Minds & Machines (2007) 17:135–167
DOI 10.1007/s11023-007-9060-8

Three Paradigms of Computer Science

Amnon H. Eden

Published online: 31 July 2007

Abstract We examine the philosophical disputes among computer scientists concerning methodological, ontological, and epistemological questions: Is computer science a branch of mathematics, an engineering discipline, or a natural science? Should knowledge about the behaviour of programs proceed deductively or empirically? Are computer programs on a par with mathematical objects, with mere data, or with mental processes? We conclude that distinct positions taken in regard to these questions emanate from distinct sets of received beliefs or *paradigms* within the discipline:

- The *rationalist paradigm*, which was common among theoretical computer scientists, defines computer science as a branch of mathematics, treats programs on a par with mathematical objects, and seeks certain, a priori knowledge about their 'correctness' by means of deductive reasoning.
- The *technocratic paradigm*, promulgated mainly by software engineers and has come to dominate much of the discipline, defines computer science as an engineering discipline, treats programs as mere data, and seeks probable, a posteriori knowledge about their reliability empirically using testing suites.
- The *scientific paradigm*, prevalent in the branches of artificial intelligence, defines computer science as a natural (empirical) science, takes programs to be entities on a par with mental processes, and seeks a priori and a posteriori knowledge about them by combining formal deduction and scientific experimentation.

A. H. Eden (✉)
Department of Computer Science, University of Essex, Colchester, Essex, UK

A. H. Eden
Center for Inquiry, Amherst, NY, USA

We demonstrate evidence corroborating the tenets of the scientific paradigm, in particular the claim that program-processes are on a par with mental processes. We conclude with a discussion in the influence that the technocratic paradigm has been having over computer science.

Keywords Philosophy of computer science · Ontology and epistemology of computer programs · Scientific paradigms

1 Introduction

In his seminal work on scientific revolutions, Thomas Kuhn (1992) defines scientific paradigms as "some accepted examples of actual scientific practice... [that] provide models from which spring particular coherent traditions of scientific research". The purpose of this paper is to investigate the paradigms of computer science and to expose their philosophical origins.

Peter Wegner (1976) examines three definitions of computer science: as a branch of mathematics (e.g., Knuth 1968), as an engineering ('technological') discipline, and as a natural ('empirical') science. He concludes that the practices of computer scientists are effectively committed not to one but to either one of three 'research paradigms'[1]. Taking a historical perspective, Wegner argues that each paradigm dominated a different decade during the 20th century: the scientific paradigm dominated the 1950s, the mathematical paradigm dominated the 1960s, and the technocratic paradigm dominated the 1970s—the decade in which Wegner wrote his paper[2]. We take Wegner's historical account to hold and postulate (§5) that to this day computer science is largely dominated by the tenets of the technocratic paradigm. We shall also go beyond Wegner and explore the philosophical roots of the dispute on the definition of the discipline.

Timothy Colburn (2000, p. 154) suggests that the different definitions of the discipline merely emanate from complementary interpretations (or 'views') of the activity of writing computer programs, and therefore they can be reconciled as such.

Jim Fetzer (1993) however argues that the dispute is not restricted to definitions, methods, or reconcilable views of the same activities. Rather, Fetzer contends that disagreements extend to philosophical positions concerning a broad range of issues which go beyond the traditional confines of the discipline: "The ramifications of this dispute extend beyond the boundaries of the discipline itself. The deeper question that lies beneath this controversy concerns the paradigm most appropriate to computer science". Not unlike Kuhn, Fetzer takes 'paradigm' to be that set of coherent research practices that a community of computer scientists share amongst them. By calling the disagreements 'paradigmatic' Fetzer claims that their roots

[1] To which Wegner also refers as 'cultures' or 'disciplines' interchangeably.

[2] The "Denning report" (Denning et al. 1989) authored by the task force which was commissioned to investigate "the core of computer science" also lists three "paradigms" of the discipline: theory/mathematics, abstraction/science, and design/engineering. According to this report, these paradigms are "cultural styles by which we approach our work". They conclude however that "in computing the three processes are so intricately intertwined that it is irrational to say that any one is fundamental".

extend into philosophical questions of existence (ontological) and knowledge (epistemological) about computers and programs:

> Some of the most important philosophical issues that arise within this context concern questions of a philosophical character. These involve "ontic" (or ontological) questions about the kind of things computers and programs are, as well as "epistemic" (or epistemological) questions about the kind of knowledge we can possess about things of this kind. (Fetzer 1993)

Like Fetzer and Wegner, we contend that computer scientists generally subscribe to distinct paradigms, which emanate from distinct, inconsistent, and mutually exclusive methodological positions concerning the choice of methods for investigating programs (MET, §1.1), ontic positions concerning the nature of programs (ONT, §1.2) and epistemic positions concerning the nature of knowledge about them (EPI, §1.3). In the remainder of this section we examine the philosophical disputes among computer scientists. Seeking to spell out the philosophical position underlying each paradigm of computer science, we proceed in the following sections to examine the tenets of each, contending that—

(§2) The **rationalist paradigm**, which was common among theoretical computer scientists, defines the discipline as a branch of mathematics (MET-RAT), treats programs on a par with mathematical objects (ONT-RAT), and seeks certain, a priori knowledge about their 'correctness' by means of deductive reasoning (EPI-RAT).

(§3) The **technocratic paradigm**, promulgated mainly by software engineers, defines computer science as an engineering discipline (MET-TEC), treats programs as mere data (ONT-TEC), and seeks probable, a posteriori knowledge about their reliability empirically using testing suites (EPI-TEC).

(§4) The **scientific paradigm**, prevalent in artificial intelligence, defines computer science as a natural (empirical) science (MET-SCI), takes programs to be on a par with mental processes (ONT-SCI), and seeks a priori and a posteriori knowledge about them by combining formal deduction and scientific experimentation (EPI-SCI).

Since arguments supporting the tenets of the rationalist and technocratic epistemological positions have already been examined elsewhere (e.g., Colburn's (2000) detailed account of the 'verification wars'), their treatment in §2 and §3 is brief. Instead, we expand on the arguments of *complexity, non-linearity*, and *self-modifiability* for the unpredictability of programs and conclude that knowledge concerning certain properties of all but the most trivial programs can only be established by conducting scientific experiments.

In §4 we proceed to examine seven properties of program-processes (temporal, non-physical, causal, metabolic, contingent upon a physical manifestation, and non-linear) and conclude that program-processes are, in terms of category of existence, on a par with mental processes. This discussion shall lead us to concur with Colburn and conclude that the tenets of the scientific paradigm are the most appropriate for computer science. Nonetheless, in §5 we demonstrate evidence for the dominance of the technocratic paradigm which has prevailed since Wegner (1976) described the 1970s as the decade of the 'technological paradigm' and examine its consequences. Our discussion will lead us to conclude that this domination has not benefited software engineering, and that for the discipline to become as effective as its sister, established engineering disciplines it must abandon the technocratic paradigm.

1.1 The Methodological Dispute

Computer science textbooks, classics, research articles, conferences, and curricula of undergraduate programs are dominated by radically different methods of conducting research and teaching about computer programs. Mathematical methods of investigation guide the research in computability, automata theory, computational complexity, and the semantics of programming languages; design rules of thumb, extensive testing suites, and regimented development methods dominate the branches of software engineering, design, architecture, evolution, and testing; and the methods of natural sciences, which combine mathematical theories with scientific experiments, govern the research in artificial intelligence, machine learning, evolutionary programming, artificial neural networks, artificial life, robotics, and modern formal methods. This methodological incongruity manifests itself in many ways. For example, in some research institutes computer science is a department in the school of mathematics, in others it is a part of the engineering faculty, while other computer science departments are grouped with the natural sciences.

The dispute concerning the definition of the discipline and its most appropriate methods of investigation can thus be paraphrased as follows:

> **MET** Is computer science a branch of mathematics, on a par with logic, geometry, and algebra; is it an engineering discipline, on a par with chemical or aeronautical engineering; or is it indeed a natural, experimental (empirical) science, on a par with astronomy and geology? Should computer scientists rely primarily on deductive reasoning, on test suites and regimented software development process, or should they employ scientific practices which combine theoretical analysis with empirical investigation? How is the notion of a scientific experiment different from a test suite, if at all? What is the relation between theoretical computer science and computer science?

We shall demonstrate that the methods employed by each paradigm of computer science emanate from the stance that each paradigm takes in the ontological (ONT, §1.2) and the epistemological disputes (EPI, §1.3), examined below.

1.2 The Ontological Dispute

We take the notion of a computer program to be central to computer science. In this paper we focus our discussion in the ontological dispute concerning the nature of programs.

In his discussion in questions that arise from Artificial Life ('A-Life'), Eric Olson poses the following ontological question:

> What ontological category would computer [programs] belong to? Are they supposed to be material objects? ... If so, what matter; and if not, what are they made of? ... Events or processes? Platonic complexes of pure information? ... If not, where are they? ... Are they located in space and time at all? ... Or are the traditional ontological categories of the philosophers adequate to account for this new phenomenon? (Olson 1997)

We take into consideration all sorts of entities that computer scientists conventionally take to be 'computer programs', such as numerical analysis programs, database and World Wide Web applications, operating systems, compilers/interpreters, device drivers, computer viruses, genetic algorithms, network routers, and Internet search engines. We shall thus restrict most of our discussion to such conventional notions of computer programs, and generally assume that each is encoded for and executed by silicon-based von-Neumann computers. We therefore refrain from extending our discussion to the kind of programs that DNA computing and quantum computing are concerned with.

The ontological dispute in computer science may be recast in the terminology we shall introduce below as follows:

> **ONT** Are program-scripts mathematical expressions? Are programs mathematical objects? Alternatively, should program-scripts be taken to be just 'a bunch of data' and the existence of program-processes dismissed? Or should program-scripts be taken to be on a par with DNA sequences (such as the genomic information representing a human), the interpretation of which is on a par with mental processes?

