Stanford Encyclopedia of Philosophy

The Philosophy of Computer Science

First published Tue Aug 20, 2013; substantive revision Tue Jan 19, 2021

The philosophy of computer science is concerned with the ontological and methodological issues arising from within the academic discipline of computer science, and from the practice of software development and its commercial and industrial deployment. More specifically, the philosophy of computer science considers the ontology and epistemology of computational systems, focusing on problems associated with their specification, programming, implementation, verification and testing. The complex nature of computer programs ensures that many of the conceptual questions raised by the philosophy of computer science have related ones in the philosophy of mathematics, the philosophy of empirical sciences, and the philosophy of technology. We shall provide an analysis of such topics that reflects the layered nature of the ontology of computational systems in Sections 1–5; we then discuss topics involved in their methodology in Sections 6–8.

1. Computational Systems

Computational systems are widespread in everyday life. Their design, development and analysis are the proper object of study of the discipline of computer science. The philosophy of computer science treats them instead as objects of theoretical analysis. Its first aim is to define such systems, i.e., to develop an ontology of computational systems. The literature offers two main approaches on the topic. A first one understands computational systems as defined by distinct ontologies for software and hardware, usually taken to be their elementary components. A different approach sees computational systems as comprising several other elements around the software-hardware dichotomy: under this second view, computational systems are defined on the basis of a hierarchy of levels of abstraction, arranging hardware levels at the bottom of such a hierarchy and extending upwards to elements of the design and downwards to include the user. In the following we present these two approaches.

1.1 Software and Hardware

Usually, computational systems are seen as composed of two ontologically distinct entities: software and hardware. Algorithms, source codes, and programs fall in the first category of abstract entities; microprocessors, hard drives, and computing machines are concrete, physical entities.

Moore (1978) argues that such a duality is one of the three myths of computer science, in that the dichotomy software/hardware has a pragmatic, but not an ontological, significance. Computer programs, as the set of instructions a computer may execute, can be examined both at the symbolic level, as encoded instructions, and at the physical level, as the set of instructions stored in a physical medium. Moore stresses that no program exists as a pure abstract entity, that is, without a physical realization (a flash drive, a hard disk on a server, or even a piece of paper). Early programs were even hardwired directly and, at the beginning of the computer era, programs consisted only in patterns of physical levers. By the software/hardware opposition, one usually identifies software with the symbolic level of programs, and hardware with the corresponding physical level. The distinction, however, can be only pragmatically justified in that it delimits the different tasks of developers. For them, software may be given by algorithms and the source code implementing them, while hardware is given by machine code and the microprocessors able to execute it. By contrast, engineers realizing circuits implementing hardwired programs may be inclined to call software many physical parts of a computing machine. In other words, what counts as software for one professional may count as hardware for another one.

Suber (1988) goes even further, maintaining that hardware is a kind of software. Software is defined as any pattern that is amenable to being read and executed: once one realizes that all physical objects display patterns, one is forced to accept the conclusion that hardware, as a physical object, is also software. Suber defines a pattern as "any definite structure, not in the narrow sense that requires some recurrence, regularity, or symmetry" (1988, 90) and argues that any such structure can indeed be read and executed: for any definite pattern to which no meaning is associated, it is always possible to conceive a syntax and a semantics giving a meaning, thereby making the pattern an executable program.

Colburn (1999, 2000), while keeping software and hardware apart, stresses that the former has a dual nature, it is a "concrete abstraction" as being both abstract and concrete. To define software, one needs to make reference to both a "medium of description", i.e., the language used to express an algorithm, and a "medium of execution", namely the circuits composing the hardware. While software is always concrete in that there is no software without a concretization in some physical medium, it is nonetheless abstract, because programmers do not consider the implementing machines in their activities: they would rather develop a program executable by any machine. This aspect is called by Colburn (1999) "enlargement of content" and it defines abstraction in computer science as an "abstraction of content": content is enlarged rather than deleted, as it happens with mathematical abstraction.

Irmak (2012) criticizes the dual nature of software proposed by Colburn (1999, 2000). He understands an abstract entity as one lacking spatio-temporal properties, while being concrete means having those properties. Defining software as a concrete abstraction would therefore imply for software to have contradictory properties. Software does have temporal properties: as an object of human creation, it starts to exist at some time once conceived and implemented; and it can cease to exist at a certain subsequent time. Software ceases

to exist when all copies are destroyed, their authors die and nobody else remembers the respective algorithms. As an object of human creation, software is an artifact. However, software lacks spatial properties in that it cannot be identified with any concrete realization of it. Destroying all the physical copies of a given software would not imply that a particular software ceases to exist, as stated above, nor, for the very same reason, would deleting all texts implementing the software algorithms in some high-level language. Software is thus an abstract entity endowed with temporal properties. For these reasons, Irmak (2010) definies software as an *abstract artifact*.

Duncan (2011) points out that distinguishing software from hardware requires a finer ontology than the one involving the simple abstract/concrete dichotomy. Duncan (2017) aims at providing such an ontology by focusing on Turner's (2011) notion of specification as an expression that gives correctness conditions for a program (see §2). Duncan (2017) stresses that a program acts also as a specification for the implementing machine, meaning that a program specifies all correct behaviors that the machine is required to perform. If the machine does not act consistently with the program, the machine is said to malfunction, in the same way a program which is not correct with respect to its specification is said to be flawed or containing a bug. Another ontological category necessary to define the distinction software/hardware is that of artifact, which Duncan (2017) defines as a physical, spatio-temporal entity, which has been constructed so as to fulfill some functions and such that there is a community recognizing the artifact as serving that purpose. That said, software is defined as a set of instructions encoded in some programming language which act as specifications for an artifact able to read those instructions; hardware is defined as an artifact whose function is to carry out the specified computation.

1.2 The Method of Levels of Abstractions

As shown above, the distinction between software and hardware is not a sharp one. A different ontological approach to computational systems relies on the role of abstraction. Abstraction is a crucial element in computer science, and it takes many different forms. Goguen & Burstall (1985) describe some of this variety, of which the following examples are instances. Code can be repeated during programming, by naming text and a parameter, a practice known as procedural abstraction. This operation has its formal basis in the abstraction operation of the lambda calculus (see the entry on the lambda calculus) and it allows a formal mechanism known as polymorphism (Hankin 2004). Another example is typing, typical of functional programming, which provides an expressive system of representation for the syntactic constructors of the language. Or else, in object-oriented design, patterns (Gamma et al. 1994) are abstracted from the common structures that are found in software systems and used as interfaces between the implementation of an object and its specification.

All these examples share an underlying methodology in the Levels of Abstraction (henceforth LoA), used also in mathematics (Mitchelmore and White 2004) and philosophy (Floridi 2008). Abstractions in mathematics are piled upon each other in a never-ending search for more and more abstract concepts. On this account, abstraction is self-contained: an abstract mathematical object takes its meaning only from the system within which it is defined and the only constraint is that new objects be related to each other in a consistent system that can be operated on without reference to previous or external meanings. Some argue that, in this respect at least, abstraction in computer science is fundamentally different from abstraction in mathematics: computational abstraction must leave behind an implementation trace and this means that information is hidden but not destroyed (Colburn & Shute 2007). Any details that are ignored at one LoA must not be ignored by one of the lower LoAs: for example, programmers need not worry about the precise location in memory associated with a particular variable, but the virtual machine is required to handle all memory allocations. This reliance of abstraction on different levels is reflected in the property of computational systems to depend upon the existence of an implementation: for example, even though classes hide details of their methods, they must have implementations. Hence, computational abstractions preserve both an abstract guise and an implementation.

A full formulation of LoAs for the ontology of digital computational systems has been devised in Primiero (2016), including:

- Intention
- Specification
- Algorithm

- High-level programming language instructions
- Assembly/machine code operations
- Execution

Intention is the cognitive act that defines a computational problem to be solved: it formulates the request to create a computational process to perform a certain task. Requests of this sort are usually provided by customers, users, and other stakeholders involved in a given software development project. *Specification* is the formulation of the set of requirements necessary for solving the computational problem at hand: it concerns the possibly formal determination of the operations the software must perform, through the process known as requirements elicitation. *Algorithm* expresses the procedure providing a solution to the proposed computational problem, one which must meet the requirements of the specification. *High-level programming language* (such as C, Java, or Python) *instructions* constitute the linguistic implementation of the proposed algorithm, often called the source code, and they can be understood by trained programmers but cannot be directly executed by a machine. The instructions coded in high-level language are compiled, i.e., translated, by a compiler into *assembly code* and then assembled in *machine code operations*, executable by a processor. Finally, the *execution* LoA is the physical level of the running software, i.e., of the computer architecture executing the instructions.

