Lecture Notes in The Philosophy of Computer Science

Matti Tedre

Department of Computer Science and Statistics, University of Joensuu, Finland

This .pdf file was created February 19, 2007.
Available at cs.joensuu.fi/~mmeri/teaching/2007/philcs/

3 Three Traditions in Computer Science

- Is computer science a branch of mathematics?
- Is computer science a branch of engineering?
- Is computer science a science?
- What is the "fundamental question underlying all of computing"?
- What aspects are there to the fundamental question?
- Are the latest trends undermining that fundamental question?
- Why should we care?

In the beginning of this course it was noted that there are multiple interpretations of the term *science.* Also *computer science* can be understood in a number of ways. For instance, it can be understood as *specific classes of activities*, such as modeling, developing, automating, and/or designing (see, e.g., Denning et al., 1989). It can be understood as a *way of thinking* (Arora & Chazelle, 2005), especially one in which an individual is able to switch between abstraction levels and simultaneously consider microscopic and macroscopic concerns (Dijkstra, 1974; Knuth, 1974). It can also be understood as an *umbrella term* for a large variety of topics such as robotics, e-commerce, visualization, and data mining (Zadeh, 1968; Denning, 2003). Computer science can be understood as a *body of knowledge* about topics such as digital computation or information processing.

Computer science can be understood as an *institution* run by cliques of scientists and dedicated to computing research. Or it can be understood as an institution run by power elite and dedicated to the technocratic endeavor. Computer science can also be understood as having a temporal dimension, thus forming a *historical continuum*: "the 45-year path of computer science". It can be understood *broadly* as studies of phenomena surrounding computers or *narrowly*, say, computer science = programming. Computer science can be interpreted as incorporating *subjective* and *objective* issues: the *art* and *science* of processing information. Computer science can be also understood as a *profession*[12]. Many of the above-mentioned interpretations of computer science can have both normative and descriptive meanings, and they can be combined, pooled, and re-organized to produce an even greater variety of interpretations of computer science.

There are parts of computer science that are considerably stable in the sense that they have served as an uncontested basis for research for long periods of time. One of the considerably stable parts of computer science is the *stored-program paradigm*, which was formed during the early 20th century from innovations such as the Church-Turing thesis, and which was epitomized in the *First Draft of a Report on the EDVAC* (Neumann, 1945) and in the construction of the computers *BINAC* and *EDSAC*. Although many constituents of modern computer science—such as the Church-Turing Thesis, the stored-program paradigm, and especially the concept of algorithm—date back a long time (e.g. Knuth, 1972), the field at large has changed radically between the 1950s and 2000s. The

12 See Computing Sciences Accreditation Board's web page for *Computer Science: The Profession*: `http://www.csab.org/comp_sci_profession.html`

diversification of the field has led to ambiguity about the proper topic and proper methods of computer science as well as how the term *computer science* should be operationalized.

THE STORED-PROGRAM PARADIGM

In these lecture notes the term *stored-program paradigm* refers to the constellation of innovations that surround the stored-program computer architecture. Those innovations include a formalization of computable functions (the Church-Turing Thesis); Turing's idea that instructions can be encoded as strings (the Universal Turing Machine); the idea that instructions and data reside in the same memory storage; random-access memory; and the separation between memory, the processing unit(s), control unit, and input/output unit(s) (von Neumann architecture).

In the previous lecture notes the disciplinary history of computer science was discussed. Different authors have argued that the subject matter of computer science is, for instance, *computers* (Hamming, 1969); *computers and phenomena surrounding them*, such as algorithms (Newell et al., 1967); *information representations* and processing, especially with digital computers (Forsythe, 1967; Atchison et al., 1968); *algorithms* and other phenomena surrounding computers (Knuth, 1974); *classes of computations* (Dijkstra, 1972); theory and practice of *programming* computers (Khalil and Levy, 1978); *complexity* (Minsky, 1979; Simon, 1981); and *processes* of information flow and transformation (Denning et al., 1981; Denning et al., 1989). Different points of view of what computer science studies naturally lead to different views of how computer scientists should work.

The activities of computer science have been argued to include, for instance, *representing* and *processing* (Forsythe, 1967), *designing* (Dijkstra, 1972; Denning et al., 1989), *mastering complexity* (Dijkstra, 1974), *formulating* (Dijkstra, 1974), *programming* (Khalil and Levy, 1978), *empirical research* (Wegner, 1976), and *modeling* (Denning et al., 1989). Computer science has been conceptualized as *mathematics*, as *engineering and design*, as an *art*, as a *science*, as a *social science*, and as an *interdisciplinary endeavor* (Goldweber et al., 1997).