Below we clarify some of the technical terms mentioned in ONT and in the remainder of this paper.

Terminology

We seek to distinguish between two fundamentally distinct senses of the term 'program' in conventional usage: The first is that of a static script, namely a well-formed sequence of symbols in a programming language, to which we shall refer as a **program-script**. The second sense is that of a process of computation generated by 'executing' a particular program-script, to which we shall refer as a **program-process**. Any mention of the term 'program' shall henceforth apply to both senses[3].

Rather than attempting to define these terms formally we shall illustrate them with an example. Each *program-script* is associated with a *programming language*. What distinguishes a program-script from a mere sequence of symbols is the requirement that program-scripts are expressions that are *well formed* according to the syntactical and semantic rules of a specific programming language.

The programming languages we are concerned are generally divided into *machine* and *high order* programming languages. By *machine programming language* we shall refer to programming languages which restrict themselves to

[3] For example, the statement 'programs are abstract' shall be taken to assert that 'program-scripts and program-processes are abstract'.

Table 1 Program-script encoded in a machine programming language[5]

```
75 E0 73 75 E1 73 FA D3 75 52 D5 75 74 F9 A2 21 F0 73
58 71 F9 A2 21 F0 73 30 73 E4 73 31 73 E5 73 32 73 E6
73 33 73 E7 44 70 34 F6 43 73 E6 43 73 E7 44 70 35 F6
73 34 73 E0 73 35 73 E1 75 60 5E D5 75 31 D3 75 30 F6
```

primitive machine instructions for a given von-Neumann, silicon-based, mass produced class of microprocessors. For example, the Intel 8086 class of microprocessors effectively defines a specific machine programming language, such that a program-script encoded therein[4] is decipherable by any computer based on the 808X microprocessor family. In such machines, each program-script is represented as a configuration of electrical charges of the machine's memory, normally transcribed in binary or hexadecimal code, as demonstrated in Table 1.

Not surprisingly, programs in machine programming languages proved to be exceptionally difficult for humans to understand, reason about, and adapt.[6] Worse still, rapid developments in computing and microprocessor technology make programs encoded for older generation microprocessors obsolete along with the class of machines for which they were specifically tailored.

Improvements in the processing power of computers during the 1950s have enabled the introduction of *high-order programming languages* (IEEE 1990), also *compiled* or *interpreted* programming languages. High-order programming languages allow programmers to harness the power of other powerful programs, such as compilers (interpreters) and operating systems, which interpret and execute program-scripts encoded in these languages. As a result from their prevalence, claims about the 'text of the program' (e.g., Hoare's §2) most commonly refer to programs encoded in high-order programming languages, such as the program-script depicted in Table 2.

The second sense of the word 'program' is that of a process (also *thread, task*, or *bot*). The term *program-process* is thus reserved to that entity which is generated from executing a program-script in the appropriate operational environment. Once generated, a copy of the program-script is loaded into the computer's memory (the program's 'image'), followed by executing of the first instruction copied to the image. For example, executing the program-script in Table 2 with the numbers 4 and 5 as input using Intel Pentium IV PC equipped with the Linux operating system and the ComLisp compiler (Georick et al. 1997) shall generate to a program-process which calculates the value of the expression 3+(4×5), the proceedings of which are depicted in Table 3.

Confusion concerning the notion of a computer program can often be traced to the two senses of the term. But program-scripts must not be confused with program-

4 Also known as *machine code* or *object code*.

5 The program adds 3 to the product of two numbers, encoded in the 8086 microprocessor assembly language (Adapted from Georick et al. 1997).

6 For example, consider the difficulty of spotting and correcting errors in the program in Table 1.

Table 2 Program-script encoded in Lisp[7]

```
(define example
    (lambda (x y)
        (+ (*  y) 3)))
```

Table 3 Steps in a sample program-process generated from executing the program in Table 2

```
> (example 4 5)
(+ (* 4 5) 3)
(+ 20 3)
23
```

processes: The first is an inert sequence of symbols; the second is a causal and a temporal entity. Any number of program-processes can be potentially be generated from each program-script. Furthermore, certain operating systems allow the simultaneous generation of a large number of program-processes from a single program-script executed concurrently by a single microprocessor. For example, my Personal Computer can generate and concurrently execute large numbers of program-processes from the program-script in Table 2.

1.3 The Epistemological Dispute

Program specifications are statements that assert our expectations from a program. If specifications are defined before the program-script is encoded they can be used to articulate the objectives of the encoding enterprise and drive the software development process, which is often complex and arduous. For example, a specification asserting that the program-script in Table 2 indeed calculates the sum of the product of two numbers and the number 3 can be formally specified as a lambda expression:

$$\lambda xy.x \times y + 3 \tag{1}$$

In more conventional notation, (1) can also be represented as a two-place function:

$$\text{example}(x, y) = x \times y + 3 \tag{2}$$

Having formally articulated the specification, the *correctness* of a program can be taken to mean the extent to which it meets its specifications. The question of correctness can thus be recast as the question whether any or all of the program-processes that can, shall, and have been generated from program-script ps_s meet or 'satisfy' specification s. The hypothesis 'program ps_s is correct with relation to s' therefore asserts that ps_s satisfies s. For example, the correctness of `example`

[7] The program adds 3 to the product of two numbers, encoded here in the syntax of Scheme (Abelson and Sussman 1996), a dialect of Lisp

Table 4 Sample informal specifications

• Program *x* does not cause the space shuttle to explode
• Program *x* translates French into English
• Program *x* is a computer virus
• Program *x* never lets unauthorized persons to access sensitive data
• Program *x* never terminates unexpectedly
• Program *x* takes a regular expression (a string of text) and returns a list of World Wide Web documents sorted by their 'relevance' to this expression
• Program *x* detects whether the face of person *y* appear in any given picture
• Program *x* executes with visibly identical outcome regardless of the operating system used

(Table 2) can be defined by the extent to which it satisfies specification (2). If the specification is articulated in a mathematical language, as in (2), it is referred to as a *formal specification*, in which case the question of 'correctness' is well-defined.

Most specifications however are not quite as simple as (2). Specifications may assert not only the outcome of executing a particular program-script (e.g., adding a record to a database of moving a robotic arm) but also how efficient are the program-processes generated therefrom (e.g., how long it takes to carry out a particular calculation) and how reliable they are (e.g., do they terminate unexpectedly?). For this reason, fully formulated specifications are not always feasible, as demonstrated by the specifications in Table 4.

Indeed, although the correctness of a program can be a source of considerable damage, or even a matter of life and death, it may be very difficult—or, as Fetzer and Cohn claimed, altogether impossible—to establish formally. And while executing a program-script in various circumstances ('program testing') can discover certain errors, no number of tests can establish their absence[8]. For these reasons, the **problem of program correctness** has become central to computer science. If correctness cannot be formally specified and the problem of establishing it is not even well-defined then is it at all meaningful to ask whether a program is correct, and if so then what should 'correctness' be taken to mean and how can it be established effectively? These questions are at the heart of the epistemological dispute:

EPI Is warranted knowledge about programs a priori or a posteriori?[9] In other words, does knowledge about programs emanate from empirical evidence or from pure reason? What does it mean for a program to be correct, and how can this property be effectively established? Must we consider correctness to be a well-defined property—should we insist on formal specifications under all circumstances and seek to prove it deductively—or should we adopt a probabilistic notion of correctness ('probably correct') and seek to establish it a posteriori by statistical means?

[8] A statement most widely attributed to Dijkstra.

[9] We follow Colburn (2000) in taking a priori knowledge about a program to be knowledge that is prior to experience with it, namely knowledge emanating from analyzing the program-script, and a posteriori knowledge to be knowledge following from experience with observed phenomena, namely knowledge concerning a given set of specific program-processes generated from a given script.

Of all the philosophical questions we shall examine, computer scientists have been most explicit in the position they take concerning the epistemological dispute.

2 The Rationalist Paradigm

By the rationalist paradigm we refer to that paradigm of computer science which takes the discipline to be a branch of mathematics, the tenets of which have been common among scientists investigating various branches of theoretical computer science, such as computability and the semantics of programming languages. Tony Hoare summarized the tenets of the rationalist paradigm as follows:

> (1) Computers are mathematical machines. Every aspect of their behaviour 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 behaviour, intended or unintended, of the computer on which they are executed. ... (4) Programming is a mathematical activity... its successful practice requires determined and meticulous application of traditional methods of mathematical understanding, calculation and proof[10].

2.1 The Rationalist Methods

Concerned primarily with what is today taken to be the foundations of the discipline, theoretical computer science is the oldest and the most rigorously established branch of computer science. In the first decades following the work of Turing (1936; Turing and Copeland 2004), who is widely considered to be the father of the discipline, computer science has largely been identified with what von-Neumann described as the mathematical investigation of “the extent and limitations of mechanistic explanation”. During the 1930s, influential mathematicians such as Turing, Church, and Kleene developed mathematically potent theories which sought (and succeeded) to lend precision to intuitive notions of *mechanistic computation* (also *effective computation*). One of the earliest triumphs of theoretical computer science has been the mathematical proof according to which all the different mathematical notions of mechanistic computation on offer—turing machines, lambda expressions, and recursive functions[11]—are computationally equivalent. This important result lent considerable support to what came to be known as Turing’s Thesis (also Church–Turing Thesis, Copeland 2002), according to which *any* ‘mechanistic’ process of computation can indeed be represented as the process of computation by a turing machine, and by extension, as an algorithm, a recursive function, etc.

[10] Delivered, according to Mahoney (2002), in 1985 during his Inaugural Lecture as Professor of Computation at Oxford.

[11] Which were later accompanied by algorithms and abstract state machines.