According to this view, no LoA taken in isolation is able to define what a computational system is, nor to determine how to distinguish software from hardware. Computational systems are rather defined by the whole abstraction hierarchy; each LoA in itself expresses a semantic level associated with a realization, either linguistic or physical.

2. Intention and Specification

Intention refers to a cognitive state outside the computational system which expresses the formulation of a computational problem to be solved. Specifications describe the functions that the computational system to be developed must fulfil. Whereas intentions, *per se*, do not pose specific philosophical controversies inside the philosophy of computer science, issues arise in connection with the definition of what a specification is and its relation with intentions.

2.1 Intentions

Intentions articulate the criteria to determine whether a computational system is appropriate (i.e., correct, see §7), and therefore it is considered as the first LoA of the computational system appropriate to that problem. For instance, customers and users may require a smartphone app able to filter out annoying calls from call centers; such request constitutes the intention LoA in the development of a computational system able to perform such a task. In the software development process of non-naive systems, intentions are usually gathered by such techniques as brainstorming, surveys, prototyping, and even focus groups (Clarke and Moreira 1999), aimed at defining a structured set of the various stakeholders' intentions. At this LoA, no reference is made to *how* to solve the computational problem, but only the description of the problem *that* must be solved is provided.

In contemporary literature, intentions have been the object of philosophical inquiry at least since Anscombe (1963). Philosophers have investigated "intentions with which" an action is performed (Davidson 1963), intentions of doing something in the future (Davidson 1978), and intentional actions (Anscombe 1963, Baier 1970, Ferrero 2017). Issues arise concerning which of the three kinds of intention is primary, how they are connected, the relation between intentions and belief, whether intentions are or presuppose specific mental states, and whether intentions act as causes of actions (see the entry on intention). More formal problems concern the opportunity for an agent of having inconsistent intentions and yet being considered rational (Bratman 1987, Duijf *et al.* 2019).

In their role as the first LoA in the ontology of computational systems, intentions can certainly be acknowledged as intentions for the future, in that they express the objective of constructing systems able to perform some desired computational tasks. Since intentions, as stated above, confine themselves to the definition of the computational problem to be solved, without specifying its computational solution, their ontological and epistemological analysis does not differ from those referred to in the philosophical literature.

In other words, there is nothing specifically computational in the intentions defining computational systems which deserves a separate treatment in the philosophy of computer science. What matters here is the relation between intention and specification, in that intentions provide correctness criteria for specifications; specifications are asked to express how the computational problem put forward by intentions is to be solved.

2.2 Definitions and Specifications

Consider the example of the call filtering app again; a specification may require to create a black-list of phone numbers associated with call centers; to update the list every *n* days; to check, upon an incoming call, whether the number is on the black-list; to communicate to the call management system not to allow the incoming call in case of an affirmative answer, and to allow the call in case of negative answer.

The latter is a full-fledged specification, though expressed in a natural language. Specifications are often advanced in a natural language to be closer to the stakeaholder's intentions and only subsequently they are formalized in a proper formal language. Specifications may be expressed by means of graphical languages such as UML (Fowler 2003), or more formal languages such as TPL (Turner 2009a) and VDM (Jones 1990), using predicate logic, or Z (Woodcock and Davies 1996), focusing on set theory. For instance, Type Predicate Logic (TPL) expresses the requirements of computational systems using predicate logic formulas, wherein the type of the quantified variables is specified. The choice of the variable types allows one to define specifications at the more appropriate abstraction level. Whether specifications are expressed in an informal or formal guise often depends on the development method followed, with formal specifications usually preferred in the context of formal development methods. Moreover, formal specifications facilitate verification of correctness for computational systems (see §6).

Turner (2018) asks what difference is there between models and specifications, both of which are extensively used in computer science. The difference is located in what Turner (2011) calls the *intentional stance*: models *describe* an intended system to be developed and, in case of a mismatch between the two, the models are to be refined; specifications *prescribe* how the system is to be built so as to comply with the intended functions, and in case of mismatch it is the system that needs to be refined. Matching between model and system reflects a correspondence between intentions — describing *what* system is to be constructed in terms of the computational problem the system must be able to solve — and specifications — determining *how* the system is to be constructed, in terms of the set of requirements necessary for solving the computational problem, as exemplified for the call filtering app. In Turner's (2011) words, "something is a specification when it is given correctness jurisdiction over an artefact": specifications provide correctness criteria for computational systems. Computational systems are thus correct when they comply with their specifications, that is, when they behave according to them. Conversely, they provide criteria of malfunctioning (§7.3): a computational system malfunctions when it does not behave consistently with its specifications. Turner (2011) is careful to notice that such a definition of specifications is an idealization: specifications are themselves revised in some cases, such as when the specified computational systems cannot be realized because of physical laws constraints or cost limitations, or when it turns out that the advanced specifications are not correct formalizations of the intentions of clients and users.

More generally, the correctness problem does not only deal with specifications, but with any two LoAs defining computational systems, as the next subsection will examine.

2.3 Specifications and Functions

Fully implemented and constructed computational systems are *technical artifacts*, i.e., human-made systems designed and implemented with the explicit aim of fulfilling specific functions (Kroes 2012). Technical artifacts so defined include tables, screwdrivers, cars, bridges, or televisions, and they are distinct both from natural objects (e.g. rocks, cats, or dihydrogen monoxide molecules), which are not human-made, and artworks, which do not fulfill functions. As such, the ontology of computational systems falls under that of technical artifacts (Meijers 2000) characterized by a *duality*, as they are defined by both *functional* and *structural properties* (Kroes 2009, see also the entry on philosophy of technology). Functional properties specify the functions the artifact is required to perform; structural properties express the physical properties through which the artifact can perform them. Consider a screwdriver: functional properties may include the function of screwing and unscrewing; structural properties can refer to a piece of metal capable of being inserted on the head of the screw and a plastic handle that allows a clockwise and anticlockwise motion.

Functions can be realized in multiple ways by their structural counterparts. For instance, the function for the screwdriver could well be realized by a full metal screwdriver, or by an electric screwdriver defined by very different structural properties.

The layered ontology of computational systems characterized by many different LoAs seems to extend the dual ontology defining technical artifacts (Floridi *et al*. 2015). Turner (2018) argues that computational systems are still artifacts in the sense of (Kroes 2009, 2012), as each LoA is a functional level for lower LoAs and a structural level for upper LoAs:

- the intention expresses the functions that the system must achieve and is implemented by the specification;
- the specification plays a functional role, explaining in details the concrete functions that the software must implement, and it is realized by an algorithm, its structural level;
- the algorithm expresses the procedures that the high-level language program, its structural level, must implement;
- instructions in high level language define the functional properties for the machine language code, which realizes them;
- machine code, finally, expresses the functional properties implemented by the execution level, which expresses physical structural properties.

It follows, according to Turner (2018), that structural levels need not be necessarily physical levels, and that the notion of abstract artifact holds in computer science. For this reason, Turner (2011) comes to define high-level language programs themselves as technical artifacts, in that they constitute a structural level implementing specifications as their functional level (see §4.2).

A first consequence is that each LoA – expressing *what* function to accomplish – can be realized by a multiplicity of potential structural levels expressing *how* those functions are accomplished: an intended functionality can be realized by a specification in multiple ways; a computational problem expressed by a specification has solutions by a multiplicity of different algorithms, which can differ for some important properties but are all equally valid (see §3); an algorithm may be implemented in different programs, each written in a different high-level programming language, all expressing the *same* program if they implement the same algorithm (Angius and Primiero 2019); source code can be compiled in a multiplicity of machine languages, adopting different ISAs (Instruction Set Architectures); executable code can be installed and run on a multiplicity of machines (provided that these share the same ISA).

A second consequence is that each LoA as a functional level provides correctness criteria for lower levels (Primiero 2020). Not just at the implementation level, correctness is required at any LoA from specification to execution, and the cause of malfunctions may be located at any LoA not correctly implementing its proper functional level (see §7.3 and Fresco, Primiero (2013)). According to Turner (2018), the specification level can be said to be correct or incorrect with respect to intentions, despite the difficulty of verifying their correctness. Correctness of any non-physical layer can be verified mathematically through formal verification, and the execution physical level can be verified empirically, through testing (§6). Verifying correctness of specifications with respect to clients' intentions would require instead having access to the mental states of the involved agents.

