of programming languages, not as fully fledged new programming languages, but more as add-on's to existing languages like C, COBOL, and Ada. It is not clear whether this popularity is due to the fact that objects provide an easy mechanism to create abstract data types or that features such as inheritance are favoured. The latter concept is certainly more difficult to handle, since it relies on the modelling capabilities of software designers, and when applied incorrectly, can easily lead to bad software designs.

Will higher level programming languages be accepted in the near future? It must be said that at the moment their future as general programming language is not bright. However, for prototyping purposes or as a language tailored to a specific domain, concepts found in these languages might be very useful. Systems like MatLab or various script languages show that domain specific high level programming systems do satisfy user needs. Also, the popularity of spreadsheets shows that a different programming paradigm can be attractive for specific applications, but the added value must be very clear.

3.2. Software reuse

Complementary to the use of powerful languages, software reuse has the potential to speed up program development. By applying software reuse techniques, certain parts of a program are composed from a number of well-engineered and documented and frequently used code fragments. Although an appealing idea, the problem of software reuse in part turns out to be an organizational problem. One can only apply this technique if reuse software models are accepted on a wide scale and reuse libraries are standardized. For specific fields this has long been current practice (e.g. numerical libraries), but in other domains the sheer effort seems to discourage any real progress.

Reuse on a larger grain size level has more of a chance, meaning reusing larger software components to construct new applications. For example, a spelling checker could be reused in various editors or word processing systems. The investment question is in that case a lot simpler: either use an existing piece of software or completely do the coding yourself. The remaining question is the interface problem and a possibly not completely matching functionality.

Related is the recent interest in so-called coordination languages [4]. Coordination languages in effect form a binding component between several pieces of (existing) software. The coordination language (or system) takes care of the proper interaction between the various software components. It is advocated that applications consisting of software objects written in different programming languages can be realized faster and more flexible. As an example, consider an application working according to a client/server model. Client/server interaction could be programmed in a coordination language, while the actual code for the client and server processing is written in another language. Also in distributed systems a coordination approach to system design will often be necessary, as local systems will be implemented by using different programming languages.

3.3. Conclusion

From the above arguments, one might conclude that no real progress in software construction has been made in recent years. This is too negative a conclusion. We have seen computers change from large unfriendly mastodons to user-friendly personal computers and workstations. This is not only due to hardware developments. Frequently occurring functions in applications such as user interfaces and databases have developed into powerful reusable products with standardized interfaces. The desk top metaphor, although first critically received by many computer scientists, can be considered a true innovation.

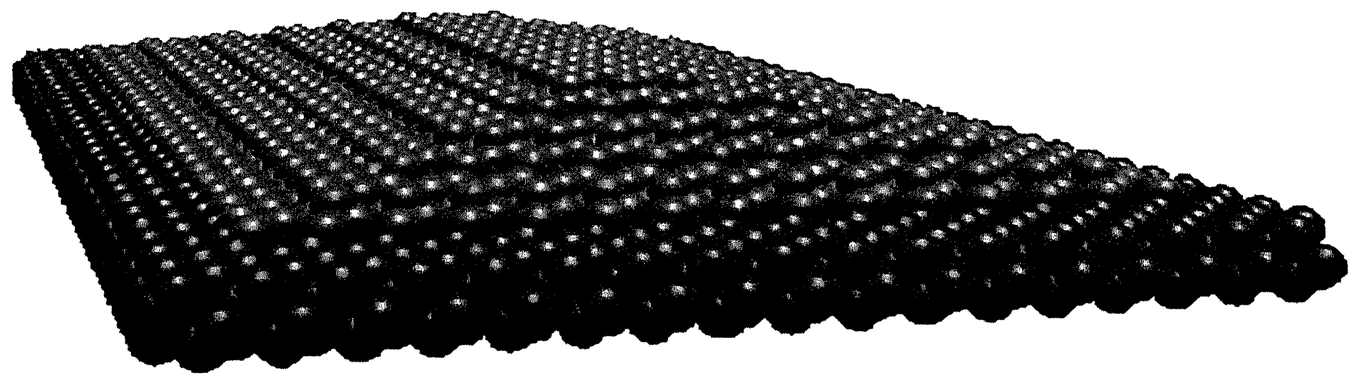

Figure 3. Upper layers of a protracted 'hut cluster', containing about ten thousand atoms created on a Si(100) 1x2 surface by molecular beam epitaxy. Computer simulations of complex systems like the dynamics of such clusters require considerable (parallel) computing power and become more and more an essential part of scientific research. (Courtesy Delft University of Technology, department of Applied Physics/Physics Informatics.)

4. The experimental side

The third view on computer science is experimental. In general, any software system is based on a set of requirements. Some of these requirements are functional, some are non-functional (such as performance). Requirements may be explicit or implicit, quantifiable or not quantifiable. More important, the functionality space is not one-dimensional. Many non-comparable aspects need to be taken into account before the question whether an application serves its purposes can be properly answered.