During the 1940s the first electronic computers appeared, and with them emerged the contemporary notions of *computer programs* (§1.2). A mathematical proof demonstrating that programs encoded in machine programming languages are computationally equivalent to the mathematical notions of mechanistic computation on offer has established the relevance of deductive reasoning to modern computer science. In particular, computational equivalence implied that any problem which can be solved (or *efficiently solved*) by a turing machine can be solved by executing a program-script encoded in a *machine programming language* (§1.2), and vice versa, namely, that any problem which cannot be (efficiently) solved by a turing machine also cannot be (effectively) solved by executing a program-script encoded in a machine programming language. For this reason machine programming languages are described as 'turing-complete' languages. High-order programming languages have thus appeared in a rich mathematical context, the design of which was heavily influenced by the mathematical notions of mechanistic computation on offer. For example, the striking resemblance between the Lisp program in Table 2 and the lambda expression specifying it (1) emanates directly from the commitment of the designer of the Lisp programming language (McCarthy 1960) to lambda calculus.

The fundamental theorems of the theories of computation have remained relevant notwithstanding generations of exponential growth in computing power. Time has thus secured the primacy of deductive methods of investigation as a source of certain knowledge about programs and led many to concur with Hoare. For example, Knuth justifies his definition of computer science as a branch of mathematics (Knuth 1968) as follows:

> Like mathematics, computer science will be somewhat different from other sciences in that it deals with man-made laws which can be [deductively] proved, instead of natural laws which are never known with certainty. (Knuth 1974)

The rationalist stance in the methodological dispute can thus be summarized as follows:

> **MET-RAT** Computer science is a branch of mathematics, writing programs is a mathematical activity, and deductive reasoning is the only accepted method of the investigating programs.

MET-RAT is justified by the rationalist ontological and epistemological positions examined below.

2.2 The Rationalist Ontology

Discovering the Turing-completeness of programming languages has established that every program-process can be adequately represented by some turing machine, and by extension, by an algorithm, a recursive function, and by any other computationally equivalent mathematical notion of mechanistic computation. The powerful insights that these mathematical notions of programs offer have led Hoare (1986), Dijkstra (1988), and Lamport (1977) to claim that program-scripts are mathematical expressions.[12] This premise motivates the rationalist position in the ontological dispute (ONT), which can be recast and justified as follows:

ONT-RAT Program-scripts are mathematical expressions. Mathematical expressions represent mathematical objects. A program p is that which is *fully and precisely* represented by s_p. Therefore p is a mathematical object.

Functional and logic programming languages lend considerable support to ONT-RAT. Indeed, the striking similarity between the Lisp program in Table 2 and the recursive function defined in expression (1) can be taken to demonstrate that what is (*fully* and *precisely*) represented by a lisp program *is* indeed an recursive function, a position which can be argued as follows:

ONT-RAT$_{\textbf{function}}$ A program-script s_p encoded in any turing-complete programming language (e.g., Lisp) is a mathematical expression representing a recursive function f_p. A program p is that which is *fully* and *precisely* represented by s_p. Therefore p *is* the mathematical function f_p.

A possible objection to the rationalist ontology stems from the proliferation of the kinds of mathematical objects on offer. Why, indeed, should programs be taken to be recursive functions or turing machines rather than algorithm or any other (computationally equivalent) class of mathematical objects? But the proliferation of mathematical explanations can be taken to corroborate rather than weaken ONT-RAT. Indeed, computational equivalence precisely means that any deduction that can be made from choosing one mathematical explanation of mechanistic computation can be made from the other. Stronger objections to ONT-RAT are examined in §4.3.

ONT-RAT raises metaphysical questions concerning the nature of mathematical objects. Prima facie, ONT-RAT may be taken to commit the rationalist to a platonist position (e.g., Balaguer 2004). Plato's sphere of perfect existence consists of ideal *universals* (or *Forms*), such as mathematical objects, which are abstract (intangible, non-physical), eternal, observer-independent, non-mental, and immutable entities, that can only be perceived thoughour intellects (which Plato takes to be a yet another sensory organ). Universals are taken to exist regardless whether humans are aware of their existence are unaffected by the creation or destruction of any number of particulars. A platonist justification to ONT-RAT$_{\text{function}}$ can be recast in these terms as follows:

ONT-RAT$_{\textbf{platonism}}$ A program-script s_p is a mathematical expression. A program p is that which is fully and precisely represented by s_p. Hence, p is a mathematical object. Mathematical objects are platonic universals. Therefore, programs are universals.

[12] Dijkstra (1988) offered an explanationto how this 'fact' escaped mathematicians and programmers alike: "Programs were so much longer formulae than [mathematics] was used to that [many] did not even recognize them as such.

ONT-RAT$_{\text{platonism}}$ has some interesting consequences. It implies that the lambda calculus, abstract automata, as well as every build and every version of Windows XP (and of every operating system) were *discovered* rather than *invented* (Turner 2007). It also implies that every program that has ever been written, will ever be written, or can be written, exists eternally in the sphere of perfect existence, regardless whether it is 'discovered', encoded, or executed.[13]

However, most theoretical computer scientists have refrained explicitly committing computer programs to any particular category of existence. Despite its appeal, we found no reference to ONT-RAT$_{\text{platonism}}$ or to any particular branch of metaphysics. Indeed, ONT-RAT is also in line with other positions in metaphysics, such as conventionalism and intuitionism. The objections to ONT-RAT we examine in §4.3 shall therefore focus on the inadequacy of mathematical objects as an account for the apparent properties of programs.

2.3 The Rationalist Epistemology

Hoare (1986) is explicit in his commitment to the primacy of a priori, certain knowledge about programs and to the role of mathematical deduction in establishing it. This position was shared by those who like Hoare sought to establish mathematically the (formal) semantics of programming languages, most notably Dana Scott and Christopher Strachey (1973). Hoare (1969) himself offers an axiomatic theory formulated in the classical mathematical logic. By Hoare' Logic, each stage in the computation process is represented by a state that can be captured by a set of axioms in mathematical logic $\{P\}$. The consequences of executing a particular statement s is represented as that state $\{Q\}$ which results from applying that rule of inference s' which is associated with the statement s to $\{P\}$. The Hoare triple $\{P\}s\{Q\}$ can therefore be taken to represent the semantics of statement s. For example, the *intended* behaviour of program example (Table 2) can be represented by the following Hoare triple:

$$\{x, y \in \mathbb{N}\}\ (\texttt{example x y})\ \{output = x \times y + 3\} \tag{3}$$

The proof of correctness of the script in Table 2 shall proceed with the attempt to prove (3) by employing the rules of inference of Hoare Logic. Once established, such a mathematical proof shall thus secure the correctness of the program-script in Table 2 with certainty otherwise reserved to mathematical theorems.

Other efforts in delivering formal semantics have followed Hoare's example in the attempt to prove program correctness using other axiomatic theories. In particular, Scott's *denotational semantics* (Stoy 1977) harnessed the axioms of Zermelo-Fraenkel to prove program correctness.

[13] Bill Rapaport (2007) notes that such a position has interesting consequences on the question whether programs can be copyrighted or patented.

The mathematical investigation of semantics of programming languages has been at least partially successful. If certain simplifying assumptions on the programming language are taken then some of the properties of the program-script and some of the consequences of executing it can indeed, at least in principle, be formally deduced. However, such a notion of program correctness required not only that *specifications* (§1.3) are *fully* and *formally* defined—a potentially unfeasible task (Table 4), and that programs-scripts can be *fully* and *precisely* represented in the same formal language, but also that these mathematical expressions lend themselves to the deductive process of formal verification. In 1962 John McCarthy even suggested that, not only that program correctness can be deductively proven, but also that it should be possible to mechanize the process of checking such proofs:

> It should be possible to eliminate debugging. ... Instead of debugging a program one should prove that it meets its specifications, and this proof should be checked by a computer program. (McCarthy 1962)

Indeed for the rationalist correctness is a well-defined, *a priori* notion which must be proven mathematically. Hoare dismisses whatever pragmatic arguments against this epistemological position and claims that *a posteriori* knowledge emanating from experience (e.g. 'debugging') must be dismissed as ineffective, anecdotal and unscientific:

> I find digital computers of the present day to be very complicated and rather poorly defined. As a result, it is usually impractical to reason logically about their behaviour. Sometimes, the only way of finding out what they will do is by experiment. Such experiments are certainly not mathematics. Unfortunately, they are not even science, because it is impossible to generalize from their results or to publish them for the benefit of other scientists. (Hoare, in Fetzer (1993))

The rationalist epistemological position can thus be recast as follows:

> **EPI-RAT** Programs can be fully and formally specified, and their 'correctness' is a well-defined problem. Certain, a priori knowledge about programs emanates from pure reason, proceeding from self-evident axioms to the demonstration of theorems by means of formal deduction. *A posteriori* knowledge is to be dismissed as anecdotal and unreliable.

EPI-RAT is in line with rationalism in traditional epistemology (Markie 2004) which holds that pure reason alone, as opposed to sense experience, play a role in our attempt to gain knowledge, and that a priori knowledge is superior to a posteriori knowledge. This motivated our choice to refer to the rationalist paradigm of computer science as such.

EPI-RAT is intimately tied with the rationalist's ontological commitment to mathematical objects (ONT-RAT). While empirical evidence can give us some intuition about the nature of mathematical objects such as numbers, triangles, and (set-theoretic), e.g., by adding up apples or by drawing triangles on paper, such evidence only offer anecdotal knowledge. If programs are taken to be mathematical objects (ONT-RAT) and the methods of computer science are the methods of mathematical disciplines, then knowledge about programs can only proceed deductively. Indeed, a rationalist position towards knowledge in branches of pure mathematics such as geometry, logic, arithmetic, topology, and set theory largely dismiss a posteriori knowledge as unreliable, ineffective, and not sufficiently general.

Objections to EPI-RAT are examined in the following sections.