This latter problem relates to the more general one of establishing how artifacts possess functions, and what it means that structural properties are related to the intentions of agents. The problem is well-known also in the philosophy of biology and the cognitive sciences, and two main theories have been put forward as solutions. According to the *causal theory of function* (Cummins 1975), functions are determined by the physical capacities of artifacts: for example, the physical ability of the heart of contracting and expanding determines its function of pumping blood in the circulatory system. However, this theory faces serious problems when applied to technical artifacts. First, it prevents defining correctness and malfunctioning (Kroes 2010): suppose the call filtering app installed on our smartphone starts banning calls from contacts in our mobile phonebook; according to the causal theory of function this would be a new function of the app. Second, the theory does not distinguish intended functions from side effects (Turner 2011): in case of a long-lasting call, our smartphone would certainly start heating; however, this is not a function intended by clients or developers. According to the *intentional theory of function* (McLaughlin 2001, Searle 1995), the function fixed by the designer or the user is the intended one of the artifact, and structural properties of artifacts are selected so as

to be able to fulfill it. This theory is able to explain correctness and malfunction, as well as to distinguish side effects from intended functions. However, it does not say where the function actually resides, whether in the artifact or in the mind of the agent. In the former case, one is back at the question of how artifacts possess functions. In the latter case, a further explanation is needed about how mental states are related to physical properties of artifacts (Kroes 2010). Turner (2018) holds that the intuitions behind both the causal and the intentional theories of function are useful to understand the relation between function and structure in computational systems, and suggests that the two theories be combined into a single one. On the one hand, there is no function without implementation; on the other hand, there is no intention without clients, developers, and users.

3. Algorithms

Even though known and widely used since antiquity, the problem of defining what algorithms are is still open (Vardi 2012). The word "*algorithm*" originates from the name of the ninth-century Persian mathematician Abū Jaʿfar Muḥammad ibn Mūsā *al-Khwārizmī*, who provided rules for arithmetic operations using Arabic numerals. Indeed, the rules one follows to compute basic arithmetic operations such as multiplication or division, are everyday examples of algorithms. Other well-known examples include rules to bisect an angle using compass and straightedge, or Euclid's algorithm for calculating the greatest common divisor. Intuitively, an algorithm is a set of instructions allowing the fulfillment of a given task. Despite this ancient tradition in mathematics, only modern logical and philosophical reflection put forward the task of providing a definition of what an algorithm is, in connection with the foundational crisis of mathematics of the early twentieth century (see the entry on the philosophy of mathematics). The notion of *effective calculability* arose from logical research, providing some formal counterpart to the intuitive notion of algorithm and giving birth to the theory of computation. Since then, different definitions of algorithms have been proposed, ranging from formal to non-formal approaches, as sketched in the next sections.

3.1 Classical Approaches

Markov (1954) provides a first precise definition of algorithm as a computational process that is *determined*, *applicable*, and *effective*. A computational process is *determined* if the instructions involved are precise enough not to allow for any "arbitrary choice" in their execution. The (human or artificial) computer must never be unsure about what step to carry out next. Algorithms are *applicable* for Markov in that they hold for classes of inputs (natural numbers for basic arithmetic operations) rather than for single inputs (specific natural numbers). Markov (1954:1) defines *effectiveness* as "the tendency of the algorithm to obtain a certain result". In other words, an algorithm is effective in that it will eventually produce the answer to the computational problem.

Kleene (1967) specifies *finiteness* as a further important property: an algorithm is a procedure which can be described by means of a finite set of instructions and needs a finite number of steps to provide an answer to the computational problem. As a counterexample, consider a *while* loop defined by a finite number of steps, but which runs forever since the condition in the loop is always satisfied. Instructions should also be amenable to *mechanical* execution, that is, no insight is required for the machine to follow them. Following Markov's determinability and strengthening effectiveness, Kleene (1967) additionally specifies that instructions should be able to recognize that the solution to the computational problem has been achieved, and halt the computation.

Knuth (1973) recalls and deepens the analyses of Markov (1954) and Kleene (1967) by stating that:

> Besides merely being a finite set of rules that gives a sequence of operations for solving a specific type of problem, an algorithm has five important features:
>
> 1. *Finiteness*. An algorithm must always terminate after a finite number of steps. […]
> 2. *Definiteness*. Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case. […]
> 3. *Input*. An algorithm has zero or more inputs. […]
> 4. *Output*. An algorithm has zero or more outputs. […]

5. *Effectiveness*. An algorithm is also generally expected to be effective, in the sense that its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper. (Knuth 1973: 4–6) […]

As in Kleene (1967), finiteness affects both the number of instructions and the number of implemented computational steps. As in Markov's determinacy, Knuth's definiteness principle requires that each successive computational step be unambiguously specified. Furthermore, Knuth (1973) more explicitly requires that algorithms have (potentially empty sets of) inputs and outputs. By algorithms with no inputs or outputs Knuth probably refers to algorithms using internally stored data as inputs or algorithms not returning data to an external user (Rapaport 2019, ch. 7, in Other Internet Resources). As for effectiveness, besides Markov's tendency "to obtain a certain result", Knuth requires that the result be obtained in a finite amount of time and that the instructions be atomic, that is, simple enough to be understandable and executable by a human or artificial computer.

3.2 Formal Approaches

Gurevich (2011) maintains, on the one hand, that it is not possible to provide formal definitions of algorithms, as the notion continues to evolve over time: consider how sequential algorithms, used in ancient mathematics, are flanked by parallel, analog, or quantum algorithms in current computer science practice, and how new kinds of algorithms are likely to be envisioned in the near future. On the other hand, a formal analysis can be advanced if concerned only with classical sequential algorithms. In particular, Gurevich (2000) provides an axiomatic definition for this class of algorithms.

Any sequential algorithm can be simulated by a sequential abstract state machine satisfying three axioms:

1. The *sequential-time* postulate associates to any algorithm *A* a set of states *S(A)*, a set of initial states *I(A)* subset of *S(A)*, and a map from *S(A)* to *S(A)* of one-step transformations of *A*. States are snapshot descriptions of running algorithms. A run of *A* is a (potentially infinite) sequence of states, starting from some initial state, such that there is a one-step transformation from one state to its successor in the sequence. Termination is not presupposed by Gurevich's definition. One-step transformations need not be atomic, but they may be composed of a bounded set of atomic operations.
2. According to the *abstract-state postulate*, states in *S(A)* are *first-order structures*, as commonly defined in mathematical logic; in other words, states provide a semantics to first-order statements.
3. Finally, the *bounded-exploration* postulate states that given two states *X* and *Y* of *A* there is always a set *T* of terms such that, when *X* and *Y* coincide over *T*, the set of updates of *X* corresponds to the set of updates of *Y*. *X* and *Y* coincide over *T* when, for every term *t* in *T*, the evaluation of *t* in *X* is the same as the evaluation of *t* in *Y*. This allows algorithm *A* to explore only those parts of states which are relative to terms in *T*.

Moschovakis (2001) objects that the intuitive notion of algorithm is not captured in full by abstract machines. Given a general recursive function f: $\mathbb{N} \rightarrow \mathbb{N}$ defined on natural numbers, there are usually many different algorithms computing it; "essential, implementation-independent properties" are not captured by abstract machines, but rather by a *system of recursive equations*. Consider the algorithm *mergesort* for sorting lists; there are many different abstract machines for *mergesort*, and the question arises which one is to be chosen as the *mergesort* algorithm. The *mergesort* algorithm is instead the system of recursive equations specifying the involved function, whereas abstract machines for the *mergesort* procedure are *different implementations of the same algorithm*. Two questions are put forward by Moschovakis' formal analysis: different implementations of the *same* algorithm should be *equivalent* implementations, and yet, an equivalence relation among algorithm implementations is to be formally defined. Furthermore, it remains to be clarified what the intuitive notion of algorithm formalized by systems of recursive equations amounts to.

Primiero (2020) proposes a reading of the nature of algorithms at three different levels of abstraction. At a very high LoA, algorithms can be defined abstracting from the procedure they describe, allowing for many different sets of states and transitions. At this LoA algorithms can be understood as *informal specifications*, that is, as informal descriptions of a procedure *P*. At a lower LoA, algorithms specify the instructions needed to solve the given computational problem; in other words, they specify a procedure. Algorithms can thus be defined as *procedures*, or descriptions in some given formal language *L* of how to execute a procedure *P*. Many important properties of algorithms, including those related to complexity classes and data structures,

cannot be determined at the procedural LoA, and instead make reference to an abstract machine implementing the procedure is needed. At a bottom LoA, algorithms can be defined as *implementable abstract machines*, viz. as the specification, in a formal language L, of the executions of a program P for a given abstract machine M. The threefold definition of algorithms allows Primiero (2020) to supply a formal definition of equivalence relations for algorithms in terms of the algebraic notions of *simulation* and *bisimulation* (Milner 1973, see also Angius and Primero 2018). A machine M_i executing a program P_i implements the same algorithm of a machine M_j executing a program P_j if and only if the abstract machines interpreting M_i and M_j are in a bisimulation relation.

