Advance Access publication on 12 September 2011
doi:10.1093/comjnl/bxr090

# Science, Mathematics, Computer Science, Software Engineering[†]

DICK HAMLET*

*Department of Computer Science, Portland State University, Portland, OR 97207, USA*
**Corresponding author: hamlet@cs.pdx.edu*

**This paper examines three ideas: First, the traditional relationship between a science, the mathematics it uses and the engineering based on it. Second, the nature of (software) computer science, which may not be a science at all, and its unusual use of mathematics. And finally, the nature of software engineering, its relationship with computer science, and its use of mathematics called 'formal methods'. These three ideas turn on the first of them, since the scientific world view seems natural for the study of computing. The paper's thesis is that while software touches science in many ways, it does not itself have a significant scientific component. For understanding programming, for teaching it and for applying it in the world, science is the wrong model. Mathematics has found its own foundations apart from science, and computer science must do the same.**

*Keywords: science; engineering; philosophy; computing*

*Received 28 April 2011; revised 11 August 2011*
Handling editor: Mark Josephs

## 1. TRADITIONAL SCIENCE AND ENGINEERING (AND MATHEMATICS)

In the well-ordered scientific world that began with Galileo and Newton, the disciplines of science, mathematics and engineering subdivide the intellectual space and support each other. For example, physics describes and explains (say) electrical phenomena. Mathematics plays a crucial helping role; Maxwell's equations precisely capture electrical phenomena.[1] Engineering is never very far from the minds of some scientists. The understanding of phenomena naturally suggests ways in which they can be controlled and exploited by artificial devices. Those devices are not only useful, but they serve to graphically test a physical theory—reason enough for experimental physicists to build them. For an engineer, scientific mathematics is an indispensable tool. The devices that an engineer designs must conform to physical reality or they will not work, and mathematics fills out the design so that reality is served. In some cases, special mathematics makes the engineer's job easier. The best example comes from electrical engineering, where the Fourier and Laplace transforms are used to convert differential equations into algebra. Where a physicist understands an electronic circuit as a set of simultaneous differential equations, an electrical engineer can describe the circuit without calculus.[2]

### 1.1. Science

When it comes to the study of how and why human beings know what they know, science has by far the best story. Other belief systems may have more adherents, but no one explains better than the scientist. Part of the reason is that the scientific world view can use its rationalist methods on itself. Science is scientific. Not the least important part of the philosophy of science is the definition of precisely what is and what is not science. Other belief systems, notably religious and economic ones, are not interested in what they do *not* cover; indeed, they usually claim universal application.

[†]Some ideas in this paper were presented in an invited talk at the First Irish Conference on Formal Methods in Computer Science, University College Cork, July, 2000.

[1]Scientists sometimes find it necessary to invent mathematics as needed, setting mathematicians nice technical problems in justifying the things being done, for example, in approximating solutions of partial differential equations.

[2]Oliver Heaviside [1] was an early proponent of 'algebrizing' differential equations. Heaviside was a complex genius, mathematician, physicist, engineer, at once standing on the side of theory (mathematics) against practice, then again at odds with mathematicians over his operator calculus that was never rigorous (and sometimes plain wrong), yet allowed him to solve problems that defeated all others. He was a pioneer in using vector notation; he cast Maxwell's equations in the concise form they are known today. The tension between science/engineering and rigorous mathematics is a fascinating dark side of the way these disciplines often interact successfully.

The scientific definition of science has its roots in physics based on only 'operationally defined' concepts [2]. Something that cannot be quantitatively measured (at least in principle) is not a proper subject for science.[3] Popper [3] is an articulate proponent of a philosophy of science built on this observable foundation. Science, a scientific theory, says Popper, must be *falsifiable*. Unless there is a way to test a theory, and a possible outcome of the test that shows the theory to be wrong, it is not science. For example, a theory postulating a new sub-atomic particle must describe the circumstances under which it should be seen, and if in those circumstances it is not seen, the theory must be abandoned. But it was science while it lasted—maybe the accelerator can be used for something else.... There are many amazing examples in which a theory predicted an observation that was then confirmed, notably Einstein's general-relativity prediction that light should bend slightly in a gravitational field. But it would be more to the point to remember cases in which a theory failed. For example, J. J. Thomson's idea of atomic structure was electrons embedded like raisins in a 'pudding' of positive charge. When material is bombarded with positively charged particles, the latter should pass through like a knife thrust into the 'pudding'. But it is not so; some alpha particles are greatly deflected by a gold foil, even being sent back toward their source. End of the Thomson atom.

The very best that a scientific theory can aspire to is a status of 'not disproved'. No matter how many experiments it successfully weathers, it remains falsifiable. Yet long-standing theories are called scientific *laws*. Some of these laws are so intertwined with so many aspects of the world that an attempt will be made to preserve them at any cost—even by adjusting definitions and descriptions.[4] An example is the law of conservation of energy, which runs through physics like a golden thread. Other 'laws' retain that name even though they have failed in experiments and been supplanted by newer theories. Newton's second law of motion relating vector force $\mathbf{F}$ to momentum $\mathbf{p}$ and time $t$,

$$\mathbf{F} = \frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t},$$

probably keeps the name despite the predations of quantum mechanics because it is so useful at the macro level and because its triumphs of prediction were and are so large. For example, it correctly describes the mechanics of the solar system with incredible accuracy.

Popper's is not the only definition of science.[5] His motivation may be discerned in the famous quotation, 'I may be wrong and you may be right, and by an effort, we may get nearer to the truth'. Without the requirement for falsifiability, neither party to a dispute ever needs to abandon a position. Popper distinguished science from other world views like religion or economic/political theory in which discussion usually leads nowhere. But others define scientific theory by its positive characteristics such as generality, precision, formality, etc. By many definitions, mathematics is a science, but by Popper's it is not.

## 1.2. Pure and applied mathematics

Mathematics, the 'queen of the sciences', is not Popperian science since it employs no observations from the physical world and therefore cannot be falsified by observation. A mathematical theory is nothing like a scientific theory, but mathematicians are also clear in their logical foundations: A mathematical theory is a body of definitions, axioms and theorems. There is a notion (well, several notions) of mathematical 'truth' defined by the reasoning that is allowed for proving theorems from the axioms. A great deal of work has gone into trying to ensure that theories are consistent, because an inconsistent theory is useless: if two contradictory theorems can be proved, *anything* can be proved in it.

Probably at some time prior to Euclid, mathematics was a physical science. For example, geometry might have expressed laws of lines drawn in the sand, checked by measurements in the sand. But proof-of-theorem mathematics is far more productive because its theorems can be startlingly revealing, expressing and proving what sand-scribblers cannot.

G. H. Hardy, in his famous 1940 essay 'A Mathematician's Apology', [4] proudly averred: 'I have never done anything "useful". No discovery of mine has made, or is likely to make, directly or indirectly, for good or ill, the least difference to the amenity of the world'. He was speaking of 'pure' mathematics, but of course he was wrong—others have shamelessly exploited his and other pure results. Mining mathematics for worldly purposes is called 'applied mathematics', and it works like this: Starting with a mathematical theory (in the literal sense of a body of definitions, axioms and theorems), the applied mathematician maps real-world objects onto objects of the theory, then checks that the axioms hold (in the real-world sense involving observation and measurement), and hence concludes that the theorems hold (again in the world). The virtue in this exercise is that for a deep theory the theorems are anything but obvious, so it is hoped that their worldly analogs will supply new insights into reality. The theory objects are precise and simple; reality is fuzzy and complex. Hence there is a real advantage in understanding the latter in terms of the former.