The number of research topics in computer science has increased dramatically since the official establishment of the discipline. The 25-item list of computer science topics in 1968 (Zadeh, 1968) consists of mathematical and engineering topics, but the 30-item list of computer science topics ("core technologies") in 2003 (Denning, 2003) consists of a variety of topics that have arisen from the cross-section of computer science and other fields, such as e-commerce (computing and business), workflow (computing and business), human-computer interaction (computing and psychology, cognitive science, and sociology), and computational science (computing and various fields). The perspectives on what kinds of research are considered to belong to computer science have changed over the 35 years between Zadeh's article in 1968 and Denning's article in 2003, and even more over the 60 year-history of the stored-program paradigm. Computer scientists of the 1950s might not have considered intellectual property issues, software project management, graphics and visual computing, data modeling, or algorithm animation[13] as computer science at all. For instance,

13 See Dijkstra's opposition towards algorithm animation in, e.g., Dijkstra, 1989, or in Dijkstra's texts *Written in Anger* and *Why Johnny Can't Understand*.

as late as 1968, computer scientists were hesitant to include even such technical topics as programming under the umbrella of computer science (Atchison et al., 1968).

The topic of this chapter is the relationship between computer science on one hand and mathematics, sciences, and engineering on the other hand. The kind of computer science that is primarily an engineering discipline is certainly very different from the kind of computer science that is primarily a mathematical discipline. It is significantly different to say that computer science is a study of classes of computations (and their applications, such as executable computer programs) than to say that computer science is a study of designing and programming computers (supported by some theoretical foundations). Different emphases of the constituents of computer science yield different views of computer science and entail different ontological and epistemological questions. Different emphases in computer science are also at the basis of many problems and debates that concern funding, departmental organization, researcher and student quotas, available professorships, and education.

After reading this chapter, the readers should know the arguments that support the views of computer science as a *mathematical discipline*, as a kind of *engineering*, and as a *science*, as well as some of their counterarguments. The readers should also understand the fundamental difficulties in computer science research that arise from the interdisciplinary character of computer science.

3.1 Is Computer Science a Mathematical Discipline?

Beginning from the Ancient Greece there has been a tight connection between mathematics and other academic disciplines; it has been argued that above the entrance of Plato's academy there was a sign that read "*Let none ignorant of geometry enter here*". In a similar manner, it has been argued that mathematics is the quintessential knowledge and skill for a computer scientist. Timothy Colburn noted that the philosophical accounts of specific sciences often center around a few *pivotal questions* (Colburn, 2004 in Floridi, 2004). Those pivotal questions concern whether specific sciences are *reducible* to other sciences. One of the pivotal questions of computer science is whether computer science is reducible to mathematics or logic.

REDUCTIONISM

Basically, *reductionism* is a view that complex things can be explained in terms of simpler things. That is, complex things are reducible to simpler things. The term *reductionism* has been used in a wide variety of meanings, and one of the meanings refers to a view that a certain discipline can be fully explained in terms of another discipline. For instance, a reductionist view of chemistry states that chemistry can be reduced to physics, a reductionist view of psychology states that psychology can be reduced to biology, and a reductionist view of mathematics states that mathematics can be reduced to logic. The most obvious reductionist view of computer science states that computer science is reducible to mathematics.

The reductionist view of computer science seems compelling. Many of the champions in computer science are originally mathematicians. The most impressive theoretical advancements in computer science are almost invariably proven and presented in the language of mathematics. Algorithm analysis requires sophisticated mathematical tools. Computer programs can be derived from their algorithmic counterparts and vice versa. For instance, Harry R. Lewis and Christos H. Papadimitriou wrote that much of the modern computer science should be based on the mathematical ideas and models that are "powerful and beautiful, excellent examples of mathematical modeling that is elegant, productive, and of lasting value" (Lewis & Papadimitriou, 1998:1). It is, however, different to say that the *basis* of computer science is mathematical than to say that all of computer science is reducible to mathematics. There are quite some things that duly recognized computer scientists actually do that might not be reducible to mathematics; things such as programming. There again, the appeal of a (mathematical) reductionist view of computer science has led even to a (mathematical) reductionist view of programming, as discussed in the following.