3.3 Informal Approaches

Vardi (2012) underlines how, despite the many formal and informal definitions available, there is no general consensus on what an algorithm is. The approaches of Gurevich (2000) and Moschovakis (2001), which can even be proved to be logically equivalent, only provide logical constructs for algorithms, leaving unanswered the main question. Hill (2013) suggests that an informal definition of algorithms, taking into account the intuitive understanding one has about algorithms, may be more useful, especially for the public discourse and the communication between practitioners and users.

Rapaport (2012, Appendix) provides an attempt to summarize the three classical definitions of algorithm sketched above stating that:

> An algorithm (for executor E to accomplish goal G) is:
>
> 1. a procedure, that is, a finite set (or sequence) of statements (or rules, or instructions), such that each statement is:
>   - composed of a finite number of symbols (or marks) from a finite alphabet
>   - and unambiguous for E—that is,
>   i. E knows how to do it
>   ii. E can do it
>   iii. it can be done in a finite amount of time
>   iv. and, after doing it, E knows what to do next—
> 2. which procedure takes a finite amount of time (that is, it halts),
> 3. and that ends with G accomplished.

Rapaport stresses that an algorithm is a procedure, i.e., a finite sequence of statements taking the form of rules or instructions. Finiteness is here expressed by requiring that instructions contain a finite number of symbols from a finite alphabet.

Hill (2016) aims at providing an informal definition of algorithm, starting from Rapaport's (2012):

> An algorithm is a finite, abstract, effective, compound control structure, imperatively given, accomplishing a given purpose, under given provisions.(Hill 2016: 48).

First of all, algorithms are *compound structures* rather than atomic objects, i.e., they are composed of smaller units, namely computational steps. These structures are finite and effective, as explicitly mentioned by Markov, Kleene, and Knuth. While these authors do not explicitly mention abstractness, Hill (2016) maintains it is implicit in their analysis. Algorithms are *abstract* simply in that they lack spatio-temporal properties and are independent from their instances. They provide *control*, that is, "content that brings about some kind of change from one state to another, expressed in values of variables and consequent actions" (p. 45). Algorithms are *imperatively given*, as they command state transitions to carry out specified operations. Finally, algorithms operate to achieve certain *purposes* under some usually well-specified *provisions*, or preconditions. From this viewpoint, the author argues, algorithms are on a par with specifications in their specifying a goal under certain resources. This definition allows to distinguish algorithms from other compound control structures. For instance, recipes are not algorithms because they are not effective; nor are games, which are not imperatively given.

4. Programs

The ontology of computer programs is strictly related to the subsumed nature of computational systems (see §1). If computational systems are defined on the basis of the software-hardware dichotomy, programs are abstract entities interpreting the former and opposed to the concrete nature of hardware. Examples of such interpretations are provided in §1.1 and include the "concrete abstraction" definition by Colburn (2000), the "abstract artifact" characterization by Irmak (2012), and programs as specifications of machines proposed by Duncan (2011). By contrast, under the interpretation of computational systems by a hierarchy of LoAs, programs are implementations of algorithms. We refer to §5 on implementation for an analysis of the ontology of programs in this sense. This section focuses on definitions of programs with a significant relevance in the literature, namely those views that consider programs as theories or as artifacts, with a focus on the problem of the relation between programs and the world.

4.1 Programs as Theories

The view that programs are theories goes back to approaches in cognitive science. In the context of the so-called Information Processing Psychology (IPP) for the simulative investigation on human cognitive processes, Newell and Simon (1972) advanced the thesis that simulative programs are *empirical theories* of their simulated systems. Newell and Simon assigned to a computer program the role of theory of the simulated system as well as of the simulative system, namely the machine running the program, to formulate predictions on the simulated system. In particular, the execution traces of the simulative program, given a specific problem to solve, are used to predict the mental operation strategies that will be performed by the human subject when asked to accomplish the same task. In case of a mismatch between execution traces and the verbal reports of the operation strategies of the human subject, the empirical theory provided by the simulative program is revised. The predictive use of such a computer program is comparable, according to Newell and Simon, to the predictive use of the evolution laws of a system that are expressed by differential or difference equations.

Newell and Simon's idea that programs are theories has been shared by the cognitive scientists Pylyshyn (1984) and Johnson-Laird (1988). Both agree that programs, in contrast to typical theories, are better at facing the complexity of the simulative process to be modelled, forcing one to fill-in all the details that are necessary for the program to be executed. Whereas incomplete or incoherent theories may be advanced at some stage of scientific inquiry, this is not the case for programs.

On the other hand, Moore (1978) considers the programs-as-theories thesis another myth of computer science. As programs can only simulate some set of empirical phenomena, at most they play the role of computational *models* of those phenomena. Moore notices that for programs to be acknowledged as models, semantic functions are nevertheless needed to interpret the empirical system being simulated. However, the view that programs are models should not be mistaken for the definition of programs as theories: theories *explain* and *predict* the empirical phenomena simulated by models, while simulation by programs does not offer that.

According to computer scientist Paul Thagard (1984), understanding programs as theories would require a *syntactic* or a *semantic* view of theories (see the entry on the structure of scientific theories). But programs do not comply with either of the two views. According to the syntactic view (Carnap 1966, Hempel 1970), theories are sets of sentences expressed in some defined language able to describe target empirical systems; some of those sentences define the axioms of the theory, and some are law-like statements expressing regularities of those systems. Programs are sets of instructions written in some defined programming language which, however, do not describe any system, insofar as they are procedural linguistic entities and not declarative ones. To this, Rapaport (2020, see Other Internet Resources) objects that procedural programming languages can often be translated into declarative languages and that there are languages, such as Prolog, that can be interpreted both procedurally and declaratively. According to the semantic view (Suppe 1989, Van Fraassen 1980), theories are introduced by a collection of models, defined as set-theoretic structures satisfying the theory's sentences. However, in contrast to Moore (1978), Thagard (1984) denies programs the epistemological status of models: programs simulate physical systems without satisfying theories' laws and axioms. Rather, programs include, for simulation purposes, implementation details for the programming language used, but not of the target system being simulated.

A yet different approach to the problem of whether programs are theories comes from the computer scientist Peter Naur (1985). According to Naur, programming is a theory building process not in the sense that

programs are theories, but because the successful program's development and life-cycle require that programmers and developers have theories of programs available. A theory is here understood, following Ryle (2009), as a corpus of knowledge shared by a scientific community about some set of empirical phenomena, and not necessarily expressed axiomatically or formally. Theories *of* programs are necessary during the program life-cycle to be able to manage requests of program modifications pursuant to observed miscomputations or unsatisfactory solutions to the computational problem the program was asked to solve. In particular, theories of programs should allow developers to modify the program so that new solutions to the problem at stake can be provided. For this reason, Naur (1985) deems such theories more fundamental, in software development, than documentations and specifications.

For Turner (2010, 2018 ch. 10), programming languages are mathematical objects defined by a formal grammar and a formal semantics. In particular, each syntactic construct, such as an assignment, a conditional or a while loop, is defined by a grammatical rule determining its syntax, and by a semantic rule associating a meaning to it. Depending on whether an operational or a denotational semantics is preferred, meaning is given in terms of respectively the operations of an abstract machine or of mathematical partial functions from set of states to set of states. For instance, the simple assignment statement $x := E$ is associated, under an operational semantics, with the machine operation $update(s, x, v)$ which assigns variable v interpreted as E to variable x in state s. Both in the case of an operational and of a denotational semantics, programs can be understood as mathematical theories expressing the operations of an implementing machine. Consider operational semantics: a syntactic rule of the form $\langle P, s \rangle \Downarrow s'$ states semantically that program P executed in state s results in s'. According to Turner (2010, 2018), a programming language with an operational semantics is akin to an *axiomatic theory of operations* in which rules provide axioms for the relation $\Downarrow$.

4.2 Programs as Technical Artifacts

Programs can be understood as technical artifacts because programming languages are defined, as any other artifact, on the basis of both functional and structural properties (Turner 2014, 2018 ch. 5). Functional properties of (high level) programming languages are provided by the semantics associated with each syntactic construct of the language. Turner (2014) points out that programming languages can indeed be understood as axiomatic theories only when their functional level is isolated. Structural properties, on the other hand, are specified in terms of the implementation of the language, but not identified with physical components of computing machines: given a syntactic construct of the language with an associated functional description, its structural property is determined by the physical operations that a machine performs to implement an instruction for the construct at hand. For instance, the assignment construct $x := E$ is to be linked to the physical computation of the value of expression E and to the placement of the value of E in the physical location x.

Another requirement for a programming language to be considered a technical artifact is that it has to be endowed with a semantics providing correctness criteria for the language implementation. The programmer attests to functional and structural properties of a program by taking the semantics to have correctness jurisdiction over the program.