Applied mathematics muddies the clear waters of discipline definition. It is not mathematics *per se*, but neither is it science. Science is involved, but it belongs to the discipline making the application. Perhaps it is best to consider applied mathematics as simply one method used by scientists. 'Science' does not 'rub

[3]Particle physics includes many ideas that cannot be directly observed, the stuff of mathematics. But the predictions of a legitimate theory must not be so obscured, even if it takes a new multi-billion Euro accelerator to find them.

[4]For some reason, such adjustments are called 'saving the phenomena' rather than 'saving the theory'.

[5]This is not the place to discuss scientific induction, which is at the heart of disputes over Popper's views. Here the concern is not with how scientists do their work (they sometimes use induction, despite Popper's beliefs), but with distinguishing their disciplines from those of a distinctly different character.

off' on the method. The science whose (scientific) theories arise from applying mathematics to physics is physics.

## 1.3. Engineering

Engineering is the making of real things in the world, although the dictionary also allows an engineer to 'arrange or manage' their making. Engineering is also a profession, which means that its things must be made according to certain standards. The dictionary is at a loss to succinctly describe these standards, using words like 'artfully contrive'. Neither do the professions define themselves very well, circularly requiring engineers to graduate from an engineering college and have work experience as an engineer.

An engineer may be something of a scientist or something of a tinkerer. Those roles define the end points of a spectrum whose middle ground is traditional engineering. As almost-scientist, the engineer studies real-world phenomena (if perhaps for an artificial device), tries to understand and predict how the world (the device) will behave and how to exploit that behavior. As almost-tinkerer, the engineer 'fools around' with a device, changing this and that without any clear guiding principle, trying to make it more functional, more reliable, cheaper to manufacture, etc.

The day-to-day work of an engineer is *design* of useful artifacts. There are very few 'philosophers of engineering', and the nature of design seems self-evident to those who do it every day. But it is helpful to define 'normal' design. Its opposite, 'radical' design, is the essentially creative process of inventing something new from scratch. In radical design there are few rules, and the engineer uses science, or tinkers, in whatever combination seems to work. Radical design is often unsuccessful, and is characteristic of the beginnings of an engineering discipline in which bridges collapse and circuits explode.

As a discipline matures, 'normal' design emerges. A body of practical knowledge accumulates over time, and in this process failures play a more important role than successes [5]. Experienced engineers learn what does and does not work and how to capture this knowledge in design rules so that other engineers can use it. There need not be much understanding of *why* the rules work, but they do. An important component of normal design is the 'safety factor', a measure of over-cautious design that compensates for mistakes. Safety factors sound like superstitious magic, and so they are, but there is real engineering knowledge involved. Overdesign is costly, and too much can be counterproductive (e.g. a bridge made too strong may fall under the additional weight). The engineer using normal design with proper safety factors has very little doubt that the device being designed is going to work.

There are two essential elements of the normal design process. (1) The design rules must work, and (2) they must work in the hands of any trained engineer. The engineer learns exactly how to design particular devices, to solve problems that have been solved before, in ways that have succeeded. (And when a normal design does occasionally fail, an intensive root-cause analysis corrects the design rules so that the failure should not be repeated.) Part (1) is obviously necessary, and normal design emerges only when enough is understood to repeatably build standard devices. But part (2) is equally important in normal design. Professionals must be able to carry out the design process. Only then can one hire an engineer, examine that person's credentials, and be assured that one's problem will be solved. Indeed, it is an important service of a professional engineer to warn a potential customer when normal design does not apply, to say in effect, 'You need a magician, not an engineer'.

Normal design eventually comes to be based on theory, on mathematics, which makes it easy to teach and apply. In today's world, the theory is scientific, or at least pays lip service to science. But examples from the past show that scientific accuracy is less important than ability to use the theory. Classical Greek architecture had its normal design, but the underlying theory was a mystical religion. Each part of a classical temple has dimensions that stand in whole-integer ratios to the other dimensions. The numbers are 'harmonious', and hence fitting for a temple.[6]

Addis [6] has studied temple building as well as the structural-steel-construction design rules of his own time. He makes a convincing case that the normal design of steel-frame buildings prior to 1950 was based on erroneous theory. Engineers made their calculations assuming that the steel was never stressed beyond its elastic limit, an assumption now known to be false. Yet the buildings stand, just as the temples do, witnesses to the safety factors included in the design rules. The point is that most of these buildings would not and could not have been built without the precise calculations of the (wrong) normal design theory. In Addis's example, the proof that accurate science is not needed is underlined by engineers' lack of interest in checking the results. No one mounted strain gauges on the beams to verify that the deformations were as calculated. It was enough for the building to stand. Safety factors today are thought to cover variations and flaws in materials; Addis says that their more important role is to cover flaws in the design rules.

One of the methods engineers use to improve design rules is experiment by *parameter variation* [7]. When there is reason to believe that a device is characterized by a handful of parameters and there is an accepted way to measure the performance of the device, it is possible to undertake a systematic search for optimum combinations.[7] An optimal design is not predicted, but directly observed, so the underlying theory need not be scientific or correct.

[6]Experiments, failures and safety factors surely played a role in selecting the ratios.

[7]The US Navy published a handbook for design of (vacuum tube) electronic circuits [8]. Its designs were obtained with a breadboard setup in which trial circuits were built and component values systematically varied. These circuits worked as no others did.

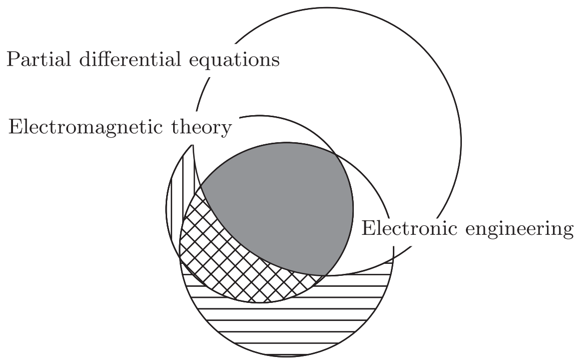

**FIGURE 1.** Traditional inclusion relationships among electrical disciplines.

## 1.4. Traditional paradigm

In brief, then, science seeks to find laws of the world, with mathematics as the preferred way to express them. Engineering must conform to its underlying science, and uses the science's mathematics to establish this conformation for each particular design. In a mature field of engineering, normal design is repeatable and can be performed by anyone with professional training. This paradigm has been spectacularly successful in mechanical, civil, and electrical engineering. Figure 1 shows a Venn diagram of a traditional relationship among mathematics, science and engineering.[8] In the central intersection (shaded), electronic devices are described by the differential equations of electromagnetic theory, from which design rules are derived. In the cross-hatched region the rules are not mathematical. The horizontally striped portion is electronic engineering (often empirical parameter variation) done by tinkerers who use neither physics nor mathematics—perhaps experimenting with over-clocking a commercial CPU falls there; the vertically striped portion is non-mathematical physics not yet used by engineers—perhaps a point there might arise when an experimental physicist stumbles on some new phenomenon. Engineering design is greedy. The cross-hatched region tends to absorb points in the vertically and horizontally striped regions and in turn to be absorbed by the shaded region.