3 The Technocratic Paradigm

By the 'technocratic paradigm'[14] we refer to that paradigm of computer science which defines the discipline as a branch of engineering, proponents of which dominate the various branches of software engineering, including software design, software architecture, software maintenance and evolution, and software testing. In line with the empiricist position in traditional philosophy, the technocratic paradigm holds that reliable, a posteriori knowledge about programs emanates only from experience, whereas certain, a priori 'knowledge' emanating from the deductive methods of theoretical computer science is either impractical or impossible in principle.

3.1 The Technocratic Methods

Wegner describes the background to the emergence of the technocratic paradigm, echoing what we shall refer to as the *argument of complexity* (§3.2):

> During the 1970s emphasis shifted away from "pure research" towards practical management of the environment, not only in computer science but also in other scientific areas. Decreasing hardware costs and increasingly complex software projects created a "complexity barrier" in software development which caused the management of software-hardware complexity to become the primary practical problem in computer science. Research was directed away from the development of powerful new programming languages and general theories of programming language structure towards the development of tools and methodologies for controlling the complexity, cost and reliability of large programs (Wegner 1976)[15].

[14] **tech · noc · ra · cy** *n.* A government or social system controlled by technicians, especially scientists and technical experts. (The American Heritage® Dictionary of the English Language: Fourth Ed., 2000.)

[15] These events have led to the seminal NATO conference held in the fall of 1968 (Naur and Randell 1969) concerning the trouble that the software industry had been experiencing in producing reliable computing systems. In the introduction to the conference's report, Robert McClure (2001) argues that although the term 'software engineering' was not in general use at that time, its adoption for the titles of these conferences was deliberately provocative and played a major role in gaining general acceptance for the term.

The technocratic turn away from the methods of theoretical computer science, indeed away from all scientific practices, was most explicitly articulated by John Pierce:

> I don't really understand the title, Computer Science. I guess I don't understand science very well; I'm an engineer. ... Computers are worth thinking about and talking about and doing about only because they are useful devices, which do something for somebody. If you are just interested in contemplating the abstract, I would strongly recommend the belly button. (Pierce 1968)

Indeed the technocratic doctrine contends that there is no room for theory nor for science in computer science. During the 1970 this position, promoted primarily by software engineers and programming practitioners, came to dominate the various branches of software engineering. Today, the principles of scientific experimentation are rarely employed in software engineering research. An analysis of all 5,453 papers published during 1993–2002 in nine major software engineering journals and proceedings of three leading conferences revealed that less than 2% of the papers (!) report the results of controlled experiments. Even when conducted, the statistical power of such experiments falls substantially below accepted norms as well as the levels found in the related disciplines (Dybå et al. 2006).

Instead of conducting experiments, software engineers use testing suites, the purpose of which is to establish statistically the reliability of specific products of the process of manufacturing software. For example, to establish the reliability of a program designed for operating a microwave oven, software engineering educators speak of a regimented process of software design (although a precise specification of which is hardly ever offered), followed by an 'implementation' phase during which the program-script is encoded (about which little can be said), concluding with the construction of a testing suite and executing (say) 10,000 program-processes generated from the given program-script. If executed in a range of actual (rather than hypothetical) microwave ovens, such a comprehensive test suite furnishes the programmer with statistical data which can be used to quantitatively establish the reliability of the computing system in question, e.g., using metrics such as *probability of failure on demand* and *mean time to failure* (Sommerville 2006).

Evidence to the decline of scientific methods is found in textbooks on software engineering (e.g., Sommerville 2006). Rarely dedicating any space to deductive reasoning[16] and never to the principles of scientific experimentation in empirical sciences, such textbooks cover the subjects of software design, software evolution, and software testing, focusing on manufacturing and testing methods borrowed from traditional engineering trades. Much discussed topics include models of software development lifecycles, methods of designing testing suites, reliability metrics, and statistical modelling.

The position of the technocratic paradigm concerning the methodological dispute can thus be recast as follows:

[16] At most, lip-service is paid to the role of verification in 'safety-critical software systems'.

MET-TEC Computer science is a branch of engineering which is concerned primarily with manufacturing *reliable* computing systems, a quality determined by methods of established engineering such as reliability testing and obtained by means of a regimented development and testing process. For all practical purposes, the methods of theoretical computer science are dismissed as 'naval gazing'.

The technocratic methods of investigation are primarily motivated by the technocratic epistemological position.

3.2 The Technocratic Epistemology

So far, there has been little philosophical discussion of making software reliable rather than verifiable. ... If another view of software could arise ..., the interests of real-life programming and theoretical computer science might both be better served. (DeMillo et al. 1979)

The technocratic rejection of the premises of the rationalist epistemology (EPI-RAT) rely on the **argument of complexity** for the inadequacy of deductive reasoning, articulated by Richard DeMillo, Richard Lipton, and Alan Perlis as follows:

Back in the real world ... the specifications for any reasonable compiler or operating system fill volumes—and no one believes that they are complete. ... The input assertions for these algorithms are not even formulable, let alone formalizable. (DeMillo et al. 1979)

Indeed, whether a particular program-process meets our expectations depends on the idiosyncrasies of the compiler, the operating system, and the particular computer executing it, which are determined by the commercial concerns of their respective vendors. These factors place specifications such as those listed in Table 4, as well as the programs implementing them, at a level of complexity which does *not* lend itself to formal deduction. By this argument, the inevitable conclusion is that formal deduction is ineffective establishing the correctness of all but the most trivial computer programs

The *argument of complexity* receives further corroboration from the technological progress during the past three decades since it was first articulated. Since 1979, the average size of programs and operating systems grew in at least four orders of magnitude. More importantly, the complexity of compilers, operating systems, microprocessors, and input is today compounded by component-based software engineering technologies (Szyperski 2002), such as JavaBeans, .NET, and CORBA. These technologies gave rise to gigantic programs such as Internet search engines and electronic commerce applications which consist of hundreds of software *components* (e.g., dynamically linked libraries, server-side and client-side threads), whose construction is often 'outsourced' or otherwise delegated to a range of independent commercial bodies or individual volunteers[17] and which execute on any one of a wide range of microprocessors (i.e., in a 'heterogeneous environment'). The notion of 'input' with regard to these programs has also been much further complicated as signals and data arrive to these programs from innumerable other interacting programs, many of which can be as complex as autonomous software agents (Fasli 2007), and which communicate via vast and very complex communication networks. Any form of deductive reasoning about such programs requires the representation of petabytes[18] of instructions and data in every one of the components of the program and of every computer, operating system, and network router that is involved (directly or indirectly) in their execution. Since these often change during the lifespan of a program-process, the very notion of a program-script is therefore not well-defined, specifications are not well-defined, and deductive reasoning about their de facto representations is an idealization that is as unrealistic and ineffective as, say, deductive reasoning about the individual atomic particles of airplanes and power stations.

From the analogy to airplanes and power stations, DeMillo et al. conclude that only probabilistic methods such as those employed by statistic mechanics and thermodynamics can effectively establish any knowledge about such gargantuan engineering feats:

> How then do engineers manage to create reliable structures? ... They have a mature and realistic view of what "reliable" means; in particular, the one thing it never means is "perfect". There is no way to deduce logically that bridges stand, or that airplanes fly, or that power stations deliver electricity. (DeMillo et al. 1979)

According to DeMillo et al. the argument of complexity is so compelling that any resistance thereto amount to 'symbol chauvinism':

> It is nothing but symbol chauvinism that makes computer scientists think that our structures are so much more important than material structures that (a) they should be perfect, and (b) the energy necessary to make them perfect should be expended. We argue rather that (a) they cannot be perfect, and (b) energy should not be wasted in the futile attempt to make them perfect. (DeMillo et al. 1979)

Rather than taking 'correctness' to be a certain, formally defined property, computer scientists must learn from the established branches of engineering that more realistic notions of correctness are in place, meaning probabilistic notions of reliability:

> It is no accident that the probabilistic view of mathematical truth is closely allied to the engineering notion of reliability. Perhaps we should make a sharp

[17] For example, the Debian GNU/Linux 3.1 version of the Linux operating system (Debian 2007) is the product of contributions made by thousands of individuals that are entirely unrelated except in their attempt to improve it.

[18] One petabyte (1PB) is 1,024 terabytes or 2^{50} bytes.

distinction between program reliability and program perfection—and concentrate our efforts on reliability. (DeMillo et al. 1979)

The technocratic position concerning the epistemological dispute may be recast in terms of the argument of complexity as follows:

EPI-TEC It is *impractical* to specify formally or to prove deductively the 'correctness' of a complete program. A priori, certain knowledge about the behaviour of actual programs is therefore unattainable. If at all meaningful, 'correctness' must be taken to mean tested and proven 'reliability', a posteriori knowledge about which is measured in probabilistic terms and established using extensive testing suites.

Fetzer (1993) and Avra Cohn (1989) offer what is essentially an ontological argument for an even stronger epistemological position, to which we shall refer as the **argument of category mistake**. According to this argument, a priori knowledge about the behaviour of machines is impossible in principle:

A proof that one specification implements another—despite being completely rigorous, expressed in an explicit and well understood logic, and even checked by another system—should still be viewed in context of many extra-logical factors which affect the correct functioning of hardware systems. (Cohn 1989)

The technocratic position concerning the nature of knowledge can be justified by the argument of category mistake as follows:

EPI-TEC$_{\mathbf{Ont}}$ It is *impossible* to prove deductively the correctness of any physical object. A priori, certain knowledge about the behaviour of actual programs is unachievable. If at all meaningful, 'correctness' must be taken to mean tested and proven 'reliability', a posteriori knowledge about which is measured in probabilistic terms and established using extensive testing suites.

Peter Markie (2004) defines empiricism as that school of thought which holds that sense experience is the ultimate source of all our concepts and knowledge. Empiricism rejects pure reason as a source of knowledge, indeed any notion of a priori, certain knowledge, claiming that warranted beliefs are gained from experience. Thus, EPI-TEC and EPI-TEC$_{\mathrm{Ont}}$ are in line with the empiricist philosophical position.