4.3 Programs and their Relation to the World

The problem of whether computer programs are theories is tied with the relation that programs entertain with the outside world. If programs were theories, they would have to represent some empirical system, and a semantic relation would be directly established between the program and the world. By contrast, some have argued that the relation between programs and natural systems is mediated by models of the outside world (Colburn *et al*. 1993, Smith 1985). In particular, Smith (1985) argues that models are abstract descriptions of empirical systems, and computational systems operating in them have programs that act as models of the models, i.e., they represent abstract models of reality. Such an account of the ontology of programs comes in handy when describing the correctness problem in computer science (see § 7): if specifications are considered as models requiring certain behaviors from computational systems, programs can be seen as models satisfying specifications.

Two views of programs can be given depending on whether one admits their relation with the world (Rapaport 2020, ch. 17, see Other Internet Resource). According to a first view, programs are "wide",

"external" and "semantic": they grant direct reference to objects of an empirical system and operations on those objects. According to a second view, programs are "narrow", "internal", and "syntactic": they make only reference to the atomic operations of an implementing machine carrying out computations. Rapaport (2020, see Other Internet Resources) argues that programs *need not* be "external" and "semantic". First, computation itself needs not to be "external": a Turing machine executes the instructions contained in its finite table by using data written on its tape and halting after the data resulting from the computation have been written on the tape. Data are not, strictly speaking, in-put-from and out-put-to an external user. Furthemore, Knuth (1973) required algorithms to have *zero* or more inputs and outputs (see [§ 3.1](#)). A computer program requiring no inputs may be a program, say, outputting all prime numbers from 1; and a program with no outputs can be a program that computes the value of some given variable x without returning the value stored in x as output. Second, programs need not be "external", teleological, i.e., goal oriented. This view opposes other known positions in the literature. Suber (1988) argues that, without considering goals and purposes, it would not be possible to assess whether a computer program is correct, that is, if it behaves as intended. And as recalled in [§3.3](#)., Hill (2016) specifies in her informal definition that algorithms accomplish "a given purpose, under given provisions." (Hill 2016: 48). To these views, Rapaport (2020, ch. 17, see Other Internet Resource) replies that whereas goals, purposes, and programmers' intentions may be very useful for a human computor to understand a program, they are not necessary for an artificial computer to carry out the computations instructed by the program code. Indeed, the principle of effectiveness that classical approaches require for algorithms (see [§3.1](#)) demands, among other properties, that algorithms be executed without any recourse to intuition. In other words, a machine executing a program for adding natural numbers does not "understand" that it is adding; at the same time, knowing that a given program performs addition may help a human agent to understand the program's code.

According to this view, computing involves just symbols, not meanings. Turing machines become symbols manipulators and not a single but multiple meanings can be associated with its operations. How can then one identify when two programs are the *same* program, if not by their meanings, that is, by considering what function they perform? One answer comes from Piccini's analysis of computation and its "*internal semantics*" (Piccini 2008, 2015 ch. 3): two programs can be identified as identical by analysing only their syntax and the operations the programs carry out on their symbols. The effects of string manipulation operations can be considered an internal semantics of a program. The latter can be easily determined by isolating subroutines or methods in the program's code and can afterwards be used to identify a program or to establish whether two programs are the same, namely when they are defined by the same subroutines.

However, it has been argued that there are cases in which it is not possible to determine whether two programs are the same without making reference to an external semantics. Sprevak (2010) proposes to consider two programs for addition which differ from the fact that one operates on Arabic, the other one on Roman numerals. The two programs compute the same function, namely addition, but this cannot always be established by inspecting the code with its subroutines; it must be determined by assigning content to the input/output strings, interpreting Arabic and Roman numerals as numbers. In that regard, Angius and Primiero (2018) underline how the problem of identity for computer programs does not differ from the problem of identity for natural kinds (Lowe 1998) and technical artifacts (Carrara et al. 2014). The problem can be tackled by fixing an identity criterion, namely a formal relation, that any two programs should entertain in order to be defined as identical. Angius and Primiero (2018) show how to use the process algebra relation of bisimulation between the two automata implemented by two programs under examination as such an identity criterion. Bisimulation allows to establish matching structural properties of programs implementing the same function, as well as providing weaker criteria for copies in terms of simulation. This brings the discussion back to the notion of programs as implementations. We now turn to analyze this latter concept.

5. Implementation

The word 'implementation' is often associated with a physical realization of a computing system, i.e., to a machine executing a computer program. In particular, according to the dual ontology of computing systems examined in [§1.1](#), implementation in this sense reduces to the structural hardware, as opposed to the functional software. By contrast, following the method of the levels of abstraction ([§ 1.2](#)), implementation becomes a wider relation holding between any LoA defining a computational system and the levels higher in the hierarchy. Accordingly, an algorithm is an implementation of a (set of) specification(s); a program

expressed in a high level programming language can be defined as an implementation of an algorithm (see §4); assembly and machine code instructions can be seen as an implementation of a set of high-level programming language instructions with respect to a given ISA; finally, executions are physical, observable, implementations of those machine code instructions. By the same token, programs formulated in a high-level language are also implementations of specifications, and, as similarly argued by the dual-ontology paradigm, executions are implementations of high-level programming language instructions. According to Turner (2018), even the specification can be understood as an implementation of what has been called intention.

What remains to be examined here is the nature of the implementation relation thus defined. Analyzing this relation is essential to define the notion of *correctness* (§7). Indeed, a correct program amounts to a correct implementation of an algorithm; and a correct computing system is a correct implementation of a set of specifications. In other words, under this view, the notion of correctness is paired with that of implementation for any LoA: any level can be said to be correct with respect to upper levels if and only if it is a correct implementation thereof.

The following three subsections examine three main definitions of the implementation relation that have been advanced in the philosophy of computer science literature.

5.1 Implementation as Semantic Interpretation

A first philosophical analysis of the notion of implementation in computer science is advanced by Rapaport (1999, 2005). He defines an implementation *I* as the *semantic interpretation* of a syntactic or abstract domain *A* in a medium of implementation *M*. If implementation is understood as a relation holding between a given LoA and any upper level in the hierarchical ontology of a computational system, it follows that Rapaport's definition extends accordingly, so that any LoA provides a semantic interpretation in a given medium of implementation for the upper levels. Under this view, specifications provide semantic interpretations of intentions expressed by stakeholders in the specification (formal) language, and algorithms provide semantic interpretations of specifications using one of the many languages algorithms can be formulated in (natural languages, pseudo-code, logic languages, functional languages etc.). The medium of implementation can be either abstract or concrete. A computer program is the implementation of an algorithm in that the former provides a semantic interpretation of the syntactic constructs of the latter in a high-level programming language as its medium of implementation. The program's instructions interpret the algorithm's tasks in a programming language. Also the execution LoA provides a semantic interpretation of the assembly/machine code operations into the medium given by the structural properties of the physical machine. According to the analysis in (Rapaport 1999, 2005), implementation is an asymmetric relation: if *I* is an implementation of *A*, *A* cannot be an implementation of *I*. However, the author argues that any LoA can be both a syntactic and a semantic level, that is, it can play the role of both the implementation I and of a syntactic domain A. Whereas an algorithm is assigned a semantic interpretation by a program expressed in a high-level language, the same algorithm provides a semantic interpretation for the specification. It follows that the abstraction-implementation relation pairs the functional-structural relation for computational systems.

Primiero (2020) considers this latter aspect as one main limit of Rapaport's (1999, 2005) account of implementation: implementation reduces to a *unique* relation between a syntactic level and its semantic interpretation and it does not account for the layered ontology of computational systems seen in §1.2. In order to extend the present definition of implementation to all LoAs, each level has to be reinterpreted each time either as syntactic or as a semantic level. This, in turn, has a repercussion on the second difficulty characterizing, according to Primero (2020), implementation as a semantic interpretation: on the one hand, this approach does not take into account *incorrect* implementations; on the other hand, for a given incorrect implementation, the unique relation so defined can relate incorrectness only to one syntactic level, excluding all other levels as potential error locations.

Turner (2018) aims to show that semantic interpretation not only does not account for incorrect implementation, but not even to correct ones. One first example is provided by the implementation of one language into another: the implementing language here is not providing a semantic interpretation of the implemented language, unless the former is associated with a semantics providing meaning and correctness criteria for the latter. Such semantics will remain external to the implementation relation: whereas correctness is associated with semantic interpretation, implementation does not always come with a semantic interpretation. A second example is given by considering an abstract stack implemented by an array; again,

the array does not provide correctness criteria for the stack. Quite to the contrary, it is the stack that specifies correctness criteria for any of its implementation, arrays included.