It is natural to apply the traditional paradigm to any new technical field $X$, and to name its parts '$X$ science' and '$X$ engineering'.[9] This paper argues[10] that computer science and software engineering do not fit the traditional paradigm because the science bridge between them is missing. Mathematics has an important place in computer science and in software engineering, but not its usual role as the handmaiden of science.

[8]The relative size of the circles in Fig. 1 has no significance—there is no way to compare such 'apples and oranges' as the number or importance of differential equations vs. electronic components.

[9]Sometimes this is merely pretentious (e.g. $X =$ 'hair styling').

[10]A referee makes the nice point that the paper's argument is itself not scientific—there are no facts that could falsify it.

## 2. COMPUTER SCIENCE?

In naming a discipline, the suffix '–ology' ('biology') is common, but 'computology' does not trip well off the tongue. Neither are the analogs of 'physics' or 'chemistry' promising. 'Informatics' (by analogy with 'mathematics'?) is good, but does not directly include the calculational, computing aspect of the field. Appending 'science' sounds best.[11] If the heart of the computer science discipline is not the study of machine hardware ('computer engineering'), nor of using hardware and software to solve problems in other fields ('applied computing'), then just what is it? There is an immediate difficulty: its established name begs the question. A statement like: 'Computer science is (or isn't) a science', is just linguistic play unless its first two words are a mere name, not a description.[12] This paper will hereafter use 'CS' as a neutral name for the discipline.

### 2.1. Software

Most of a degree program in undergraduate CS is devoted to software, that is, to programs and programming. Computer programs are most certainly artifacts, 'devices' that programmers devise. But attempting to apply the mathematics/science/engineering paradigm to software, science with its theories and natural laws is missing. What laws do programs obey? What experiments might be done to falsify theories about programs? What is the science of software, that in the traditional paradigm would underpin software engineering? Those in the field generally acknowledge that the underlying science is lacking, but they look forward confidently to the day when it will be discovered.[13] To the contrary, there is reason to doubt there will ever be a science of software. Programs and programming have a made-up, abstract nature that is disconnected from external constraints. There can be no software natural laws because the whole thing is an arbitrary human invention. No statement about software could be falsified[14] by an experiment, because there is no 'reality' for software that decides truth.

[11]An aphorism of unknown origin: 'If a discipline has "science" in its name, it isn't one'. Maurice Wilkes called his chair at Cambridge 'Professor of Computer Technology' because he said he had once been a scientist and knew the difference.

[12]One professional organization of the discipline is the Association for Computing Machinery (ACM), which also has a name problem—it is not a trade group whose mission is to tout computers. The ACM membership has steadfastly resisted a name change for at least 40 years. Had mathematics been called 'number science' it might have experienced similar difficulties.

[13]For example, in Hoare's Turing Lecture [9] he speaks of 'laws' and 'science' of programming to come. But he also acknowledges (in denigrating Ada) that: 'Almost anything [dreadful] in software can be implemented, and even used, given enough determination'. He implies that the US Department of Defence supplied the determination to the tune of \$100 million. In science, determination does not suffice: no DoD spec calls for a contractor to (say) transmit data at twice the speed of light.

[14]Mathematical theorems about software in general could be falsified (for example, the number of steps required to sort $n$ elements of a list is proportional to at least $n \log n$). Statements about a particular, fixed program can be falsified,

CS is largely concerned with a skill (programming) or with the technological details of particular devices (programs, e.g. an operating system). There is a better parallel with vocational subjects (e.g. diesel engine repair) than with science education. No one in CS likes the vocational analogy, though those in industry sometimes criticize academics for failing to teach practical programming. CS aspires to the high ground of underlying principles. In the parallel, thermodynamics is of little use in practical diesel repair, but without thermodynamics a mechanic is unlikely to design a better diesel engine. CS does supply an engineer with important ideas in the form of algorithms and their analysis. For example, Knuth's programming books[15] are invaluable, but not science.

## 2.2. CS mathematics

The mathematics of CS itself, theoretical CS, is often cited as the foundation of computing. Theoretical CS seeks mathematical descriptions of computation, most importantly of programs. For one example, in recursive function theory a program is a Gödel number in an indexing of the partial recursive functions; the indexing itself is the programming language [10]. The axioms of this theory describe crucial semantic and syntactic properties of the indices (programs).[16] Theorems of recursive function theory establish limits on program analysis, e.g. there is no algorithm to determine if two arbitrary indices (programs) compute the same function.[17] For another example of theoretical CS mathematics, parsing theory describes a programming language as a set of strings that can be recognized by an automaton or generated by a grammar.[18] These two theories abstract away many important and difficult properties of programs; years of research have provided much more precise theoretical descriptions, for example in detailed formal semantics.

In speaking of (mathematical) theories of theoretical CS, it is easy to slip into an apparent analog of applied mathematics by giving the mathematical entities names from practical computing. Is this slip only linguistic? Is 'program' just a folksy name for 'Gödel number'? Or is there a real world of real programs, which map onto the entities of theoretical CS mathematics? This question lies at the heart of the larger question, 'What is CS?'. In the famous 1967 letter to *Science* from Newell *et al.* [13], they take the 'phenomena surrounding computers' as obviously real, but their case is more persuasive for the electromagnetic physics and electrical engineering of the machines than for programs.

A working programmer, finds (say) the Linux kernel source in C to be all too real. Yet each aspect of even the most elaborate program has a mathematical description that comes from theoretical CS. Programs have no presence in the world apart from these descriptions.[19] The Linux kernel came from human minds, from ideas that stretch back to von Neumann's first subroutine. Of course the pits burned onto a disk are real, but not the essence of Linux. All its properties: syntax, functional behavior, run time, process model, etc., have mathematical existence only. What makes a program appear 'real' is how complicated its properties become, clean theoretical CS mathematics riddled with special cases and exceptions. Nevertheless, messy mathematics is still mathematics.

In contrast, applied mathematics requires a measurement-based scientific discipline to map onto a mathematical theory; thus, the domain and range of the mapping are quite different. On the mathematics range, abstract objects of a theory are images of reality; on the science domain the concrete objects are subject to measurement. In science, the real-world side imposes its version of 'truth'—that is, measurement—on the abstract side. Should measurements fail, all that a scientist can do to save the phenomena is to tinker with either the mathematics or the correspondence of the mapping; 'stubborn facts' of the world cannot be adjusted. Neither does tracing out a mathematical proof in its real-world preimages say anything about the mathematics. For example, in the solar-system model of an hydrogen atom, the electron should spiral down into the nucleus, emitting light of all wavelengths. That the observed spectra are instead discrete does not discredit the mathematics describing orbits and lost energy.