The *argument of complexity* won the hearts of many computer scientists. As a result, the technocratic doctrine has come to dominate software engineering journals (IEEE TSE) and conferences (ICSE), contributions to which are traditionally judged by experience gained from actual implementations—"concrete, practical applications"—which must be employed to demonstrate any thesis put forth, may it be theoretical or practical. Software engineering classics such as the 1969 NATO report (Naur and Randell 1969) and the grand "Software Engineering Body of Knowledge" project (Abran and Moore 2004) hold a posteriori knowledge to be superior on all other knowledge about programs and dismiss or neglect the role of formal deduction. Same position is widely embraced in all branches of software design. For example, the merits of *design patterns* (Gamma et al. 1995) and *architectural styles* (Perry and Wolf 1992) are measured almost exclusively in terms of the number of successful applications thereof.

3.3 The Technocratic Ontology

The records of the NATO conference on software engineering (Naur and Randell 1969) quote van der Pohl in suggesting that program-scripts are themselves just "bunches of data":

> A program [script] is a piece of information only when it is executed. Before it's really executed as a program in the machine it is handled, carried to the machine in the form of a stack of punch cards, or it is transcribed, whatever is the case, and in all these stages, it is handled not as a program but just as a bunch of data. (Van der Poel, in Naur and Randell 1969)

If mere "bunches of data", representing a configuration of the electronic charge of a particular printed circuit, program-scripts are on a par with (the manuscript of) Shakespeare's *Hamlet* and (the pixelized representation of) Botticelli's *The Birth of Venus*. Therefore 'that which can be represented by data' can be just about anything, including non-existent entities such as Hamlet and Venus. The existence of those putative abstract (intangible, non-physical) entities must therefore be rejected.

This objection can be attributed to a nominalist position in traditional metaphysics. Nominalism (Loux 1998) seeks to show that discourse about abstract entities is analysable in terms of discourse about familiar concrete particulars. Motivated by an underlying concern for ontological parsimony, and in particular the proliferation of universals in the platonist's putative sphere of abstract existence, the nominalist principle commonly referred to as Occam's Razor ("don't multiply entities beyond necessity") denies the existence of abstract entities. By this ontological principle, nothing exists outside of concrete particulars, including not entities that are 'that which is fully and precisely defined by the program script' (ONT-RAT). The existence of a program is therefore unnecessary.

The technocratic ontology can thus be summarized as follows:

> **ONT-TEC** 'That which is fully and precisely represented by a script s_p' is a putative abstract (intangible, non-physical) entity whose existence is not supported by direct sensory evidence. The existence of such entities must be rejected. Therefore, 'programs' do not exist.

Indeed, the recurring analogies to airplanes, power stations, chemical analyzers, and other engineered artefacts for which no ontologically independent notion of a program is meaningful seems to support ONT-TEC. But while ONT-TEC is corroborated by a nominalist position, it is not committed thereto. In absence of an explicit commitment to any particular school of thought in metaphysics, it is impossible to determine whether ONT-TEC is indeed motivated by nominalism.

4 The Scientific Paradigm

The scientific paradigm contends that computer science is a branch of natural (empirical) sciences, on a par with “astronomy, economics, and geology” (Newell and Simon 1976), the tenets of which are prevalent in various branches of AI, evolutionary programming, artificial neural networks, artificial life (Bedau 2004), robotics (Nemzow 2006), and modern formal methods (Hall 1990). Since many programs are unpredictable, or even ‘chaotic’, the scientific paradigm holds that a priori knowledge emanating from deductive reasoning must be supplanted with a posteriori knowledge emanating from the empirical evidence by conducting scientific experiments. Since program-processes are temporal, non-physical, causal, metabolic, contingent upon a physical manifestation, and nonlinear entities, the scientific paradigm holds them to be on a par with mental processes.

4.1 The Scientific Methods

Allen Newel and Herbert Simon, prominent pioneers of AI, define computer science as follows:

> Computer science is the study of the phenomena surrounding computers ... it an empirical discipline ... an experimental science ... like astronomy, economics, and geology (Newell & Simon 1976)

Scientific experiments are traditionally concerned with ‘natural’ objects, such as chemical compounds, DNA sequences, stellar bodies (e.g., Eddington’s 1919 solar eclipse experiment), atomic particles, or human subjects (e.g., experiments concerning cognitive phenomena.) It can be argued that the notion of scientific experiment is only meaningful when applied to ‘natural’ entities but not to ‘artificial’ objects such as programs and computers; namely, that programs and computers cannot be the subject of scientific experiments:

> There is nothing natural about software or any science of software. Programs exist only because we write them, we write them only because we have built computers on which to run them, and the programs we write ultimately reflect the structures of those computers. Computers are artifacts, programs are artifacts, and models of the world created by programs are artifacts. Hence, any science about any of these must be a science of a world of our own making rather than of a world presented to us by nature. (Mahoney 2002)

As a reply, Newell and Simon contend that, even if they are indeed contingent artefacts, programs are nonetheless appropriate subjects for scientific experiments, albeit of a novel sort (“nonetheless, they are experiments”. Newell and Simon 1976) Their justification for this position is simple: If programs and computers are taken to be some part of reality, in particular if the scientific ontology (ONT-SCI) is accepted, then we see no particular difficulty in employing scientific methods for investigating them. Even Turing acknowledged the role of experiments in investigating the behaviour of artificial artefacts:

> We also wish to allow the possibility than an engineer or team of engineers may construct a machine which works, but whose manner of operation cannot be satisfactorily described by its constructors because they have applied a method which is largely experimental. (Turing 1950)

Additional arguments supporting the relevance of scientific experimentation concern the limits of analytical methods. In §4.2, we shall examine the *argument of non-linearity* and the *argument of self-modifiability* and conclude that, indeed, knowledge about even some of the simplest programs can only be gained via experiments.

The scientific notion of experiment must be clearly distinguished from the technocratic notion of a *reliability test* (§3.1). The purpose of a reliability test is to establish the extent to which a program meets the needs of its users, whereas a scientific experiment is designed to corroborate (or refute) a particular hypothesis. If a test suite fails, the subject of experiment (the program) must be revised (or discarded); if an experiment 'fails', the theory must be revised (or discarded), or else the integrity of the experiment is in doubt. For example, an appropriate test suite may have prevented that programming error (in the conversion a 64-bit floating-point number to a 16-bit signed integer) which caused the Ariane 5 Flight 501 to disintegrate forty seconds after launch. The purpose of such a test suite is to prevent the space shuttle from exploding; had a test suite discovered this error, the program would have been revised. In contrast, Eddington's experiment of measuring the bending of light at a total solar eclipse in 1919 was specifically tailored to test Einstein's 1915 general theory of relativity. Had this experiment failed to corroborate this theory, General Relativity or the integrity of Eddington's experiment would have been questioned.

For this reason, experiments with programs go beyond establishing the usability of a particular manufactured artefact, even beyond the 'extent and limitations of mechanistic explanation'. Computer programs can also be used as tools in discovering and empirically establishing the laws of nature. In particular, program simulations can be used to examine the veracity of models of non-linear phenomena (such as the ones we shall examine in §4.2) in other natural sciences. For example, in cognitive psychology, an artificial intelligence programs can be taken to be a tool for empirical examinations of models of memory and learning; in bioinformatics, genetic algorithms are used to test the extent to which models of the reproduction of DNA molecules are corroborated by the laws of Darwinian natural selection; and in astronomy, the predictions of models for the creation of the universe can be tested by means of computer simulations. If computer science is concerned with the 'phenomena surrounding computers'—such as the behaviour of computer simulations—then its subject matter is distinct from any given class of natural phenomena at most in the extent to which scientific theories deviate from reality. In other words, our programs are only 'incorrect' to the extent to which the scientific theories they implement deviate from the phenomena they seek to explain. In Popper's (1963) terms, the difference between programs and the (naturalistic view of) reality is at most limited by the verisimilitude (or truthfulness) of our most advanced scientific theory. The progress of science is manifest in the increase in this verisimilitude. Since any distinction between the subject matter of computer science and natural sciences is taken to be at most the product of the (diminishing) inaccuracy of scientific theories, the methods of computer science are the methods of natural sciences.

But the methods of the scientific paradigm are not limited to empirical validation, as mandated by the technocratic paradigm. Notwithstanding the technocratic arguments to the unpredictability of programs (as well as the additional arguments we examine in §4.2), the deductive methods of theoretical computer science have been effective in modelling, theorizing, reasoning about, constructing, and even in predicting—albeit only to a limited extent—innumerable *actual* programs in countless many practical domains. For example, context-free languages has been successfully used to build compilers (Aho et al. 1986); computable notions of formal specifications (Turner 2005) offer deductive methods of reasoning on program-scripts without requiring the complete representation of petabytes of program and data; and classical logic can be used to distinguish effectively between abstraction classes in software design statements (Eden et al. 2006). If computer science is indeed a branch of natural sciences then its methods must also include deductive and analytical methods of investigation.

From this Wegner (1976) concludes that theoretical computer science stands to computer science as theoretical physics stands to physical sciences: deductive analysis therefore plays the same role in computer science as it plays in other branches of natural sciences. Analytical investigation is used to formulate hypotheses concerning the properties of specific programs, and if this proves to be a highly complex task (e.g., Table 4) it nonetheless an indispensable step in any scientific line of enquiry.