5.2 Implementation as the Relation Specification-Artifact

The fact that correctness criteria for the implementation relation are provided by the abstract level induces Turner (2012, 2014, 2018) to define implementation as the relation *specification-artefact*. As examined in §2, specifications have correctness jurisdiction over artifacts, that is, they prescribe the allowed behaviors of artifacts. Also recall that artifacts can be both abstract and concrete entities, and that any LoA can play the role of specification for lower levels. This amounts to saying that the specification-artefact relation is able to define any implementation relation across the layered ontology of computational systems.

Depending on how the specification-artifact relation is defined, Turner (2012) distinguishes as many as three different notions of implementation. Consider the case of a physical machine implementing a given abstract machine. According to an *intentional* notion of implementation, an abstract machine works as a specification for a physical machine, provided it advances all the functional requirements the latter must fulfill, i.e., it specifies (in principle) all the allowed behaviors of the implementing physical machine. According to an *extensional* notion of implementation, a physical machine is a correct implementation of an abstract machine if and only if isomorphisms can be established mapping states of the latter to states of the former, and transitions in the abstract machine correspond to actual executions (computational traces) of the artifact. Finally, an *empirical* notion of implementation requires the physical machine to display computations that match those prescribed by the abstract machine; that is to say, correct implementation has to be evaluated empirically through testing.

Primiero (2020) underlines how, while this approach addresses the issue of correctness and miscomputation as it allows to distinguish a correct from an incorrect implementation, it still identifies a unique implementation relation between a specification level and an artifact level. Again, if this account is allowed to involve the layered ontology of computational systems by reinterpreting each time any LoA either as a specification or artifact, Turner's account prevents from referring to more than one level at the same time as the cause of miscomputation: a miscomputation always occurs here as an incorrect implementation of a specification by an artifact. By defining implementation as a relation holding accross all the LoAs, one would be able to identify multiple incorrect implementations which do not directly refer to the abstract specification. A miscomputation may indeed be caused by an incorrect implementation of lower levels which is then inherited all the way down to the execution level.

5.3 Implementation for LoAs

Primiero (2020) proposes a definition of implementation not as a relation between two fixed levels, but one that is allowed to range over any LoA. Under this view, an implementation *I* is a *relation of instantiation* holding between a LoA and any other one higher in the abstraction hierarchy. Accordingly, a physical computing machine is an implementation of assembly/machine code operations; by transitivity, it can also be considered as an instantiation of a set of instructions expressed in high-level programming language instructions. A program expressed in a high-level language is an implementation of an algorithm; but it can also be taken to be the instantiation of a set of specifications.

Such a definition of implementation allows Primiero (2020) to provide a general definition of correctness: a physical computing system is correct if and only if it is characterized by correct implementations at any LoA. Hence correctness and implementation are coupled and defined at any LoA. *Functional correctness* is the property of a computational system that displays the functionalities required by the specifications of that system. *Procedural correctness* characterizes computational systems displaying the functionalities intended by the implemented algorithms. And *executional correctness* is defined as the property of a system that is able to correctly execute the program on its architecture. Each of these forms of correctness can also be classified quantitatively, depending on the amount of functionalities being satisfied. A functionally *efficient* computational system displays a minimal subset of the functionalities required by the specifications; a functionally *optimal* system is able to display a maximal subset of those functionalities. Similarly, the author defines procedurally as well as executionally efficient and optimal computational systems.

5.4 Physical Computation

According to this definition, implementation shifts from level to level: a set of algorithms defining a computational system are implemented as procedures in some formal language, as instructions in a high-level language, or as operations in a low-level programming language. An interesting question is whether *any* system, beyond computational artifacts, implementing procedures of this sort qualifies as a computational system. In other words, asking about the nature of physical implementation amounts to asking what is a computational system. If any system implementing an algorithm would qualify as computational, the class of such systems could be extended to biological systems, such as the brain or the cell; to physical systems, including the universe or some portion of it; and eventually to any system whatsoever, a thesis known as *pancomputationalism* (for an exhaustive overview on the topic see Rapaport 2018).

Traditionally, a computational system is intended as a *mechanical artifact* that takes input data, elaborates them *algorithmically* according to a set of instructions, and returns manipulated data as outputs. For instance, von Neumann (1945, p.1) states that "An automatic computing system is a (usually highly composite) device, which can carry out instructions to perform calculations of a considerable order of complexity". Such an informal and well-accepted definition leaves some questions open, including whether computational systems have to be machines, whether they have to process data algorithmically and, consequently, whether computations have to be Turing complete.

Rapaport (2018) provides a more explicit characterization of a computational system defined as any "physical plausible implementation of anything logically equivalent to a universal Turing machine". Strictly speaking personal computers are not physical Turing machines, but register machines are known to be Turing equivalent. To qualify as computational, systems must be *plausible* implementations thereof, in that Turing machines, contrary to physical machines, have access to infinite memory space and are, as abstract machines, error free. According to Rapaport's (2018) definition, *any* physical implementation of this sort is thus a computational system, including natural systems. This raises the question about which class of natural systems is able to implement Turing equivalent computations. Searle famously argued that anything can be an implementation of a Turing machine, or of a logical equivalent model (Searle 1990). His argument levers on the fact that being a Turing machine is a syntactic property, in that it is all about manipulating tokens of 0's and 1's. According to Searle, syntactic properties are not intrinsic to physical systems, but they are assigned to them by an observer. In other words, a physical state of a system is not intrinsically a computational state: there must be an observer, or user, who assigns to that state a computational role. It follows that any system whose behavior can be described as syntactic manipulation of 0's and 1's is a computational system.

Hayes (1997) objects to Searle (1990) that if everything was a computational system, the property "being a computational system" would become vacuous, as all entities would possess it. Instead, there are entities which are computational systems, and entities which are not. Computational systems are those in which the patterns received as inputs and saved into memory are able to change themselves. In other words, Hayes makes reference to the fact that stored inputs can be both data and instructions and that instructions, when executed, are able to modify the value of some input data. "If it were paper, it would be 'magic paper' on which writing might spontaneously change, or new writing appear" (Hayes 1997, p. 393). Only systems able to act as "magic paper" can be acknowledged as computational.

A yet different approach comes from Piccinini (2007, 2008) in the context of his mechanistic analysis of physical computations (Piccinini 2015; see also the entry on [computation in physical systems](#)). A physical computing system is a system whose behaviors can be *explained mechanistically* by describing the computing mechanism that brings about those behaviors. Mechanisms can be defined by "entities and activities organized such that they are productive of regular changes from start or set-up to finish or termination condition" (Machamer et al. 2000; see the entry on [mechanisms in science](#)). Computations, as physical processes, can be understood as those mechanisms that "generate output strings from input strings in accordance with general rules that apply to all input strings and depend on the input (and sometimes internal states)" (Piccinini 2007, p. 108). It is easy to identify set-up and termination conditions for computational processes. Any system which can be explained by describing an underlying computing mechanism is to be considered a computational system. The focus on explanation helps Piccinini avoid the Searlean conclusion that any system is a computational system: even if one may interpret, in principle, any given set of entities and activities as a computing mechanism, only the need to explain a certain observed phenomenon in terms of a computing mechanism defines the system under examination as computational.

6. Verification

A crucial step in the software development process is verification. This consists in the process of evaluating whether a given computational system is correct with respect to the specification of its design. In the early days of the computer industry, validity and correctness checking methods included several design and construction techniques, see for example (Arif *et al*. 2018). Nowadays, correctness evaluation methods can be roughly sorted into two main groups: formal verification and testing. Formal verification (Monin and Hinchey 2003) involves a proof of correctness with mathematical tools; software testing (Ammann and Offutt 2008) rather consists in running the implemented program to observe whether performed executions comply or not with the advanced specifications. In many practical cases, a combination of both methods is used (see for instance Callahan *et al*. 1996).

6.1 Models and Theories

Formal verification methods require a *representation* of the software under verification. In *theorem proving* (see van Leeuwen 1990), programs are represented in terms of axiomatic systems and a set of rules of inference representing the pre- and post-conditions of program transitions. A proof of correctness is then obtained by deriving formulas expressing specifications from the axioms. In *model checking* (Baier and Katoen 2008), a program is represented in terms of a state transition system, its property specifications are formalised by temporal logic formulas (Kröger and Merz 2008), and a proof of correctness is achieved by a depth-first search algorithm that checks whether those formulas hold of the state transition system.