CS has no scientific domain to map to its mathematics, yet there certainly are 'application' mappings. For example, the Linux kernel is mapped to a certain Gödel number, and we expect that theorems of recursive function theory will apply to the kernel. If a cosmetic change is made in the kernel source, there may be difficulty verifying that it *is* merely cosmetic, because the change alters the Gödel number and there is no algorithm to decide if the indices before and after compute the same function. A further consequence of recursive

notably the statement, 'This program does _____ as intended'. These apparent exceptions will be discussed in Sections 2.2 and 3.1 to follow.

[15]He entitled them *The Art of Computer Programming*. Art may have laws, but they are invented by artists.

[16]The semantic axiom asserts the existence of a 'universal machine' index; the syntactic axiom concerns effectively embedding arguments of a function at one index into another index [11].

[17]Technically, programs do not map perfectly to the mathematics of undecidability because of the unlimited nature of the theory. A more complicated theory with a better mapping has theorems like, 'There is no *tractable* algorithm ...'.

[18]For pushdown automata, the equivalent grammar is context-free, and it can be proved that languages so defined cannot require certain agreement conditions within a program, such as that the same identifier may not be declared more than once [12]. More powerful context-sensitive grammars can express such conditions.

[19]Natural languages are an intriguing borderline case: they exist outside any mathematical description, as programming languages do not. This fundamental distinction makes it almost impossible for a linguist and a computer scientist to rationally discuss 'What is a language?' A statement like '$X$ is context sensitive' could be falsified (in psycholinguistics) for $X$ = 'English', but is a matter of parsing-theory definition for $X$ = 'Algol 60'. $X$ = 'Fortran' is particularly intriguing, because Fortran predated parsing theory, yet today its compilers parse Fortran according to a grammar, and from the outset the language was defined by its compilers.

function theory is that there is no algorithm[20] for finding test values whose success will establish this equivalence [14]. Similar applications of parsing theory, and of computational complexity, etc., can be easily given. These mappings are what CS is all about, so they will be considered in detail in Section 2.3 to follow.

## 2.3. Applying theoretical CS mathematics

When theoretical CS is applied to programs, the application mapping is from one mathematical domain to another. Programs are mathematical entities whose properties are defined in one theory; another theory 'explains' something about programs. When we say that a programming language 'is a Gödel numbering', or 'is generated by a context-sensitive grammar' (or, among many other diverse examples, a data type 'is a many-sorted initial algebra' [15], etc.), we are trying to understand and explain CS entities by mapping their definitional mathematics onto other mathematics. In the typical case, the explanatory theory is more abstract, simpler, than the defining theory. For example, the accurate description of a C program being executed on a modern computer is very complicated indeed, involving not only C features like uncontrolled pointers, but hardware limitations on arithmetic precision and storage capacity, special cases of hardware efficiency, optimization by a compiler and operating-system algorithms (which in turn involve all these aspects for other programs running at the same time). Each facet of the program's definition is precise mathematics, but the composite almost defies understanding.[21] Recursive function theory 'explains' that these messy programs are 'really' only Gödel numbers, and hence no testing algorithm exists, etc.

Explanatory theories do explain. Trying to think about large issues in the presence of too much detail is difficult. Recursive function theory strips away the detail, but its definitions and axioms are enough to prove theorems that settle big issues. It is also instructive to consider the case in which the axioms of the explanatory theory do not hold[22] for programs. Then as usual, a different range of explanatory mathematics can be found or the mapping altered, but in CS there is a new option: Change the definitions of the domain theory. If we want programs to be Gödel numbers (or, to dodge unsolvable problems, we *don't* want them to be Gödel numbers), there is no reason not to change the definition of programs to make it so. Neither side of the mapping is sacrosanct.

As might be expected, experiment does not play its traditional scientific role in CS. CS experiments do not refute or support the mathematical theories on either side of the explanatory mapping. Instead, CS experiment is largely a mechanism for getting the mathematics on both sides into correspondence. For example, an operating system may be constructed to explore the ideas behind it. But if it should fail to work as expected, no theory is refuted; the author merely adjusts things by making a different system, or adjusts expectations to match what was implemented. This freedom to alter either side of the mapping is characteristic of working with two abstractions; a scientist does not have that freedom. The usual experimental computer scientist has more in common with a mechanic trying to repair an engine than with experimental science. Certainly the idea that experiments are amassed and theories induced from them, if it ever had any validity in science [16], is not appropriate for CS.

Milner [17] suggests an important role for experimentation: to discover not a mathematical theory itself, but the concepts and ideas a theory should express. He says that the best way to invent useful explanatory mathematics is to first immerse yourself in the practice to be explained. Gernot Heiser's ERTOS group exploring trustworthy systems [18] is a focused application of this idea—Heiser credits ERTOS' success to formalists studying real operating systems, whose implementers have learned theory. The crucial difference between these efforts and scientific theory building is exposed when something goes wrong: in science, the theory has to be scrapped; in CS it can be saved by modifying the (mathematical) definition of the 'phenomena'.

Physical scientists might say that the world and its real laws are the bane of theoretical work. How many elegant theories have been scuttled by hard facts? But given a wide-open field devoid of facts, theories are pure mathematics, difficult to evaluate. Elegance, depth, insight and ease of application are important goals for CS theories, as for any mathematics, but evidently these are subjective. Lacking any experimental failure, the proponents of an unproductive theory will be reluctant to give it up. Thus the CS landscape is littered with dead theories that were never refuted (for how could they be?), but are not used. The development of Algol 68 is a good example. Its defining committee had a number of very good ideas: They decoupled a finite-state input-token mapping from a two-level van Wijngaarden grammar that captured context conditions like type equivalence; and they defined its semantics operationally over that grammar. Despite their success in applying these theoretical ideas to a full-blown programming language [19], these techniques are not much used in programming-language definition today.[23]

[20]Since the kernel is just one program and the change may be of some restricted kind, there may very well be algorithms for this special case, but they must fail in general.

[21]Indeed, the facets are developed separately and can come into conflict. An operating system may undo a compiler optimization or nullify hardware efficiency.

[22]Technically, the explanatory theory is required to be consistent. Otherwise, no theorem has any significance, since its negation is also a theorem. It is a kind of 'falsification' if a contradiction is exposed by the application, but what falls is not a scientific theory but the mathematical one. If recursive function theory is consistent, then its application to programs better not turn up any logical surprises.

[23]The example illustrates that even dead theories have lasting benefits: although Algol 68 and its technical definition are dead, the mathematics allowed its committee to understand important ideas like static type agreement and orthogonal design, today in wide circulation.

### 2.4. What kind of 'Science' could CS be?

The computing literature contains many assertions that CS is truly a science.