3.1.1 The Mathematical Argument for Programming

C. A. R. Hoare is one of the strong proponents of a reductionist view of computer science. In 1969 he argued that computer science (including programming) is reducible to mathematics. He argued for four principles in computer science, and he argued that those principles are *self-evident*:

> *1. Computers are mathematical machines. Every aspect of their behavior can be defined with mathematical precision, and every detail can be deduced from this definition with mathematical certainty by the laws of pure logic.*
>
> *2. Computer programs are mathematical expressions. They describe with unprecedented precision and in every minutest detail the behavior, intended or unintended, of the computer on which they are executed.*
>
> *3. A programming language is a mathematical theory. It includes concepts, notations, definitions, axioms and theorems, which help a programmer to develop a program which meets its specification, and to prove that it does so.*
>
> *4. Programming is a mathematical activity. Like other branches of applied mathematics and engineering, its successful practice requires determined and meticulous application of traditional methods of mathematical understanding and proof.*
>
> (Hoare, 1969)

Hoare argued that the fact that computers seem to work pretty badly, that they are not as logical and predictable as they should be, is just because modern computers and programs are poorly defined and badly constructed. He complained that because computers and programs are not constructed with mathematical rigor, the only way to work with them is by experiment. However, the grim reality of computer science does not mean that computer scientists should not aspire to rectify the bad situation. Hoare's confidence in the mathematical view of programming is perhaps best visible in his argument that:

> [1969] *Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in a given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning.*
>
> (Hoare, 1969)

That is, Hoare argued that the exact workings of a computer system could be deduced from program code with absolute certainty. Along with Hoare, also authorities such as Edsger Dijkstra (Dijkstra, 1972), Niklaus Wirth (Wirth, 1971), and Peter Naur (Naur, 1966b; Naur, 1969; Naur, 1972) have promoted views according to which computer science is the study of certain kinds of mathematical expressions, namely *algorithms*. Dijkstra wrote—although not strictly in context of programming:

> [1972] *We must not forget that it is not our* [computing scientists'] *business to make programs; it is our business to design classes of computations that will display a desired behavior.*
>
> (Dijkstra, 1972)

In those accounts of computer science where computer science is clearly seen as being reducible to mathematics, programming is either not considered to be a part of computer science at all, or also programming is considered to be mathematical activity. For example, Dijkstra suggested that one should start designing computer programs by *formally* (in mathematical terms) specifying what the program should do. Also in Hoare's account of programming the programmer starts with an *abstract* specification of what the computer should do. Hoare wrote that specifying the program in enough detail is the most difficult part of the programming task, and thus requires the most powerful *mathematical* tools.

Program Specification

Program specification is a description of what the computer program is expected to do. Usually program specifications are informal descriptions of what the program should do, but a *formal specification* describes, often in mathematical terms, the correct functioning of the program in terms of inputs and outputs of the program.

In Dijkstra's and Hoare's accounts of programming, once the specification is at place, the computer program must be mathematically derived from the specification. Derivations are mathematical transformations between two formal systems: *specifications* and *programs*.

The case of the proponents of the mathematics-based view of programming is strong: The logico-mathematical foundations of computer science are clear. The dominant view of what can be computed by machines is a well-defined formal notion. Formal verification is more rigorous approach to program construction than *not* verifying programs formally.

Formal Verification

Formal verification is a view of computer program construction that begins with a well-defined formal program specification, and computer program is constructed so that it corresponds to that specification. Formal verification can, according to the proponents of formal verification, be used to *prove that the behavior of a program corresponds exactly to the formal program specification* (or at least that the program code corresponds to the program specification). If one does not formally verify the correctness of a program, even exhaustive testing cannot show that the program works always correctly (a bug can coincidentally work correctly!).

Because the number of program input combinations is characteristically beyond astronomical proportions, computer programs usually defy exhaustive black-box testing (e.g. Dahl et al., 1972:4; Arden, 1980:139). Consider, for instance, a very simple program (or a piece of hardware) that gets two 32-bit integer numbers as its input, and outputs their sum. The number of possible input pairs for that program is 2^{64}. If executing that program and checking its output would take one nanosecond, testing the program or piece of hardware with all possible inputs would take approximately 585 years. When programs are introduced into real use, there is an interesting phenomenon: Although most computer programs will actually get only a tiniest fraction of the *possible* inputs (practically zero percent of them), programs are still expected to work correctly with any combination of inputs. Even simple programs must cope with arbitrarily large numbers of possible computations, and it can be argued that mathematics offers by far the best tools for dealing with "infinite number of cases" (Arden, 1980:139).