Axiomatic systems and state transition systems used for correctness evaluation can be understood as *theories* of the represented artifacts, in that they are used to predict and explain their future behaviors. Methodologically state transition systems in model checking can be compared with scientific models in empirical sciences (Angius and Tamburrini 2011). For instance, Kripke Structures (see Clarke *et al*. 1999 ch. 2) are in compliance with Suppes' (1960) definition of scientific models as set-theoretic structures establishing proper mapping relations with models of data collected by means of experiments on the target empirical system (see also the entry on [models in science](#)). Kripke Structures and other state transition systems utilized in formal verification methods are often called system specifications. They are distinguished from common specifications, also called property specifications. The latter specify some required behavioral properties the program to be encoded must instantiate, while the former specify (in principle) all potential executions of an already encoded program, thus allowing for algorithmic checks on its traces (Clarke *et al*. 1999). In order to achieve this goal, system specifications are considered as *abductive* structures, *hypothesizing* the set of potential executions of a target computational system on the basis of the program's code and the allowed state transitions (Angius 2013b). Indeed, once it has been checked whether some temporal logic formula holds of the modeled Kripke Structure, the represented program is empirically tested against the behavioral property corresponding to the checked formula, in order to evaluate whether the model-hypothesis is an adequate representation of the target computational system. Accordingly, property specifications and system specifications differ also in their intentional stance (Turner 2011): the former are requirements *on* the program to be encoded, the latter are (hypothetical) descriptions of the encoded program. The descriptive and abductive character of state transition systems in model checking is an additional and essential feature putting state transition systems on a par with scientific models.

6.2 Testing and Experiments

Testing is the more 'empirical' process of launching a program and observing its executions in order to evaluate whether they comply with the supplied property specifications. Such technique is extensively used in the software development process. Philosophers and philosophically-minded computer scientists have considered software testing under the light of traditional methodological approaches in scientific discovery (Snelting 1998; Gagliardi 2007; Northover *et al*. 2008; Angius 2014) and questioned whether software tests can be acknowledged as *scientific experiments* evaluating the correctness of programs (Schiaffonati and Verdicchio 2014, Schiaffonati 2015; Tedre 2015).

Dijkstra's well-known dictum "Program testing can be used to show the presence of bugs, but never to show their absence" (Dijkstra 1970, p.7), introduces Popper's (1959) principle of *falsifiability* into computer science (Snelting 1998). Testing a program against an advanced property specification for a given interval of

time may exhibit some failures, but if no failure occurs while observing the running program one cannot conclude that the program is correct. An incorrect execution might be observed at the very next system's run. The reason is that testers can only launch the program with a finite subset of the potential program's input set and only for a finite interval of time; accordingly, not all potential executions of the program to be tested can be empirically observed. For this reason, the aim of software testing is to detect programs' faults and not to guarantee their absence (Ammann and Offutt 2008, p. 11). A program is falsifiable in that tests can reveal faults (Northover *et al.* 2008). Hence, given a computational system and a property specification, a test is akin to a scientific experiment which, by observing the system's behaviors, tries to falsify the hypothesis that the program is correct with respect to the interested specification.

However, other methodological and epistemological traits characterizing scientific experiments are not shared by software tests. A first methodological distinction can be recognized in that a falsifying test leads to the revision of the computational system, not of the hypothesis, as in the case of testing scientific hypotheses. This is due to the difference in the intentional stance of specifications and empirical hypotheses in science (Turner 2011). Specifications are requirements whose violation demands for program revisions until the program becomes a correct instantiation of the specifications.

For this, among other reasons, the traditional notion of scientific experiment needs to be 'stretched' in order to be applied to software testing activities (Schiaffonati 2015). *Theory-driven experiments*, characterizing most of the experimental sciences, find no counterpart in actual computer science practice. If one excludes the cases wherein testing is combined with formal methods, most experiments performed by software engineers are rather *explorative*, *i.e.* aimed at 'exploring' "the realm of possibilities pertaining to the functioning of an artefact and its interaction with the environment in the absence of a proper theory or theoretical background" (Schiaffonati 2015: 662). Software testers often do not have theoretical control on the experiments they perform; exploration on the behaviors of the computational system interacting with users and environments rather allows testers to formulate theoretical generalizations on the observed behaviors. Explorative experiments in computer science are also characterized by the fact that programs are often tested in a real-like environment wherein testers play the role of users. However, it is an essential feature of theory-driven experiments that experimenters do not take part in the experiment to be carried out.

As a result, while some software testing activities are closer to the experimental activities one finds in empirical sciences, some others rather define a new typology of experiment that turns out to belong to the software development process. Five typologies of experiments can be distinguished in the process of specifying, implementing, and evaluating computational systems (Tedre 2015):

- *feasibility experiments* are performed to evaluate whether a system performs the functions specified by users and stakeholders;
- *trial experiments* are carried out to evaluate isolated capabilities of the system given some set of initial conditions;
- *field experiments* are performed in real environments and not in simulated ones;
- *comparison experiments* test similar systems, instantiating in different ways the same function, to evaluate which instantiation better performs the desired function both in real-like and real environments;
- finally, *controlled* experiments are used to appraise advanced hypotheses on the behaviors of the testing computational system and are the only ones on a par with scientific theory-driven experiments, in that they are carried out on the basis of some theoretical hypotheses under evaluation.

6.3 Explanation

A software test is considered successful when miscomputations are detected (assuming that no computational artifact is 100% correct). The successive step is to find out what caused the execution to be incorrect, that is, to trace back the fault (more familiarly named 'bug'), before proceeding to the debugging phase and then testing the system again. In other words, an *explanation* of the observed miscomputation is to be advanced.

Efforts have been made to consider explanations in computer science (Piccinini 2007; Piccinini and Craver 2011; Piccinini 2015; Angius and Tamburrini 2016) in relation to the different models of explanations elaborated in the philosophy of science. In particular, computational explanations can be understood as a specific kind of *mechanistic explanation* (Glennan 1996; Machamer *et al.* 2000; Bechtel and Abrahamsen

2005), insofar as computing processes can be analyzed as mechanisms (Piccinini 2007; 2015; see also the entry on computation in physical systems).

Consider a processor executing an instruction. The involved process can be understood as a mechanism whose components are states and combinatory elements in the processor instantiating the functions prescribed by the relevant hardware specifications (specifications for registers, for the Arithmetic Logic Unit etc..), organized in such a way that they are capable of carrying out the observed execution. Providing the description of such a mechanism counts as advancing a mechanist explanation of the observed computation, such as the explanation of an operational malfunction.

For every type of miscomputation (see §7.3), a corresponding mechanist explanation can be defined at the adequate LoA and with respect to the set of specifications characterizing that LoA. Indeed, abstract descriptions of mechanisms still supply one with a mechanist explanation in the form of a mechanism *schema*, defined as "a truncated abstract description of a mechanism that can be filled with descriptions of known component parts and activities" (Machamer *et al*. 2000, p. 15). For instance, suppose the very common case in which a machine miscomputes by executing a program containing syntax errors, called slips. The computing machine is unable to correctly implement the functional requirements provided by the program specifications. However, for explanatory purposes, it would be redundant to provide an explanation of the occurred slip at the hardware level of abstraction, by advancing the detailed description of the hardware components and their functional organization. In such cases, a satisfactory explanation may consist in showing that the program's code is not a correct instantiation of the provided program specifications (Angius and Tamburrini 2016). In order to explain mechanistically an occurred miscomputation, it may be sufficient to provide the description of the incorrect program, abstracting from the rest of the computing mechanism (Piccinini and Craver 2011). Abstraction is a virtue not only in software development and specification, but also in the explanation of computational systems' behaviors.

7. Correctness

Each of the different approaches on software verification examined in the previous section assumes a different understanding of correctness for software. Standardly, correctness has been understood as a relation holding between an abstraction and its implementation, such that it holds if the latter fulfills the properties formulated by the former. Once computational systems are described as having a layered ontology, correctness needs to be reformulated as the relation that any structural level entertains with respect to its functional level (Primiero, 2020). Hence, correctness can still be considered as a mathematical relationship when formulated between abstract and functional level; while it can be considered as an empirical relationship when formulated between the functional and the implementation levels. One of the earlier debates in the philosophy of computer science (De Millo *et al*. 1979; Fetzer 1988) was indeed around this distinction.

7.1 Mathematical Correctness

Formal verification methods grant an *a-priori* analysis of the behaviors of programs, without requiring the observation of any of their implementation or considering their execution. In particular, theorem proving allows one to *deduce* any potential behavior of the program under consideration and its behavioral properties from a suitable axiomatic representation. In the case of model checking, one knows in advance the behavioural properties displayed by the execution of a program by performing an algorithmic search of the formulas valid in a given set-theoretic model. These considerations famously led Hoare (1969) to conclude that program development is an "exact science", which should be characterized by mathematical proofs of correctness, epistemologically on a par with standard proofs in mathematical practice.