Richard Snodgrass has proposed a 'natural science of computing' he calls 'ergalics'. He gives Denning's principle of locality as an example of an ergalics law. Ergalics satisfies Popper's criteria for science when it falls back on psychology or sociology of human programming for theories. Snodgrass and Morrison [20] describe inducing 'laws' from observations on program executions. They admit that program behavior is in principle mathematically defined, but argue correctly that experiments are needed because the mathematics is intractable. There is a cognitive dissonance in needing to experiment with our own creations because we defined them to be incomprehensible. Scientists do not have to kick themselves for creating a difficult universe. The 'laws' induced for a program could very well be wrong.[24] The locality example illustrates both points: Locality is explained as a result of how programmers think about data (science that is part of psychology); and, although locality is striking in parts of many programs, it is not universal (or working-set page-replacement algorithms would work much better than they do!) and many programs do not exhibit it at all (for example, one that searches a large space sequentially). Section 3.1 to follow will further consider narrow statements about particular programs.

Other papers do not so much deny Popper's definition as ignore it.

In two columns in *Communications of the ACM* with provocative titles [21, 22], Peter Denning calls CS 'science', even 'natural science', because CS shares some characteristics with science. For example, one of his arguments is that computing is making important contributions to natural sciences, a kind of 'guilt (or in this case, science) by association'. For another, he considers the 'surprise' potential of a discipline, which sciences share with CS. But religions, the primary world view that Popper excluded with his definition, are also full of surprises.

In his later column [22], Denning speaks not of CS but of 'information processes', to make the point that information is a real-world entity. Perhaps Denning sees the day to come when the information-rich parts of many sciences will be regrouped and named 'information science'[25]. Existing natural sciences will probably not relinquish territory when they consider information. The biologist will claim (say) genetic information for biological science. 'Information' seems to just add a dimension to 'applied computing'. But there may be a scientific niche open for CS: the storage and transformation of information in general. Shannon's law [23] is certainly open to falsification, and electrical engineers who use it fit the traditional paradigm with CS information theory as their science. This is a slippery slope; coding theory seems similar to recursive function theory in that it can not be wrong unless the mathematical theory is unsound[26]. In any case, a scientific 'informatics' does not seem to underlie the programming core of CS, which is artificial and mathematical.

Hoare is the most articulate and respected proponent of 'scientific' CS. In writing [24] and speaking [25, 26] about his 'grand challenge' to apply verification to substantial computer systems, he makes frequent reference to science and to laws of programming. However, Hoare's laws are not physical observations, but unabashedly mathematics. '... [P]rograms themselves, as well as their specifications, are mathematical expressions' [24, p. 686]. For him, mathematics is 'science' because of its generality, striving for perfection and unification, originality, formality, etc. Popper's falsifiability criterion does not come up. The distinction between Hoare's statements and the thesis of this paper would disappear if for his 'scientific' theory, laws, etc., we substitute 'mathematical'. His many important insights and contributions would be unchanged by this substitution, notably his description of the most promising basis for a theory: it should describe the components of systems and interfaces, and permit proofs that the whole will behave properly if its parts do.

Hoare says that the ultimate question a scientist must ask of laws is: 'How do we know' they are right? Here it is plain that he speaks of mathematics, not observational science, since '... this final assurance can in principle be given by mathematical reasoning and proof ...' [25, p. 257]. That is the way it is done in mathematics, but not in (Popper's) science, which excludes subjects in which disputes might never be settled because observational falsification[27] is not a possibility. Faced with two competing mathematical theories that explain computer systems, even granting that neither contests the other's definition of 'computer system', there is no way to objectively compare them—supporters of each could dispute profitlessly about their merits forever.[28] Thus the difficult road to establishing CS laws is made more difficult, because trial and error will not work—there may be no agreement on what constitutes error.

In his Turing-award lecture [27] Juris Hartmanis is less dogmatic than Hoare, concluding that '[CS] differs so basically from the other sciences that it has to be viewed as a new species among the sciences ...' [27, p. 39]. His background in physics and his foundational work in computational complexity make him a thoughtful commentator on CS, hard to force into an easy

[24]Engineers are used to getting on with their jobs despite intractable mathematics. Perhaps what ergalics should study is safety factors necessary to use its 'laws' in design. Unfortunately, the lack of continuity in digital systems make this difficult or impossible.

[25]'Informatics' would be a good name.

[26]As an aside, is it coincidental that the most dramatic and unexceptionable results of CS theory, in computability, complexity, coding, etc., are negative?

[27]Hoare does allude to this sort of scientific proof, and to the role that computers play in analyzing experimental data, but he does not suggest CS observations to falsify his mathematical laws.

[28]If just one theory is consistent, it might be judged the better; if both are inconsistent they are trivially the same.

mould. Although, like Hoare, the 'science' he clearly sees in CS can be identified (mostly) as mathematics, his insights are not reconciled with this paper simply by substituting 'mathematics' for 'science'. His treatment of CS theory agrees with this paper's thesis: '... theories do not compete ... for which better explains the fundamental nature of information. Nor are new theories developed to reconcile theory with experimental results ...'; '... there are no experiments ... which could resolve ... problems [like $P = NP$?] ...' [27, p. 40]. Hartmanis's take on CS experiments is particularly good; he calls them 'dramatic demos to show the possibility to do what was thought to be impossible' ([27, p. 41] [juxtaposition of several phrases]). He sees that engineering draws directly on CS theory without the intervention of a science: '[CS] is the engineering of mathematics' [27, p. 41].

*Computing Surveys* reprinted Hartmanis's Turing lecture in its March, 1995 issue, with 12 short essays responding to the issues he raises. One essay (Stewart, p. 40) mentions Popper's definition of science and the falsification criterion, but fails to apply it to CS. Several of the essays agree with Hartmanis that CS is something very special, '... both [science and engineering], neither, and more [mathematics] ...' (Freeman, p. 27). Several are impatient with the 'What is CS?' question as a counterproductive discussion about labels. Wulf (p. 56) rejects categorization as '... a failure [of other disciplines] that ... [CS is] not required to emulate'. In response, Hartmanis believes that understanding the nature of CS '... will lead to more shared values in the community, better understanding ... by other scientists, and even to better research' (p. 59). These are goals of a report Freeman cites, by the Task Force on the Core of Computer Science [28], which begins with an attempt to define the field. This report views CS as an amalgam of theory (mathematics), abstraction (science) and design (engineering). Its definition of design is weak (it includes 'design and implement the system'), and its science includes 'analyze results', with the unquestioned assumption of real-world measurements. According to the thesis of this paper, the report's 'theory' and 'abstraction' would be much the same; 'analysis' is mathematical, not scientific.

## 3. SOFTWARE ENGINEERING

Software engineering (SE)[29] is placed in a peculiar position by the lack of science in CS. Whereas other engineers design to their clients' needs within the constraints of physical law, software engineers have nothing to work from except what the customer wants. This may well be capricious and ill-conceived, even illegal or dangerous. But if a software engineer says, 'You can't do that', it reflects on the engineer's ability, not the implacable stubbornness of reality.[30] The role of mathematics in other kinds of engineering is to check that the design fits the world, and so the device will work; in software, 'working' is reduced to pleasing a client. Einstein believed that God was subtle in framing the laws of physics, but not so 'damn mean' that the laws cannot be captured in elegant mathematics. An incompetent business executive, in framing laws (requirements) for software, *is* damn mean.