Tim Colburn concurs with this view and concludes that in reality the tenets of the scientific paradigm offer the most complete description of the methods of computer science:

> Computer science "in the large" can be viewed as an experimental discipline that holds plenty of room for mathematical methods, including formal verification, within theoretical limits of the sort emphasized by Fetzer (Colburn 2000, p. 154)

To summarize, the scientific position concerning the methodological question (MET) can therefore be distinguished from the rationalist (MET-RAT) and the technocratic (MET-TEC) positions as follows:

MET-SCI Computer science is a natural science on a par with astronomy, geology, and economics, any distinction between their respective subject matters is no greater than the limitations of scientific theories. Seeking to explain, model, understand, and predict the behaviour of computer programs, the methods of computer science include both deduction and empirical validation. Theoretical computer science therefore stands to computer science as theoretical physics stands to physics.

4.2 The Scientific Epistemology

The *argument of complexity* (§3.2) demonstrates that deductive reasoning is impractical for large programs. The following arguments however demonstrate that the outcome of executing even very small and programs cannot be determined analytically.

The **argument of self-modifiability** for the unpredictability of programs concerns the fact that certain program-processes modify the very set of their instructions (the program-script) during the process of computation. For example, in genetic and evolutionary programming the program-script is treated as a chromosome, namely as a sequence of symbols that is subjected to *mutation* and *crossover* during the process of computation. Therefore a genetic program-process, even if entirely deterministic, does not follow a fixed set of instructions. Similarly, the instructions encoded in the program-script for computer viruses are modified by infected program-processes. For example, in the attempt to defeat anti-virus scanners, polymorphic viruses randomly change their effect—indeed their very program script (the virus 'signature')—arbitrarily with each 'infection'; thus, any instruction can change arbitrarily to any other instruction. As a result, the behaviour of 'infected' programs is veritably impossible to predict analytically, not even when government secrets or large fortunes are at stake. The behaviour of self-modifying programs of other kinds is almost equally volatile.

Once one program is contaminated, *any* other program-process sharing the same resources is likely to be affected. Since computer viruses and other forms of malware are likely to infect (at one point of another) almost every networked computer, almost *any* program in the Internet era carries the risk of becoming self-modifiable.

The **argument of non-linearity** for the unpredictability of programs relies on the fact that the vast majority of program-processes belong to the *deterministically chaotic* class of phenomena. Dynamic systems theory (also *complexity theory*), which accounts for a very large class of 'natural' phenomena (including weather systems, traffic jams, ecosystems, and stock markets), states that the outcome of chaotic and deterministically chaotic systems cannot be determined analytically because "tiny deviations of initial data lead to exponentially increasing computational efforts to analyze future data, limiting long-term predictions, although the dynamics is in principle uniquely determined". (Mainzer 2004)

A phenomenon is classified as 'deterministic chaos' if the following conditions hold:

(1) Arbitrarily close to every state s_1 of the system, there is a state s_2 whose future eventually is significantly different from that of s_1. That is, the tiniest changes can cause arbitrarily large changes in the future course of events.

(2) Arbitrarily close to every state s_1 of the system, there is a state s_2 whose future behaviour eventually returns exactly to s_2.

(3) Given any two states s_1 and s_2, the futures of some states near s_1 eventually become near s_2 (Devaney 1989).

For example, the future state of a program calculating the *n*th value of formula (4) for some $r > 3$ satisfies the conditions of deterministically chaotic phenomenon, and therefore cannot be determined analytically:

$$\text{State}(n+1) = r \times \text{State}(n) \times (1 - \text{State}(n)) \tag{4}$$

Already in 1946, before the principles of chaos theory have been developed and evidence to its widespread applicability has been presented, von Neumann observed that the outcome of programs computing non-linear mathematical functions cannot be analytically determined:

> Our present analytical methods seem unsuitable for the solution of the important problems arising in connection with non-linear partial differential equations and, in fact, with virtually all types of non-linear problems in pure mathematics. (von Neumann, in Mahoney 2002)

In 1979, DeMillo et al. illustrated how ‘chaotic’ computer programs are using the example of weather systems, for which an event as minute as the flap of a butterfly’s wings may potentially have a disproportionate effect, indeed a result as catastrophic as causing a hurricane:

> Every programmer knows that altering a line or sometimes even a bit can utterly destroy a program or mutilate it in ways that we do not understand and cannot predict. ... Until we know more about programming, we had better for all practical purposes think of systems as composed, not of sturdy structures like algorithms and smaller programs, but of butterflies’ wings. (DeMillo et al. 1979)

In other words, even if a program was not specifically encoded to calculate a non-linear function, in effect its behaviour amounts to such a program. The reason is that one part or another of it is non-linear. DeMillo et al. specifically mention operating systems and compliers, which in effect take large part in the behaviour or almost any program. Therefore, it is very unlikely that any knowledge about all but the most trivial programs can be established without conducting experiments.

Knuth conceded the weight of the argument of non-linearity, in particular with relation to the class of programs that are the concern of artificial life:

> It is abundantly clear that a programmer can create something and be totally aware of the laws that are obeyed by the program, and yet be almost totally unaware of the consequences of those laws; [for example,] running a program from a slightly different configuration often leads to really surprising new behaviour. (Knuth Undated)

Berry et al. corroborate the *argument of non-linearity* by showing that the very behaviour of microprocessors is chaotic when executing certain program-processes:

As a consequence, the performance of these microprocessors during the execution of certain programs displays complex non-repetitive variations that challenge traditional analysis.... Our results show that ... for several [programs], the complex dynamics observed result from deterministic chaos. This suggests that a detailed prediction of microprocessor performance at long execution times is unlikely with these programs. (Berry et al. 2005)

Without specifically referring to non-linearity, Turing, in a remark which can be taken to be an (anticipatory) rebuttal to Hoare (EPI-RAT), acknowledged already in 1950 that the behaviour of some programs is inevitably a source of surprises:

The view that machines cannot give rise to surprises is due, I believe, to a fallacy to which philosophers and mathematicians are particularly subject. This is the assumption that as soon as a fact is presented to a mind all consequences of that fact spring into the mind simultaneously with it. It is a very useful assumption under many circumstances, but one too easily forgets that it is false. (Turing 1950)

In conclusion from the compelling arguments of *complexity* (§2.3), *self-modifiability*, and *non-linearity* for the unpredictability of programs, the behaviour of some programs is inevitably a source of a surprise, and a priori knowledge about them is severely limited. Therefore, while it may be possible *in principle* to deduce some of the properties of the program and all the consequences of executing it (EPI-RAT), *in practice* it is very often impossible.

The tenets of the scientific epistemology can therefore be summarized as follows:

EPI-SCI While it may be possible *in principle* to deduce some of the properties of the program and all the consequences of executing it, *in practice* it is very often impossible. Therefore, while some knowledge about programs can be established *a priori*, much of what we know about programs must necessary be limited to some probabilistic, *a posteriori* notion of knowledge.

4.3 The Scientific Ontology

To him who is a discoverer ... the products of his imagination appear so necessary and natural that he regards them, and would like them regarded by others, not as creations of thought but as given realities.

—Albert Einstein (1934)

We postulate that an adequate ontological explanation for program-processes must offer an account for the following unique set of their apparent properties:

1. Temporal: The existence of program-processes extends in time in the interval between being created and being destroyed[19];
2. Non-physical: Program-processes are non-physical, intangible entities;
3. Causal: Program-processes can interact with and move physical devices;
4. Metabolic: Program-process 'consume' energy[20];
5. Contingent upon a physical manifestation: The existence of program-processes depends on the existence of that physical computer which is said to be 'executing' it;
7. Nonlinear: The outcome of a program-process, in the general case, cannot be analytically determined.

Let us examine briefly the weaknesses of the rationalist and of the technocratic ontological explanations with relation to the apparent properties of program-processes. Rationalism (ONT-RAT) asserts that programs are mathematical objects. But mathematical objects, such as turing machines, recursive functions, triangles, and numbers cannot be meaningfully said to metabolize nor have a causal effect on the physical reality in any immediate sense. It would be particular difficult to justify also a claim that mathematical objects have a specific lifespan or that they are contingent upon any specific physical manifestation (except possibly as mental artefacts). In this respect, ONT-RAT is inadequate.

Alternatively, the technocratic paradigm (ONT-TEC) reduces program-scripts to mere "bunches of data". It is hostile towards assertions of existence of any abstract, ontologically independent manifestations of whatever the data is taken to represent. But program-processes *do* have causal effect on physical reality: They control robotic arms, artificial limbs, machine guns (*BBC* 8-Apr-2006), 'smart bombs', the navigation of automated and semi-automated vehicles, the sale and purchase of stocks in stock exchanges, and to some degree almost every single home appliance. Programs also treat depression (*Medical News Today* 22-Feb-2006), determine whether your child shall receive her vaccination (*Observer* 26-Feb-2006), shortlist job applications (*Int'l Herald Tribune* 26-Sep-2006), count votes in national elections, and spread copies of themselves over the Internet. Program-processes came to have a tangible effect on concrete, physical reality, an effect which ONT-TEC fails to account for.

The inadequacy of both the rationalist and the technocratic ontological accounts has led Dijkstra to conclude that program-processes are a 'radical novelty':

> It is the most common way of trying to cope with novelty: by means of metaphors and analogies we try to link the new to the old, the novel to the familiar. Under sufficiently slow and gradual change, it works reasonably well; in the case of a sharp discontinuity, however, the method breaks down: though we may glorify it with the name "common sense", our past experience is no longer relevant, the analogies become too shallow, and the metaphors

[19] We ignore, for the moment, difficulties arising from concurrency and the possibility of suspending the execution of program-processes.

[20] That is, the computational process by the central processing unit depends on the consumption of energy; if suspended, program-processes cease to exist.

become more misleading than illuminating. This is the situation that is characteristic for the "radical" novelty. (Dijkstra 1988)

According to Dijkstra, the ontological question (ONT) remains open.