James H. Fetzer argued that the advocates of formal verification embrace the following analogy:

> *In mathematics, proofs begin by identifying the propositions to be proven and proceed by deriving those propositions from premises ("axioms") as conclusions ("theorems") that follow from them by employing exclusively deductive reasoning. In computer science, proofs may begin by identifying the proposition to be proven (in this case, specifications of desired program performance), where deductive reasoning applied to the text of a program might show it satisfies those specifications and thereby prove it is "correct".*
>
> (Fetzer, 2000:253)

That is, the proponents of formal verification believe that one can begin from program code and, by repeatedly applying some derivation rules, to arrive to the formal specifications. If one succeeds in linking the program code with the formal specifications using a well-defined set of derivation rules, one can argue that the program code corresponds to the specifications, i.e., that the program code is "correct".

Especially when computer science is considered to be a study of abstract things such as algorithms, classes of computations, or symbol derivations, computer science can perhaps be considered to be reducible to mathematics and logic. Yet, most proponents of the mathematical view of computer science today are not as strict as, say, Hoare and Dijkstra. Although the mathematical tradition can be argued to be superior to other traditions of computer science in quite many branches of computer science, usually the mathematically-oriented computer scientists are well-aware of the limited applicability of the logico-mathematical tradition. However, it must be remembered that in the 1960s and 1970s many considered the formal verificationists' position towards programming to be tenable.

3.1.2 Limitations of the Mathematical View of Programming

In the 1970s and 1980s the dispute between the proponents of the mathematical view of computer science and the proponents of the engineering- or empirically-flavored computer science boiled down to the debate around *formal verification*. Because that debate is one of the characterizing features of the 1970s and 1980s computer science, it is detailed here to some extent. Three especially powerful objections to formal verification are presented below: the objection that proofs are a product of social processes; the objection that physical world, vis-à-vis mathematical abstractions, is uncertain; and the objection that there is an unsurmountable gap between models and the world.

Objection 1: Proofs in Computer Science And Mathematics Are Different

Formal verification of computer programs is often considered to have begun in the 1960s with seminal articles on the topic by John McCarthy, Robert Floyd, and Peter Naur (McCarthy, 1962; Floyd, 1967; Naur, 1966b). C. A. R. Hoare's article *An Axiomatic Basis for Computer Programming* (Hoare, 1969), however, made the breakthrough of formal verification movement. Although formal verification did not make it extensively into programming *practice*, its intellectual foundations were not seriously challenged until 1979, when Richard A. De Millo et al.'s article *Social Processes and Proofs of Theorems and Programs* (De Millo et al., 1979) was published.

De Millo et al.'s case is based on Imre Lakatos' work in the philosophy of mathematics. In *Proofs and Refutations* (Lakatos, 1976) Lakatos gave a historical account of how Euler's formula[14] was reformulated again and again for almost two hundred years after its first statement, until it finally reached its current, stable form. Lakatos' work is a criticism of dogmatist mathematics and puts forth an idea that mathematical knowledge grows through proposals, speculation, and criticism—by proofs and refutations. According to Lakatos, even the mathematician, the researcher of the "king [or queen] of sciences", does not know anything for certain. De Millo et al. agreed that computer science is like mathematics, and that similar to mathematics, also in computer science the acceptance of proofs ought to be a product of social processes and credibility. That is, proofs do not get accepted due to incontestable logic or due to their indisputable fit with existing theoretical computer science. Many proofs are refuted, many are reformulated and rewritten, and in the course of time some proofs are included in the commonly accepted *hard core* of theoretical computer science.