### 3.1. Formal methods

A particular program's properties (principally its functional behavior) can be very complicated. But as argued in Section 2.3, each program has a mathematical definition that is its only reality. Turing [29] and others originally defined program functions 'operationally', by describing the steps taken in an imagined 'machine'. Today it is also common to use a 'logic' for semantic description[31] [31]. Given the precise description of what a program $P$ *does* do, it is natural to come up with another mathematical description of what $P$ *should* do (according to human requirements),[32] and to prove mathematically that the two descriptions agree, i.e. that $P$ is correct. This plan goes under the name of 'formal methods' in SE.

The situation is exactly as described in Section 2.3, but for a single program. The explanatory mathematics is what the program should do; the definitional mathematics is what it does do. Theorems of the explanatory mathematics express behavior that the defined executions of a program are expected to observe. Just as in the general case, the 'scientific' view that a program is a real-world object whose correctness might be falsified by testing experiments is wrong in principle. Consider what happens when an experiment (test) fails. 'Not to worry!' says the software engineer, and sets about 'debugging'. Either the formal requirements or the program (or both) are adjusted to bring them into line. No one has any doubt that this can be done.[33] Perhaps after many experiments (tests), the requirements description and the program will converge so that a proof of correctness can be carried out. Nothing like this is possible for a scientific experiment—the real-world side cannot be adjusted to fit and proof is out of the question.

Those who espouse formal-methods program development are sometimes called 'Formal Methodists'. They know that their scheme involves two mathematical domains and a mapping

[29]Although constructing software *is* engineering by almost any rational definition, there is no harm in avoiding the value-laden name by writing 'SE'. But it most often occurs in speaking of software developers as 'software engineers', for which there is no easy textual remedy.

[30]A star-wars project general, responding to persuasive arguments by those who believe its software cannot be constructed, is reported to have said something like, 'I will support those who say they can do it, not those who say they can't'. Inventors of perpetual-motion machines will be disappointed to learn that star-wars dollars are not for them.

[31]Robert Floyd did the seminal work [30] that suggested the connection.

[32]What is recommended by Dijkstra and others is of course the other way around: it is the programmer's job to come up with 'does' in response to 'should'.

[33]It is acknowledged that debugging is difficult, but no amount of fruitless effort will convince the engineer that it is hopeless. Indeed, the program will usually be released as is when debugging time runs out, to be fixed later. That this almost never happens deters no one.

between them, not a science of programs. There are two ways to exploit this unique situation:

***Practice can be changed to fit theory*.**

When Eddington said, 'It is also a good rule not to put overmuch confidence in the observational results that are put forward until they are confirmed by theory', he was joking about physics; it is a literal description of formal-methods development. Edsger Dijkstra is usually credited with the thesis that programming languages should help programmers to write clear specifications and prove programs correct. He is certainly behind stigmatizing the **goto**, a construct that is a horror to formalize. Many of the foremost contributors to verification theory and technology have recognized the opportunity to make formal methods work better by changing the mathematical definition of 'program' [32, 33].

Programming languages and SE design techniques have grown up without much direction. Anyone who has attempted a correctness proof sees that there are parts of widespread languages like C that if eliminated would make things ever so much easier.[34] If a language forbids the use of powerful but hard to verify constructions, it might be harder to use but also harder to misuse. In some cases power is needed for efficiency, but excluding interactive games, most consumer software could forswear pointers, for example. Unfortunately, general recursion (or unbounded iteration) and dynamic storage are proof-challenging constructions needed in almost any useful program.[35] Perhaps there is a place for more radical restrictions than have been proposed. Some programs with a finite state space are trivial to verify (by exhaustive test with execution-state monitoring); a language of such programs could solve many problems safely, including initial segments of wider problems.

***Engineers are part of a solution, not a problem*.**

Theoretical research has largely taken the engineering processes that follow formal requirements as given; but they are as artificial as formal methods themselves. No other engineering discipline is free to invent its own design rules without consulting implacable reality; SE has so far made little use of this freedom. The efforts of present Formal Methodists are directed toward eliminating conventional programming (replacing engineers with mathematicians?) by constructing formal requirements that can be compiled into code. When engineers object to Hoare's grand challenge [32], they suggest that shifting design from implementation to specification just raises the real problem to a different plane: they expect to fail to capture what is really wanted there, too. It would be productive to admit that engineers are a valuable and necessary talent pool, and to find design rules for them to do what they have done so well in other fields, on whatever level works best.

Hoare looks forward to a meeting of CS theory and SE practice [26], when software will be not the weak link in engineering systems, but the strongest. His prediction is all the more credible when CS is seen as mathematics rather than science: Only CS has the ability to define its entities for SE success, leaving the error-prone bits to the other engineers! In a sense SE just took on too much too soon. Had it started modestly with tractably provable programs, perhaps responsibilities from other engineering branches could have been slowly assumed and software spared its poor record.

### 3.2. SE theory

In most engineering disciplines, the theoretical engineer is an honorable maverick, bridging the gap to science[36] and even to art. For example, a theoretical structural engineer does basic research in materials science and is also part architect [34]. But SE theory is just narrower CS theory,[37] mathematics, not science. SE theories are (mathematical) explanations for the (mathematical) entities of SE.

Mistaking theoretical SE for science can be detrimental to SE research:

***Unnecessary implementation is expected*.**

For explanatory mathematics, it is specious to ask for a real-world realization, because there is no science, no 'real'. Indeed, the other mathematics being explained (software) is itself a kind of implementation of an explanatory theory. It may even be demanded that a 'tool' be constructed for the explanation, that is, a program that allows the explanatory mathematics to be 'run' mechanically. Implementation does expose mistakes and aid understanding—it is easier to think about concrete examples than abstractions. An implementation also shows off its

[34]Pointers, dynamic arrays and dynamic union types come immediately to mind.

[35]There is some confusion about so-called 'safe' languages like Java in which explicit run-time exceptions are always thrown by conditions like buffer overflow. There is no question but that safe programs are far easier to test and debug. But they are really safe only when there is a way to handle the exception and continue, or when the exception can be proved impossible (in which case there need be no exception processing!). When the Ariane-5 rocket had to be destroyed, a routine in its control program had thrown a type exception, but no recovery was possible. Error messages are no use when a routine *has* to work, and exception safety is hard to prove.

[36]Value judgements in the mathematics/science/engineering hierarchy are clearly defined: 'pure' mathematics is better than 'applied', yet applied mathematics is what the 'theoretical' scientist does, who is in turn better than the lowly 'experimental' scientist, who might be a 'theoretical' engineer. The 'practical' engineer is at the end of the line with nobody to lord it over.

[37]Or CS just wider Formal Methodism?

author's programming skills. However, implementation can take years of expensive work. Good ideas get lost because they are evaluated not as explanations, but as tools. Imagine Turing's 1937 paper being delayed because a reviewer insisted on construction of a practical Turing machine.

***Superfluous experimental evidence is expected.***
To demonstrate that an explanatory theory does explain some SE entity, it is only necessary to check that the proposed theory's axioms hold for the mathematics defining that entity. Explanatory mathematics does not need 'practical' evidence as a scientific theory would. For example, suppose Peano arithmetic is used to explain counting stones ('Stones are really natural numbers ...'). It is enough to check Peano's axioms, and unreasonable to expect a vast experiment in which thousands of stones are partitioned and counted to check that $1 + 0 = 1$, $2 + 2 = 4$, etc. It is not impossible that this procedure will turn up a startling new insight about stones, but it is immensely wasteful. Checking axioms is proper work for the theoretician; counting stones is not.