Others contend that the misleading similarities to mathematical objects and to engineered artefacts arise because program-processes are on a par with mental processes. For example, Alan Bundy calls them 'mental machines':

> The reason that it is possible to have this analogy both with applied mathematics and pure engineering is that computer programs are strange beasts; they are both mathematical entities and artifacts. They are formal abstract objects which can be investigated symbolically as if they were statements in some branch of mathematics. But they are also artifacts, in that they can do things, e.g., run a chemical plant. They are machines, but they are not physical machines, they are mental machines. (Bundy 2005, p. 218)

Indeed, cognitive and other mental processes are non-physical, causal, metabolic, contingent upon a physical manifestation (e.g., the human brain), and non-linear processes too. Bundy's metaphor is therefore adequate at least according to the criteria of their apparent properties listed above.

A symmetrical contention is made by computational theories of mind (McLaughlin 2004), which suggest that the brain is a (programmable) computer and that the computation of cognitive functions—that is, the exercise of mental abilities, or simply 'thinking'—is in effect a program-process. For example, Hilary Putnam (1975) and more recently Eric Steinhart (2003) argued that the human mind is in effect a finite-state automaton. Strong AI, which holds that intelligent mental processes—artificial "thinking" processes—can be effectively reproduced by executing programs using existing technology. Strong AI was upheld by the pioneers of AI and to this day it the working assumption of those computer scientists who investigate machine learning, evolutionary algorithms, and artificial life (Bedau 2004). The same stance was in effect taken by Turing as early as in 1950[21]:

> May not machines carry out something which ought to be described as thinking but which is very different from what a man does? This objection is a very strong one, but at least we can say that if, nevertheless, a machine can be constructed to play the imitation game satisfactorily, we need not be troubled by this objection. ... I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted. (Turing 1950)

The analogy to mental processes receives considerable support from recent results in computational theories of the DNA. Brent and Bruck suggest that the DNA molecule can be taken to be a program-script encoded in a turing-complete

[21] Turing forecast named the year 2000 as a target. During that year, Jim Moor conducted an experiment which refuted Turing's prediction, but he hastens to add: "Of course, eventually, 50 years from now or 500 years from now, an unrestricted Turing test might be passed routinely by some computers. If so, our jobs as philosophers would just be beginning". (Moor 2000)

(‘procedural’) programming language, and the mechanisms of interpreting it to be on a par with (turing-complete) digital computing machines:

> It seems reasonable to view the DNA script in the genome as executable code that could have been specified by a set of commands in a procedural imperative [programming] language. (Brent and Bruck 2006)

If a monist, materialist (Stack 1998) position is taken, then the human mind is indeed largely the product of the interpretation of the human genome. If the DNA representing the human brain is taken to be a program-script then program processes are indeed on a par with mental processes.

The scientific ontology and the arguments in its favour can thus be summarized as follows:

> **EPI-SCI** Program-scripts are on a par with DNA sequences, in particular with the genetic representation of human organs such as the brain, the product of whose execution—program-processes—are on a par with mental processes: temporal, non-physical, causal, metabolic, contingent upon a physical manifestation, and non-linear entities.

5 Discussion

We examined the basic tenets of three paradigms of computer science, each of which holds different positions concerning the definition of the discipline, warranted notions of program correctness, and whether programs are mathematical objects. We concluded that the disputes among computer scientists go beyond the boundaries of the discipline and extend to philosophical positions concerning the nature of computer programs and the nature of knowledge about them. We expanded on the arguments that corroborate the scientific position concerning these questions and concluded that, since almost all programs are non-linear or self-modifiable, *a priori* knowledge about them is unattainable. Therefore, the methods of computer science must combine deductive reasoning with scientific experimentation. Our analysis of the apparent properties of program-processes has also demonstrated that the category of mental process offer the most compelling account for their existence.

The significant increase in the complexity of software systems has lent much support to the *argument of complexity*, leading almost all who upheld the rationalist paradigm to abandon it[22]. But while most computer scientists pledge allegiance to the scientific position, at least in principle, mainstream computer science is yet to concede the ontological commitments of the scientific paradigm. Rather, since Wegner observer the prevalence of the technocratic paradigm in 1976, the failure of the methods of theoretical computer to deliver effective solutions to this crisis and the vested interest of the multi-billion dollar software industry (Ophir 2006) have

[22] Hoare (2006) has recently conceded that “Because of its effective combination of pure knowledge and applied invention, Computer Science can reasonably be classified as a branch of Engineering Science.”

contributed to the dominance of the technocratic doctrine in all but some branches of AI.

As a result of the increasing influence that the technocratic paradigm has been having on undergraduate curricula, ‘computer science’ academic programs are seldom true to their name. Courses teaching computability, complexity, automata theory, algorithmic theory, and even logic in undergraduate programs have been dropped in favour of courses focusing on technological trends teaching software design methodologies, software modelling notations (e.g., the Unified Modelling Language2005[23]), programming platforms, and component-based software engineering technologies. As a result, a growing proportion of academic programs churn increasing numbers of graduates in ‘computer science’ with no background in the theory of computing and no understanding of the theoritical foundations of the discipline.

In 1988, Dijkstra scathingly attacked the decline of mathematical, conceptual, and scientific principles, a trend which has turned computer science programmes into semi-professional schools which train students in commercially driven, short-lived technology:

> So, if I look into my foggy crystal ball at the future of computing science education, I overwhelmingly see the depressing picture of “Business as usual”. The universities will continue to lack the courage to teach hard science, they will continue to misguide the students, and each next stage of infantilization of the curriculum will be hailed as educational progress. (Dijkstra 1988)

It is difficult to determine precisely the outcome of the domination of the technocratic doctrine on computer science education. but the anti-scientific attitude has evidently taken its toll on the software industry. Since it was declared in the 1968 NATO conference (Naur and Randell 1969), the never-ending state of ‘software crisis’ has been renamed to ‘software’s chronic crisis’ (Gibbs 1994) and in 2005 it was pronounced ‘software hell’ (Carr 2004). The majority of multimillion-dollar software development projects, government and commercial, largely continues to end with huge losses and no gains (Carr 2004). As a standard, software manufacturers sign their clients on an End-User Licence Agreements (EULA) which offer less of a guarantee for their merchandise than any other commodity with the possible exception of casinos and used cars. Much of the professional literature refers to software in a jargon borrowed from mathematics, melodrama, and witchcraft in almost equal measures (e.g., Raymond 1996). Crimes involving bypassing security bots guarding the most heavily protected electronically stored secrets and spreading a wide spectrum of software malware have become part of daily life. The correct operation of the majority of computing devices has become largely dependent on daily—even hourly—updates of a host of defence mechanisms: firewalls, anti-virus, anti-spyware, anti-trojans, anti-worms, anti-dialers, anti-rootkits, etc. Even with the widespread use of these defence mechanisms, virtually no computer is invulnerable to malicious programs that disable and overtake global

[23] To which Bertrand Meyer (1997) satirical critique offers valuable insights.

networks of millions of zombie computers ('botnets') through the Internet. Paradoxically, the doctrine preached primarily by software engineers and practitioners has done little but deepen the disparity between the state of practice in 'software engineering' and estabilished engineering disciplines.

David Parnas, who became known for his contributions to software design (e.g., Parnas 1972) pointed out that even in software engineering the technocratic stance is untenable, upholding instead the basic tenets of the scientific paradigm:

> There is no engineering profession in which testing and mathematical validation are viewed as alternatives. It is universally accepted that they are complementary and that both are required. (David Parnas, in Denning 1989)

Parnas' argument is upheld by the analogy between software engineering and established and more successful branches of engineering such civil engineering, chemical engineering, and even genetic engineering. These branches of engineering would not exist if not for the rigour their scientific and theoretical counterparts, e.g., material sciences, chemistry, and molecular biology, respectively. Robin Milner (2007) concurs and concludes that indeed, the failures of software engineering emanate from the decline in the role of theoretical computer science and its methods. Therefore, before software engineering matures to that level of established engineering disciplines and stand to computer science as chemical engineering stands to chemistry, computer scientists must abandon the technocratic paradigm.

Epilogue

If the scientific paradigm comes to dominate, and mainstream computer science is recognized as a branch of natural sciences, the question is how computer science can mature as such. Quine offers the following criterion of maturity for a scientific discipline:

> A branch of science would qualify for recognition and classification at all ... only when it had matured to the point of clearing up its similarity standards [between natural kinds]. ... In general we can take it as a very special mark of the maturity of a branch of science that it no longer needs an irreducible notion of similarity and kind. It is that final stage where the animal vestige is wholly absorbed into the theory. (Quine 1969)

An interesting open question is therefore whether computer programs are natural kinds (Copeland 2006) and if not then what mature scientific theory of computer programs can lead us to better understanding of the technology that civilization has come to depend upon.

Acknowledgements Special thanks go to Ray Turner for reviewing draft arguments and for his guidance and continuous support, without which this paper would not have been possible; to Jack Copeland for his guidance on matters of traditional philosophy; and to Bill Rapaport for his detailed comments. We also thank Tim Colburn (2000) and Bill Rapaport (2005) without whose extensive contributions the nascent discipline of philosophy of computer science would not exist; Barry Smith for his guidance; Susan Stuart for developing the contentions made of this paper; Naomi Draaijer for her support; Yehuda Elkana, Saul Eden-Draaijer, and Mary J. Anna for their inspiration. This research was supported in part by grants from UK's Engineering and Physical Sciences Research Council and the Royal Academy of Engineering.

References

Abran, A., & Moore, J. W. (Eds.) (2004). *Guide to the Software Engineering Body of Knowledge—SWEBOK* (2004 ed.) Los Alamitos: IEEE Computer Society.

Abelson, H., Sussman, J.J. (1996). *Structure and Interpretation of Computer Programs*. (2nd ed.) Cambridge: MIT Press.

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools*. Reading: Addison Wesley.

Balaguer, M. (2004). Platonism in metaphysics. In: E. N. Zalta (Ed.), *The Stanford Encyclopedia of philosophy* (Summer 2004 ed.) Available http://plato.stanford.edu/archives/sum2004/entries/platonism. (Accessed March 2007.)