Then De Millo et al. continued to argue that those proofs that formal methods of program verification produce are very different from the proofs in mathematics. In mathematics, De Millo et al. wrote, proofs generate excitement: "no mathematician grasps a proof, sits back, and sighs happily at the knowledge that he [sic] can now be certain of the truth of his theorem" (De Millo et al., 1979). Instead, De Millo et al. argued that when a mathematician comes up with a proof, he or she runs out of the office, looking for someone to tell about it; shows it to colleagues; sends it to graduate students and tells them to check it; phones colleagues around the world; and so forth. If all the colleagues and graduate students also find the proof to be interesting and believable, the mathematician sends the proof to publication. If the referees and editors also find it attractive and convincing, the proof gets published, and it gets read and evaluated by international audience. At any stage of this process, flaws in the proof may be found, some parts may get rewritten, connections to other areas of mathematics can be made, generalizations can be derived, and so forth (cf. De Millo et al., 1979). In Lakatos' account of how mathematics works, it is important that a large number of mathematicians critically and thoroughly review proofs that are proposed.

De Millo et al. argued that in program verification, proofs of program correctness do not create or undergo similar social processes because the proofs of program correctness have nothing exciting in

14 Euler's formula for planar graphs without edge intersections is now known as V-E+F=2, where V is the number of vertices, E is the number of edges, and F is the numbers of faces in the graph.

them. They are long and complex and peculiar, they are just too cumbersome and boring to read. Proofs of program correctness fill up large volumes of books:

> *The verification of even a puny program can run into dozens of pages, and there's not a light moment or a spark of wit on any of those pages. Nobody is going to run into a friend's office with a program verification. Nobody is going to sketch a verification out on a paper napkin. Nobody is going to buttonhole a colleague into listening to a verification. Nobody is ever going to read it. One can feel one's eyes glaze over at the very thought.*
>
> (De Millo et al., 1979)

In a sense, it seems that the proponents of axiomatic basis of programming and formal program verification took a task comparable to the logicists' project that was culminated with Whitehead and Russell's *Principia Mathematica*, which was an attempt to use the inference rules of symbolic logic to derive *all mathematical truths* from a well-defined set of axioms.

De Millo et al. are probably correct in that *in practice* nobody is going to read and check hundreds of pages of long but trivial chains of substitutions. However, noting that checking proofs of program correctness is tedious, boring, and unrewarding is not a sustainable refutation of formal verification if formal verification were a sustainable position *in principle*[15]. That is, just that something is burdensome does not necessarily mean that it should be ignored. But formal verification is not a sustainable position even in principle, as the next two sections show.

Objection 2: The Physical World Is Uncertain

Although De Millo et al.'s text raised a lot of discussion, a stronger argument against formal verification came a decade later, in 1988, from philosopher (not a computer scientist!) James H. Fetzer (Fetzer, 1988). Fetzer was sympathetic towards De Millo et al.'s work, but he went on to argue that the real problem of formal verification is the fundamental difference between the abstract world of mathematics and the physical world of actual computers. Consider, for instance, the following text passage by Hoare:

> [1969] *When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of electronics.*
>
> (Hoare, 1969)

Hoare's text blurs the notion that although one can mathematically reason about programs *as abstract representations of algorithms*, one cannot mathematically reason about programs as they reside in the computer memory, as swarms of electrons in circuits. That notion is the weak point of program verification that Fetzer attacked. For this discussion, the double-faced nature of computer programs must be made clear.

15 If one ignored De Millo et al.'s appeal to consequences (formal verification causes a lot of work), their notion that formal proofs are not carved in stone is still a noteworthy remark that undermines the eternal character of *any* formal proofs. This aspect of De Millo et al.'s argument goes often unnoticed.

In his book *On the Origin of Objects*, philosopher of computing Brian Cantwell Smith gave an illustrative example about two typical ways of viewing computer programs (Smith, 1998:29-32). He introduced his readers to two programmers named "McMath" and "McPhysics" who disagree on what computer programs really are. McPhysics claims that computer programs are *physical things*: computer programs are swarms of electrons in the circuits of a computer. They are physical phenomena and they can make physical phenomena happen. Computer programs are a part of the causal world (world where particles move in and interact with fields of force), and computer programs can affect the causal world. For instance, computer programs can make monitors blink and printers rattle; computer programs can land airplanes and guide missiles to their targets.

According to McPhysics, the quintessential feature of computer programs is that they do material work—for instance, print characters on screen. Executing a computer program makes things (electrons at least) move. McPhysics admits that computer programs can be presented as algorithms, which are purely abstract things that cannot do any material work, but McPhysics goes on to argue that algorithms are *models* of programs. McPhysics takes the position that an algorithm "could no more *be* a program than a mathematical model of a rainstorm could itself be a storm, could itself cause things to get wet" (Smith, 1998:30). (Smith borrowed the rainstorm example from philosopher John Searle, who wrote that "no one supposes that computer simulations of a five-alarm fire will burn the neighborhood down or that a simulation of a rainstorm will leave us all drenched" (Searle, 1980:423).)