As a consequence, many software systems are so complex that the only way to validate new concepts is to set up experiments. This is normal practice in any experimental science and may take the larger share of a project's funding. Surprisingly, this is hardly ever done in computer science [2]. There is a lack of experimental evidence in most computer science projects, mainly because there is no money left (or asked for) for validation. Here the binding of computer science to mathematics works out negatively. The main research method in mathematics is analytical and there is no real tradition in performing experiments as part of the research method. As a consequence, most computer science results only consist of claims, without

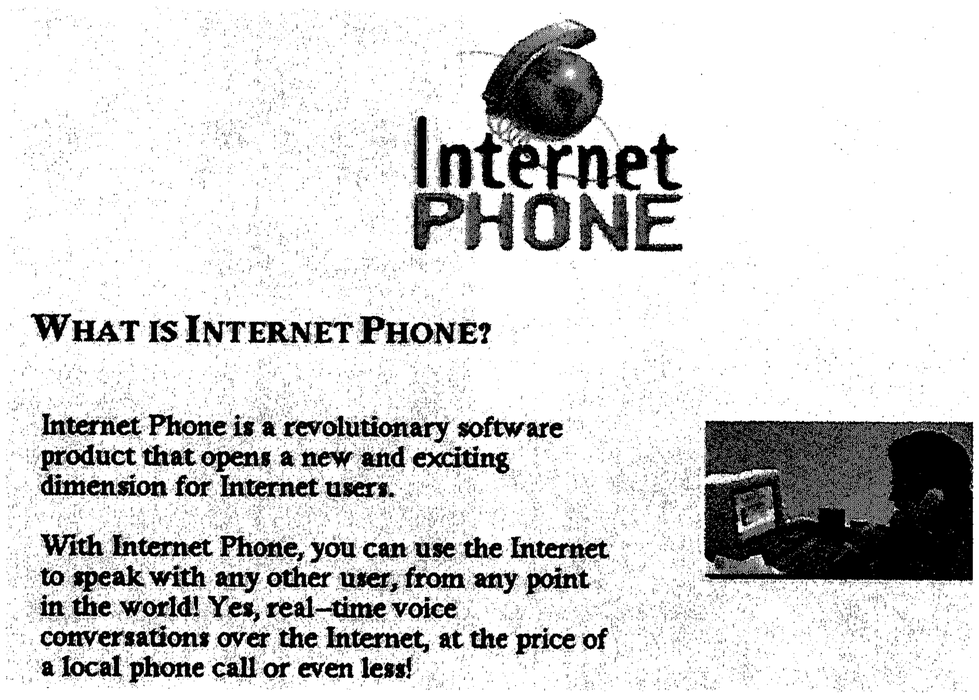

Figure 4. Recent developments such as those around Internet corroborate the ongoing dramatic influence of computing on communication.

ever proving them to come true in an experimental setting.

Another problem is that building software systems is often a tremendous task, which usually does not contribute to academic research records. Even worse, time spent on writing programs cannot be spent on writing papers. With the current emphasis in academia on the quantity of publications, this indeed will remain a problem for some time.

5. IMPACT ON OTHER SCIENCES

Apart from internal developments, computer science also has introduced a new research method in traditional sciences. For instance, in physics (computer) simulation has become an important third research method, complementing theory and laboratory experiment (see figure 3). In general, the field of modelling and simulation has been given a large impulse through the availability of powerful and relatively cheap computers.

Besides having introduced a new research method, computer science has also extended the corpus of other sciences. An example is management science, where information technology is considered a new production factor along with human resources and capital.

6. WHAT NEXT? COMPUTATION AND COMMUNICATION

Meanwhile research in computer science itself is very much driven by the astonishing development of computer hardware. It is not that the basic principles of digital computation have changed so much, in fact nothing really fundamental has changed since the day of Von Neumann's conception of the principle of digital computers, it is the mass production and minia-

turization of basic devices such as memories and processors which is the main driving force in today's computer science research.

The above developments are also bringing together the field of computation and communication (see also figure 4). Research and developments in both areas have long been quite separate, even each with their own jargon and terminology. Surely, digital computation did enter the communication field years ago, but mainly for its internal operation (i.e. in digital branch exchanges). But the merge of communication with computation opens completely new fields of application. For the first time computers will be used to create new economic activities rather than just automating the existing ones.

The impact on research will be large. Many research questions need to be addressed. If we can link computers together without (technical) problems, irrespective where they are placed, questions arise whether we can manage such complex systems. System configurations will become much more dynamic and will have to be maintained while in operation. The question of interoperability of systems and languages will have to be addressed again. The strange thing is that within the sequential computer we have not been able to realize proper solutions for this problem. However, distributed systems can simply not be realized without having solutions for the interoperability problem.

In trying to come up with answers to these questions we are faced with the problem that we really do not know where we are heading with this technology. And we cannot find out without really building and experimenting with systems and applications. In short, all faces of Computer Science are needed, in mutual cooperation, to find the appropriate answers to the challenges imposed upon us.

REFERENCES

1. A. RALSTON, E.D. REILLY (EDS.). (1993). *Encyclopedia of Computer Science*, 3rd ed., IEEE Press, Van Nostrand Rheinhold.
2. R.L. GLASS (1994). The software research crisis. *IEEE Software*, November issue.
3. R. KURKI-SUONIO (1994). Real Time: further misconceptions (or half-thruths), *IEEE Computer*, June issue.
4. N. CARRIERO, D. GELERNTER (1992). Coordination languages and their significance. *Communications of the ACM, 35(2)*.
5. (1994). *European Information Technology Observatory 94*, EITO, Frankfurt.