***Spurious surrogates are studied.***
Because nothing in SE is 'real', experiments often use surrogates that miss the point. For example, software testing has long flirted with the idea of tests 'covering' the structure of a program.[38] Many variations of this idea have been proposed, the quality of a new proposal almost universally judged by some other kind of coverage. It is an open question which (if any) kind of structural coverage actually achieves what testing should accomplish (finding failures, or demonstrating that failures are unlikely). So to compare coverages is like calibrating a new instrument against an old one of unknown accuracy. In the worst examples, the researcher winds up comparing the coverage to itself. Having implicitly defined as 'good' a technique being proposed, an experiment measures the extent of something highly correlated with it, and concludes that the technique is good[39]!

Each of these misdirections in SE carries twin dangers. The more evident is that important new ideas may be spuriously rejected. On the other side, trivial ideas are inflated with 'scientific' trappings.[40] One of this paper's referees raises a wider danger: SE mathematics, treated as science, can contaminate the real science in an application domain. A scientist using a program can confuse its mathematical properties with real experimental checks on its results. The referee gives the example of an image-processing algorithm validated by its convergence rather than by examining the real objects whose images are involved.

Practical software engineers are impatient with theoretical mathematical explanations, sometimes with good reason. Acknowledging that SE theory is mathematics, they say in effect, 'Since you are making this up, why isn't it easier to understand and use?' Just as the software engineer cannot point to physical laws to constrain a crazy customer, so SE cannot point to them in justification of its theory.[41] When a scientific theory explains hard facts and captures them down to the fifth decimal place, it seems worth knowing. When there are no facts, but just (say) a formal description and a program, why pay any attention to the one that will not be delivered and paid for? One answer is: Because the formal description is a clearer, simpler explanation than the program. There is a long way to go.

## 3.3. The process vs. product controversy

In passing, there is an entirely different kind of SE research, not subject to the critique above. The managerial and technical practices that human beings employ when they develop software are called 'software process'. Process research is a social science, primarily sociology. The real world is certainly involved, and there are traditional sociological theories and experiments to check their validity that are good (and bad) science.[42] If results are slow to appear, it may be because software development is a complex, time-consuming, expensive process that is difficult to study, more so because its participants are diverse human beings given the considerable scope to exercise their individual talents.

Normal design in other fields of engineering has its basis in scientific analysis of the engineered product, but in SE the lack of science makes 'process' an enticing alternative. Organizations and software engineers are exhorted [39] to clock

[38]In the simplest example, a program test suite achieves 'statement coverage' iff running it causes each and every statement of a program to be executed in the course of testing.

[39]A 1989 comparison study [35] rated the branch-coverage technique as best, since it achieved the best branch coverage (100%!). A more recent experiment [36] had subjects use an interactive tool to test spreadsheets and achieve a kind of dataflow coverage, against a control group that tested without the tool. The quality of the resulting tests was judged by this same dataflow-coverage measure, leading to the conclusion (supported at the $p = .0001$ confidence level!) that the tool-supported tests are better. (In a subsequent journal publication [37] the circular reasoning was removed, along with the control group.)

[40]There is a devastating 1978 parody entitled 'Publication policy in a primitive scientific society' [38] written about management science, which applies in spades to SE. It describes 'Roller Science'—placing logs under heavy objects to move them—and the way in which this 'discipline' evolves, splits into theoretical and practical camps, etc. Theoreticians look at cross-section equations, ellipses, squares, etc., and stumble on (but reject) rollers whose cross section is a circle with a smaller concentric circle removed; a practical paper says, 'Rollers made from trees work better if you cut off the branches'.

[41]A physicist could say, 'Relativity may be difficult, but it's needed to describe light'.

[42]Many of the 'laws' located by internet search are sociological, e.g. 'Adding manpower to a late project makes it later' (Fred Brooks?). But some are Zen-like: 'Lyall's Conjecture: If a computer cable has one end, then it has another'. It is bizarre that a substantial number of postings found by the query 'information theory laws' concern 'creation science'.

what they are doing, keep good records, and hence achieve greater control of software development. Control is good for managers, but less so for working engineers. To quote one aerospace engineer, 'Management is not a skill or a craft or a profession but a command relationship; a sort of bad habit inherited from the army or the church'.[43] The crucial property of applicability is at work here: engineers need something they can routinely apply, and so they will use (or be forced by management to use) what there is, without the luxury of evaluating its validity.

There is a 19th-century parallel to software process control in the machine shops of the industrial revolution and Fred Taylor's 'scientific management' [41]. (Unable to stomach the value-laden name, his opponents called it not 'SM' but 'Taylorism'.) Taylorism is often called time and motion study, the breaking down of labor-intensive tasks into steps that can be performed mechanically. The Marxist position would be that this deskilling of labor is done by capital-intensive industries in order to eliminate creative human beings from the production process [40]. On the other hand, a humanistic analysis of the software industry might say that its lack of normal design makes it the last engineering bastion of human creativity. Without comment, here are three observations about this situation:

(i) Software engineers work long hours at very demanding tasks, and they mostly do not complain.
(ii) The human toll of SE work is great. It is a common story that: 'We delivered the compiler, but none of the marriages in the development group survived'.
(iii) Any kind of union organization is anathema to software engineers.

## 4. SCIENCE-FREE CS AND SE

Figure 2 is a Venn diagram illustrating the situation in computing, its most striking feature being the empty intersection of both 'CS' and 'SE' with the 'Science' set.[44] The central area in Fig. 2 (shaded) is theory of programming and programming languages. Some other interesting areas are indicated: The vertical striping marks CS outside of both mathematics and programming theory, where (say) operating systems like UNIX lie. The cross-hatched area is non-mathematical algorithms, for example, the many clever sorting techniques like radix sort. Formal methods includes the horizontally striped area, but the extent to which it overlaps all of SE and programming theory is controversial. The part of SE at the bottom of the figure (unmarked) contains management-oriented 'software process', discussed in Section 3.3. An important property of Fig. 2 that distinguishes it from Fig. 1 is that there is little reason for the

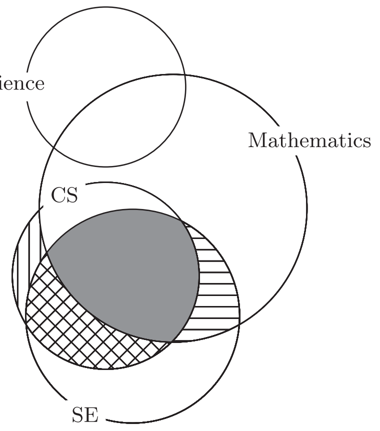

**FIGURE 2.** Inclusion relationships among computing disciplines.

boundaries to shift in Fig. 2; it will not help SE to incorporate (say) operating systems.