McMath contradicts McPhysics, and claims that programs are *abstract things* in the same way that mathematical objects are abstract. Computer scientists can construct procedures and programs in the same way mathematicians construct functions, theorems, and proofs—in their minds or with a pen and paper. Computers are not necessary: computer programs need not have any physical counterparts (executable programs in computer memory or hard drive or such). McMath admits that abstract programs are not useful in any practical sense, but McMath goes on to argue that that is irrelevant: "that is a practical consideration [...], having nothing to do with the abstract theory of programs itself" (Smith, 1998:30). McMath's argument is embraced by many: For instance, the currently dominant theory of computability treats programs as McMath sees them; as purely abstract things.

So, there are two ways of looking at programs. Fetzer called abstract programs (that are not in an executable form) *programs-as-texts* and executable programs (that reside in, e.g., computer memory) *programs-as-causes* (Fetzer, 2000:267). Programs-as-causes can always be presented as programs-as-texts, and programs-as-causes can often be derived from programs-as-texts. Smith argued that neither McMath or McPhysics alone is right: programs can be considered to be both abstract and concrete objects, depending on one's viewpoint. Herbert A. Simon (1916-2001) also noted that computers are both abstract objects and "empirical objects" (Simon, 1981:22-26). But problems arise if one (1) makes arguments that assume programs-as-causes or assume programs-as-texts, but (2) lets the meaning of the term loosely slide between programs as McMath means them (programs-as-texts) and programs as McPhysics means them (programs-as-causes).

Mathematicians need not care about the uncertainties of the physical world, but those uncertainties cannot be completely abstracted away from real computers. Simply put, contrary to what Hoare wrote, one cannot establish the running of a computer *with mathematical certainty*. A program run on a real computer can be disrupted by Justus stumbling on the computer's power cord, by a cosmic ray interfering with the circuitry, or by too few electrons being at the right place at the right time. Computers, the machines, are physical objects and although one could prove computer blueprints to be theoretically correct, the physical world does not work with mathematical certainty (see Uncertainty Principle). One cannot prove how single electrons in computer circuits will behave (although one can make very good predictions of how very large clouds of electrons behave). Fetzer wrote:

> [1988] *The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.*
>
> (Fetzer, 1988)

Fetzer's article, although philosophically and technically sound, raised fervent opposition. Ten well-known computer scientists lashed out in the March 1989 issue of CACM, writing that the publication of Fetzer's "*ill-informed, irresponsible, and dangerous*" article showed that CACM's editorial process had failed (Ardis et al., 1989). Fetzer was no more amiable in his acrid response to Ardis et al.'s letter; Fetzer wrote that the pathetic quality of thought embodied in Ardis et al.'s letter was a manifest of their failure to comprehend, among other things, either the basis or the consequences of Fetzer's argument. Fetzer wrote, "*In its inexcusable intolerance and insufferable self-righteousness,* [Ardis et al.'s] *letter exemplifies the attitudes and behavior ordinarily expected from religious zealots and ideological fanatics, whose degrees of conviction invariably exceed the strength of their evidence*" (Fetzer, 1989).

The debate between the proponents and opponents of Fetzer's article went on for quite some time. The technical correspondence in CACM 32(3) and CACM 32(4) dealt entirely with Fetzer's argument, there was an invited article to sort out the differences (Dobson & Randell, 1989), and there were a lot of letters to the CACM forum (see CACM 32(6), CACM 32(7), CACM 32(8), CACM 32(9)). I recommend reading Fetzer's entertaining, personal coverage of the things that led to his argument and of the course of events that followed (Fetzer, 2000 in Bynum & Moor, 2000).

In the end, the formal verification debate revealed that many proponents of formal verification failed to recognize the fundamental difference between programs as abstract things and programs as executable implementations on a computer; and the fundamental difference between pure mathematics and applied mathematics. Whenever computers as physical machinery are in the picture pure mathematics turns out to be inadequate, and also some other intellectual frameworks must be utilized.

But the failure to formally verify causal systems (or the workings of actual computers) may not be a very serious weakness of a mathematical approach to computer science. Those proponents of formal methods who recognize the limited applicability of formal verification still have a strong argument for a mathematical view of computer science. Much of computer science is not concerned with executing programs but with studying properties of computations.