Bedau, M. A. (2004). Artificial life. In: L. Floridi (Ed.), *The Blackwell guide to philosophy of computing and information*. Malden: Blackwell.

Berry, H., Pérez, D. G., & Temam, O. (2005). Chaos in computer performance. *Nonlinear Sciences* arXiv:nlin.AO/0506030.

Brent, R., & Bruck, J. (2006). Can computers help to explain biology? *Nature, 440*, 416–417.

Bundy, A. (2005). What kind of field is AI? In: D. Partridge, & Y. Wilks (Eds.), *The foundations of artificial intelligence*. Cambridge: Cambridge university Press.

Carr, N. G. (2004). *Does IT matter? Information technology and the corrosion of competitive advantage*. Harvard Business School Press.

Cohn, A. (1989). The notion of proof in hardware verification. *Journal of Automated Reasoning, 5*(2), 127–139.

Colburn, T. R. (2000). *Philosophy and computer science*. Armonk, N.Y.: M.E. Sharpe.

Copeland, B.J. (2002). The Church-Turing thesis. In: Edward N. Zalta (Ed.) The Stanford Encyclopedia of Philosophy (Fall 2002 ed.) Available http://plato.stanford.edu/archives/fall2002/entries/church-turing/ (Accessed Mar. 2007).

Copeland, B.J. (2006). Are computer programs natural kinds? Personal correspondence.

Devaney, R. L. (1989). *Introduction to chaotic dynamical systems* (2nd ed.). Redwood: Benjamin-Cummings Publishing.

Debian Project, The. http://www.debian.org. Accessed March 2007.

DeMillo, R. A., Lipton, R. J., & Perlis, A. J. (1979). Social processes and proofs of theorems and programs. *Communications of the ACM, 22*(5), 271–280.

Denning, P. J. (1989). A debate on teaching computing science. *Communications of the ACM, 32*(12), 1397–1414.

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., & Young, P. R. (1989). Computing as a discipline. *Communication of the ACM, 32*(1), 9–23.

Dijkstra, E.W. (1988) On the cruelty of really teaching computing science. Unpublished manuscript EWD 1036.

Dybå, T., Kampenesa, V. B., & Sjøberg, D. I. K. (2006) A systematic review of statistical power in software engineering experiments. *Information and Software Technology, 48*(8), 745–755.

Eden, A. H., Hirshfeld, Y., & Kazman, R. (2006) Abstraction classes in software design. *IEE Software, 153*(4), 163–182. London, UK: The Institution of Engineering and Technology.

Einstein, A. (1934). *Mein Weltbild*. Amsterdam: Querido Verlag.

Fasli, M. (2007). *Agent technology for E-commerce*. London: Wiley.

Fetzer, J. H. (1993). Program verification. In: J. Belzer, A. G. Holzman, A. Kent, & J. G. Williams (Eds.), *Encyclopedia of computer science and technology* (Vol. 28, Supplement 13). New York: Marcel Dekker Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading: Addison-Wesley.

Georick, W., Hoffmann, U., Langmaack, & H. (1997). Rigorous compiler implementation correctness: How to prove the real thing correct. *Proc. Intl. Workshop Current Trends in Applied Formal Method*. Lecture Notes in Computer Science, Vol. 1641, pp. 122–136. London, UK: Springer-Verlag.

Gibbs, W. W. (1994) Software's chronic crisis. *Scientific American, 271*(3), 86–95.

Hall, A. (1990). Seven myths of formal methods. *IEEE Software, 7*(5), 11–19.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM, 12*(10), 576–583

Hoare, C. A. R. (1986). *The mathematics of programming: an inaugural lecture delivered before the Univ. of Oxford on Oct. 17, 1985*. New York: Oxford University Press

Hoare, C. A. R. (2006). The ideal of program correctness. Transcript of lecture, *Computer Journal*. London: British Computer Society. Available: http://www.bcs.org/upload/pdf/correctness.pdf (Accessed Mar. 2007)

IEEE Std 610.12-1990 (1990). *IEEE Standard Glossary of Software Engineering Terms*. Los Alamitos: IEEE Computer Society.

Knuth, D. E. (1968). *The art of computer programming, Vol. I: Fundamental algorithms*. Reading, MA: Addison Wesley.

Knuth, D. E. (1974). Computer science and its relation to mathematics. *The American Mathematical Monthly, 81*(4), 323–343.

Knuth E. D. (Undated). On the game of life, free will and determinism.(Video). Available: http://www.meta-library.net/ssq/sj1-body.html (Accessed Mar. 2007)

Kuhn, T. (1962). *The structure of scientific revolutions*. Chicago: University of Chicago Press.

Loux, M. J. (1998). Nominalism. *Routledge Encyclopedia of Philosophy* (electronic Ver. 1.0). London and New York: Routledge.

Mahoney, M. S. (2002). Software as science—Science as software. In: U. Hashagen, R. Keil-Slawik, & A. Norberg (Eds.), *History of computing: software issues*. Berlin: Springer Verlag.

Mainzer, K. (2004). System: an introduction to systems science. In: L. Floridi (Ed.), *The Blackwell guide to philosophy of computing and information*. Malden: Blackwell.

Markie, P. (2004). Rationalism vs. Empiricism. In: E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy*. Available http://plato.stanford.edu/archives/fall2004/entries/rationalism-empiricism (Accessed Mar. 2007).

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM, 3*(4), 184–195.

McCarthy, J. (1962). Towards a mathematical science of computation. *Proceedings of IFIP*.

McClure, R. M. (2001). Introduction. Addendum to: (Naur & Randell 1969). Available: http://www.homepages.cs.ncl.ac.uk/brian.randell/NATO/Introduction.html (Accessed Mar. 2007)

McLaughlin, B. (2004). Computationalism, connectionism, and the philosophy of mind. In: L. Floridi (Ed.), *The Blackwell guide to philosophy of computing and information*. Malden: Blackwell.

Meyer, B. (1997). UML—The Positive Spin. *American Programmer, 10*(3). Available: http://archive.eiffel.com/doc/manuals/technology/bmarticles/uml/page.html (Accessed Mar. 2007)

Milner, R. (2007). Memories of Gilles Kahn, and the informatic future. Transcript of speech before Colloquium in memory of Gilles Kahn, INRIA Research Institute.

Naur, P., & Randell, B. (Eds.) (1969). Software Engineering: Report of a conference sponsored by the NATO Science Committee (7–11 Oct. 1968), Garmisch, Germany. Brussels, Scientific Affairs Division, NATO.

Newell, A., & Simon, H. A. (1976). Completer science as empirical inquiry: Symbols and search. *Communications of the ACM, 19*(3), 113–126.

Olson, E. T. (1997). The ontological basis of strong artificial life. *Artificial Life, 3*(1), 29–39.

OMG (Object Management Group). (2005). *Unified Modeling Language (UML)*, Ver. 2.0. Technical report (2005). Available http://www.omg.org/technology/documents/formal/uml.htm (Accessed Mar. 2007)

Ophir, S. (2006). Computer science and commercial forces: Can computer science be considered science? In *Proc. 4th European conf. Computing And Philosophy—ECAP*, Trondheim, Norway.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053–1058.

Perry, D. E., & Wolf, A. L. (1992). Foundation for the study of software architecture. *ACM SIGSOFT Software Engineering Notes, 17*(4), 40–52.

Pierce, J. R. (1968). Keynote address. *Conference on Academic and Related Research Programs in Computing Science* (5–8 June 1967). Reprinted in: A. Finerman (Ed.), *University Education in Computing Science*. New York: Academic Press.

Popper, K. (1963). *Conjectures and refutations: The growth of scientific knowledge*. Routledge, London.

Putnam, H. (1975). Minds and machines. In: *Philosophical papers, Vol. 2: Mind, Language, and reality*. pp. 362–385. Cambridge: Cambridge University Press.

Quine, W. V. O. (1969). Natural kinds. In: *Ontological reality and other essays*. Columbia University Press.

Rapaport, W. J. (2007). Personal correspondence.

Rapaport, W. J. (2005). Philosophy of computer science: An introductory course. *Teaching Philosophy, 28*(4), 319–341.

Raymond, E. S. (1996). *The New Hacker's Dictionary* (3rd ed.). Cambridge: MIT Press.

Simon, H. A. (1969). *The sciences of the artificial* (1st ed.) Boston: MIT Press.

Sommerville, I. (2006). *Software engineering* (8th ed.) Reading: Addison Wesley.

Stack, G. S. (1998). Materialism. *Routledge Encyclopedia of Philosophy* (electronic Ver. 1.0). London and New York: Routledge.

Steinhart, E. (2003). Supermachines and superminds. *Minds and Machines, 13*(1), 155–186.

Stoy, J. E. (1977). *Denotational semantics: The Scott-Strachey approach to programming language theory*. Cambridge: MIT Press.

Strachey, C. (1973). *The varieties of programming language*. Tech. Rep. PRG-10 Oxford University Computing Laboratory.

Szyperski, C. A. (2002). *Component software—Beyond object-oriented programming* (2nd ed.). Reading: Addison-Wesley.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. In *Proc. London Math. Soc. Ser.*, 2, *43*(2198). Reprinted in Turing & Copeland (2004).

Turing, A. M. (1950). Computing machinery and intelligence. *Mind, 59*, 433–460.

Turing, A. M., & Copeland, B. J. (Ed.) (2004). *The essential Turing: Seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life plus the secrets of Enigma*. Oxford, USA: Oxford University Press.

Turner, R. (2005). The foundations of specification. *Journal of Logic & Computation, 15*(5), 623–663.

Turner, R. (2007). Personal correspondence.

Wegner, P. (1976). Research paradigms in computer science. In *Proc. 2nd Int'l Conf. Software engineering*, San Francisco, CA, pp. 322–330.