Does it matter whether CS is called science or mathematics? Shakespeare is often quoted to support the irrelevance of names: '. . . that which we call a rose// By any other name would smell as sweet . . .'. Juliet did not want Romeo's name to matter, but in the end their surnames killed them. Names matter because they color everything around them. So long as CS is seen as a science, its theories and methods will be evaluated using inappropriate standards. Important research directions will appear to be closed, while others without substance appear to be promising. Acknowledging what CS and SE really do is a first step in doing it better.

[43]The context is that at Lucas Aerospace Engineering there was an industrial action in which engineers tried to restructure the failing organization [40].

[44]Informatical aspects of sciences like biology, and sociology of programming, etc., do fall in 'Science'; perhaps CS should have an 'informatics' piece of this pie, as indicated in Section 2.4.

## ACKNOWLEDGEMENTS

Lorenzo Strigini pointed me to Bill Addis's work. The anonymous referees suggested several important references that I initially discounted, and contributed many thought-provoking ideas.

## REFERENCES

[1] Nahin, P.J. (1988) *Oliver Heaviside: Sage in Solitude, The Life, Work, and Times of an Electrical Genius of the Victorian Age*. IEEE Press, New York.
[2] Bridgman, P. (1928) *The Logic of Modern Physics*. Macmillan, New York.
[3] Popper, K.R. (1959) *The Logic of Scientific Discovery*. Hutchinson and Co., London.
[4] Hardy, G.H. (1940) *A Mathematician's Apology*. Cambridge University Press, Cambridge (reissued, 2004).
[5] Petroski, H. (1985) *To Engineer is Human: The Role of Failure in Successful Design*. St. Martin's Press, New York.
[6] Addis, W. (1991) *Structural Engineering: The Nature of Theory and Design*. Ellis Horwood, Chichester.

[7] Vincenti, W.G. (1993) *What Engineers Know and How They Know It*. Johns Hopkins University Press, Baltimore.

[8] Anon. (1955) *Preferred Circuits Navy Aeronautical Electronic Equipment*. Government Printing Office, Washington, DC. NAVWEPS 16-1-519.

[9] Hoare, C.A.R. (1981) The Emperor's old clothes. *Commun. ACM*, **24**, 75–83.

[10] Kleene, S.C. (1952) *Introduction to Metamathematics*. North-Holland, Amsterdam.

[11] Rogers, H. (1967) *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York.

[12] Floyd, R.W. (1962) On the nonexistence of a phrase structure grammar for ALGOL 60. *Commun. ACM*, **5**, 483–484.

[13] Newell, A., Perlis, A.J. and Simon, H.A. (1967) Computer science. *Science*, **157**, 1373–1374.

[14] Howden, W.E. (1976) Reliability of the path analysis testing strategy. *IEEE Trans. Softw. Eng.*, **2**, 208–215.

[15] Goguen, J., Thatcher, J. and Wagner, E. (1978) An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In Yeh, R. (ed.), *Current Trends in Programming Methodology*, pp. 80–149. Prentice Hall, Englewood Cliffs, NJ.

[16] Kuhn, T. (1996) *The Structure of Scientific Revolutions* (3rd edn). University of Chicago Press, Chicago.

[17] Milner, R. (1987) Is computing an experimental science? *J. Inf. Technol.*, **2**, 58–66. Reprinted from LFCS inaugural address ECS-LFCS-86-1 (1986).

[18] Klein, G. *et al.* (2010) SEL4: formal verification of an operating-system kernel. *Commun. ACM*, **53**, 107–115.

[19] van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T. and Fisker, R.G. (1976) *Revised Report on the Algorithmic Language Algol 68*. Springer, Berlin.

[20] Morrison, C.T. and Snodgrass, R.T. (2011) Computer science can use more science. *Commun. ACM*, **54**, 36–38.

[21] Denning, P.J. (2005) Is computer science science? *Commun. ACM*, **48**, 27–31.

[22] Denning, P.J. (2007) Computing is a natural science. *Commun. ACM*, **50**, 13–18.

[23] Shannon, C.E. (1949) Communication in the presence of noise. *Proc. Inst. Radio Eng.*, **37**, 10–21.

[24] Hoare, C.A.R., Hayes, I.J., Jifeng, H., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M. and Sufrin, B.A. (1987) Laws of programming. *Commun. ACM*, **30**, 672–686.

[25] Hoare, C.A.R. (1987) The ideal of program correctness. *Comput. J.*, **50**, 254–260. Third Computer Journal lecture, presented in 1986.

[26] Hoare, C.A.R. (2009) *The science of computing and the engineering of software*. http://www.infoq.com/presentations/tony-hoare-computing-engineering. Keynote address, QCON.

[27] Hartmanis, J. (1994) On computational complexity and the nature of computer science. *Commun. ACM*, **37**, 37–43.

[28] Denning, P.J., Comer, D.E., Gries, D., Mulder, M.C., Tucker, A., Turner, A.J. and Young, P.R. (1989) Computing as a discipline. *Commun. ACM*, **32**, 9–23.

[29] Turing, A. (1936) On computable numbers, with an application to the entscheidungsproblem. *London Math. Soc. Ser. 2*, **42**, 230–265. Correction, ibid **43**, 44–46.

[30] Floyd, R.W. (1967) Assigning Meanings to Programs. In Schwartz, J.T. (ed.), *Proceedings Symposium Applied Mathematics*, pp. 19–32. American Mathematical Society, Providence, RI.

[31] Hoare, C.A.R. (1969) An axiomatic basis for computer programming. *Commun. ACM*, **12**, 576–583.

[32] Josephs, M. *et al.* (1987) Discussion on Hoare's 'The ideal of program correctness'. *Comput. J.*, **50**, 261–268.

[33] Hoare, C.A.R. (1987) Respone to discussion on 'The ideal of program correctness'. *Comput. J.*, **50**, 269–273.

[34] Addis, W. (2001) *Creativity and Innovation: The Structural Engineer's Contribution to Design*. Architectural Press, Oxford.

[35] Lauterbach, L. and Randall, W. (1989) Experimental Evaluation of Six Test Techniques. *4th Annual Conf. Computer Assurance (COMPASS)*, Gaithersburg, MD, June, pp. 36–41. IEEE Press, Piscataway, NJ.

[36] Rothermel, K.J., Cook, C.R., Burnett, M.M., Schonfeld, J., Green, T.R.G. and Rothermel, G. (2000) WYSIWYT Testing in the Spreadsheet Paradigm: an Empirical Evaluation. *Proc. ICSE 2000*, Limerick, Ireland, June, pp. 230–239. IEEE Comp. Soc., Los Alamitos, CA.

[37] Rothermel, G., Burnett, M., Li, L., Dupuis, C. and Sheretov, A. (2001) A methodology for testing spreadsheets. *ACM Trans. Softw. Eng. Methodol.*, **10**, 110–147.

[38] Rothkopf, M.H. (1978) Publication policy in a primitive scientific society. *Interfaces*, **8**, 43–45.

[39] Humphrey, W.S. (1997) *Introduction to the Personal Software Process*. Addison-Wesley, Boston.

[40] Cooley, M. (1982) *Architect or Bee? The Human/Technology Relationship*. South End Press, Boston.

[41] Taylor, F.W. (1911) *The Principles of Scientific Management*. Harper and Bros., New York.