Objection 3: Fundamental Gap Between Programs And the World

Another limitation of a purely mathematical approach to computer science, also implicit in Hoare's 1969 article, is the fundamental gap between program, model, and the world:

> [1969] *The most important property of a program is whether it accomplishes the intentions of its user. If these intentions can be described rigorously by making assertions about the values of variables at the end (or at intermediate points) of the execution of the program, then the techniques described in this paper may be used to prove the correctness of the program.*
>
> (Hoare, 1969)

Hoare's statement raises a number of questions: "What does it mean that 'a program accomplishes the intentions of its users'?", "Is it really reasonable to assume that user intentions can be described with mathematical rigor?", and "What kinds of intentions *can* one express in formal systems?".

The types and uses of computer programs today are diverse. When the intention of the user is to computationally implement functions or other mathematical objects, it may be the case that one can straightforwardly tell whether the program corresponds to the user's intention. For instance, consider that the user wants a program that implements a mathematical function. The program has input value(s) and an output value. Now, if the program output for all possible inputs corresponds to the result of the mathematical function with the same inputs, then the program corresponds to the user's intention (insofar as the user's intention is to have a program that computes correctly). However, this kind of a criterion of program correctness works only with programs that implement mathematical or other abstract, formally definable objects.

But, clearly, not all programs implement functions or other abstract, formally definable objects. In fact, most of the programs today do something else. Many programs have some kind of a relationship with the physical world. For instance, some programs model phenomena, some programs measure physical phenomena, some programs are tools for work, some are for entertainment—and some programs are parts of larger causal systems that affect the physical world, like those programs that control nuclear plants and land airplanes. It is much harder to tell whether "a program accomplishes the intentions of its users" if one considers programs that somehow relate to the physical world than if one considers programs whose purpose is to implement formally defined functions.

That is, it is difficult to describe, in mathematical rigor, the correct functioning of programs that land airplanes, steer missiles, control factories, or monitor medical equipment. And especially with programs like word processors, operating systems, spreadsheets, or computer games, "correct functioning", in the sense of doing what the user intends the program to do, is impossible to establish with mathematical rigor. In a word, just that a program is *proven correct* does not mean that the program does what the user *intends* the program to do. The issues of programs, models, and the world are discussed in more detail later. As of now it is enough to note that, except for very simple programs, describing program specifications or requirements with mathematical rigor seems, at our current state of art, impossible.

Unless one wishes to limit computer science only to implementations of formally definable functions, there is more to computer science than mathematics. That is not to say that the basis of computer science could not be mathematical. It quite certainly is. The fundamental principles and theories of computer science originate from mathematics and are expressed in the language of mathematics. Today many proponents of the mathematical view of computer science concede that actual working computers are indeed physical objects, but they still hold that physical abstractions of computers (such as their molecular structure) are of little use for computer scientists, and that the really useful abstractions of computers are mathematical (e.g. Lewis & Papadimitriou, 1998:2). Nonetheless, nowadays the proponents of a mathematical *basis* for computer science usually recognize also the utility of empirical and engineering methods.

Although the formal verification movement has mostly disappeared, *formal methods* are anything but dead. For instance, in the April 1995 issue of IEEE Computer Jonathan P. Bowen and Michael G. Hinchey argued that proper application of formal methods results in increased correctness of hardware and software as well as in improved structuring and maintainability, but not in increased costs or delays in schedule (Bowen & Hinchey, 1995). Bowen and Hinchey argued for a number of principles of formal methods, such as appropriate notation, sensible level of formalization, cost-estimation, sufficient documentation, high quality standards, and significant reuse of specifications and code. Yet, Bowen and Hinchey also noted that other development methods are useful too, that formal methods cannot guarantee correctness, and that formal methods do not obviate testing.

Ten years after the publication of their article, Bowen and Hinchey revisited their original article and argued that all their principles of formal methods are still valid (Bowen & Hinchey, 2005). However, they noted that to their disappointment, formal methods have not gained the popularity they had hoped. They argued that apparently, “the software engineering community is not willing to abandon formal methods [...] but neither is it willing to embrace them” (Bowen & Hinchey, 2005). Bowen and Hinchey listed a number of formal methods and techniques that have been successful during the past decade, and predict some significant progress to happen during the next ten years, especially in the integration of formal methods and traditional development practices.