De Millo et al. (1979) question Hoare's thesis: correct mathematical proofs are usually *elegant* and *graspable*, implying that any (expert) reader can "see" that the conclusion follows from the premises (for the notion of elegance in software see also Hill (2018)). What are often called *Cartesian proofs* (Hacking 2014) do not have a counterpart in correctness proofs, typically long and cumbersome, difficult to grasp and not explaining why the conclusion necessarily follows from the premises. Yet, many proofs in mathematics are long and complex, but they are in principle *surveyable*, thanks to the use of lemmas, abstractions and the analytic construction of new concepts leading step by step to the statement to be proved. Correctness proofs,

on the contrary, do not involve the creation of new concepts, nor the modularity one typically finds in mathematical proofs (Turner, 2018). And yet, proofs that are not surveyable cannot be considered mathematical proofs (Wittgenstein 1956).

A second theoretical difficulty concerning proofs of correctness for computer programs concerns their complexity and that of the programs to be verified. Already Hoare (1981) admitted that while verification of correctness is always possible in principle, in practice it is hardly achievable. Except for trivial cases, contemporary software is modularly encoded, is required to satisfy a large set of specifications, and it is developed so as to interact with other programs, systems, users. Embedded and reactive software are cases in point. In order to verify such complex software, correctness proofs are carried out automatically. Hence, on the one hand, the correctness problem shifts from the program under examination to the program performing the verification, e.g. a theorem prover; on the other hand, proofs carried out by a physical process can go wrong, due to mechanical mistakes of the machine. Against this infinite regress argument, Arkoudas and Bringsjord (2007) argue that one can make use of a proof checker which, by being a relatively small program, is usually easier to verify.

Most recently, formal methods for checking correctness based on a combination of logical and statistical analysis have given new stimulus to this research area: the ability of Separation Logics (Reynolds, 2002) to offer a representation of the logical behavior of the physical memory of computational systems, and the possibility of considering probabilistic distributions of inputs as statistical source of errors, have allowed formal correctness check of large interactive systems like the Facebook platform (see also Pym *et al*. 2019).

7.2 Physical Correctness

Fetzer (1988) objected that deductive reasoning is only able to guarantee for the correctness of a program with respect to its specifications, but not for the correctness of a computational system, that is also accounting for the program's physical implementation. Even if the program were correct with respect to any of the related upper LoAs (algorithms, specifications, requirements), its implementation could still violate one or more of the intended specifications due to a physical malfunctioning. The former kind of correctness can in principle be proved mathematically, but the correctness of the execution LoA requires an empirical assessment. As examined in §6.2, software testing can show only in principle the correctness of a computational system. In practice, the number of allowed executions of non-trivial systems are potentially infinite and cannot be exhaustively checked in a finite (or reasonable) amount of time (Dijkstra 1974). Most successful testing methods rather see both formal verification and testing used together to reach a satisfactory correction level.

Another objection to the theoretical possibility of mathematical correctness is that since proofs are carried out by a theorem prover, i.e. a physical machine, the knowledge one attains about computational systems is not *a-priori* but empirical (see Turner 2018 ch. 25). However, Burge (1988) argues that computer-based proofs of correctness can still be regarded as *a-priori*, in that even though their possibility depends on sensory experience, their justification does not (as it is for *a-posteriori* knowledge). For instance, the knowledge that red is a color is *a-priori* even though it requires having sensory experience of red; this is because 'red is a colour' is true independently of any sensory experience. For further discussion on the nature of the use of computers in mathematical proofs, see (Hales 2008; Harrison 2008; Tymoczko 1979, 1980).

The problem of correctness eventually reduces to asking what it means for a physical machine to satisfy an abstract requirement. According to the *simple mapping account*, a computational system S is a correct implementation of specification SP only if:

    i. there can be established a morphism from the states ascribed to S to the states defined by SP, and

    ii. for any state transition $s_1 \to s_2$ in S there is a state transition $s_1' \to s_2'$ in SP between state s_1' mapping to s_1 and state s_2' mapping to s_2.

The simple mapping account only demands for an extensional agreement between the description of S and SP. The weakness of this account is that it is quite easy to identify an extensional agreement between any couple of physical system-specification, leaving room for a pancomputationalist perspective.

The danger of pancomputationalism has led some authors to attempt an account of correct implementation that somehow restricts the class of possible interpretations. In particular,

1. The *causal account* (D. J. Chalmers 1996; Copeland 1996) suggests that the material conditional (if the system is in the physical state s_1 …) is replaced by a counterfactual one.
2. The *semantic account* argues that a computational system must be associated with a semantic description, specifying what the system is to achieve (Sprevak 2012). For example, a physical device could be interpreted as an AND gate or an OR gate but without a definition of the device there is no way of fixing what the artifact is.
3. The *syntactic account* demands that only physical states that can be defined as syntactic can be mapped onto computational states. What remains to be examined is what defines a syntactic state (see Piccinini 2015 or the entry on [computation in physical systems](#) for an overview of the *syntactic account*
4. The *normative account* (Turner 2012) maintains not only that abstract and physical computational processes must be in agreement, but also that the abstract specification has normative force over the system. According to such an account, computations are physical processes whose function is fixed by an abstract specification. This relationship is stronger than both the semantic account, asking for a simple descriptive relationship, and the syntactic account, focusing on a syntactic object and its semantic interpretation.

7.3 Miscomputations

From what has been said so far, it follows that correctness of implemented programs does not automatically establish the well-functioning of a computational system. Turing (1950) already distinguished between *errors of functioning* and *errors of conclusion*. The former are caused by a faulty implementation unable to execute the instructions of some high-level language program; errors of conclusion characterize correct abstract machines that nonetheless fail to carry out the tasks they were supposed to accomplish. This may happen in those cases in which a program instantiates correctly some specifications which do not properly express the users' requirements on such a program. In both cases, machines implementing correct programs can still be said to miscompute.

Turing's distinction between errors of functioning and errors of conclusion has been expanded into a complete taxonomy of miscomputations (Fresco and Primiero 2013). The classification is established on the basis of the different LoAs defining computational systems. Errors can be:

- *conceptual*: they violate validity conditions requiring consistency for specifications expressed in propositional conjunctive normal form;
- *material*: they violate the correctness requirements of programs with respect to the set of their specifications;
- *performable*: they arise when physical constraints are breached by some faulty implementing hardware.

Performable errors clearly emerge only at the execution level, and they correspond with Turing's (1950) error of functioning, also called *operational malfunctions*. Conceptual and material errors may arise at any level of abstraction from the intention level down to the physical implementation level. Conceptual errors engender *mistakes*, while material errors induce *failures*. For instance, a mistake at the intention level consists of an inconsistent set of requirements, while at the physical implementation level it may correspond to an invalid hardware design (such as in the choice of the logic gates for the truth-functional connectives). Failures occurring at the specification level may be due to a design that is deemed to be incomplete with respect to the set of desired functional requirements, while a failure at the algorithm level occurs in those frequent cases in which the algorithm is found not to fulfill the specifications. Beyond mistakes, failures, and operational malfunctions, *slips* are a source of miscomputations at the high-level programming language instructions level: they may be conceptual or material errors due to, respectively, a syntactic or a semantic flaw in the program. Conceptual slips appear in all those cases in which the syntactical rules of high-level languages are violated; material slips involve the violation of semantic rules of programming languages, such as when a variable is used but not initialized.

A further distinction has to be made between *dysfunctions* and *misfunctions* for software-based computational systems (Floridi, Fresco and Primiero 2015). Software can only misfunction but cannot ever dysfunction. A software token can dysfunction in case its physical implementation fails to satisfy intentions or specifications.

Dysfunctions only apply to single tokens since a token dysfunctions in that it does not behave as the other tokens of the same type do with respect to the implemented functions. For this reason, dysfunctions do not apply to the intention level and the specification level. On the contrary, both software types and tokens can misfunction, since misfunctions do not depend on comparisons with tokens of the same type being able to perform some implemented function or not. Misfunction of tokens usually depends on the dysfunction of some other component, while misfunction of types is often due to poor design. A software token cannot dysfunction, because all tokens of a given type implement functions specified uniformly at the intention and specification levels. Those functions are implemented at the algorithm implementation level before being performed at the execution level; in case of correct implementation, all tokens will behave correctly at the execution level (provided that no operational malfunction occurs). For the very same reason, software tokens cannot misfunction, since they are implementations of the same intentions and specifications. Only software types can misfunction in case of poor design; misfunctioning software types are able to correctly perform their functions but may also produce some undesired side-effect. For the application of the notion of malfunctioning to the problem of malware classification, see (Primiero et al. 2019).

8. The Epistemological Status of Computer Science

Between the 1960s and the 1970s, computer science emerged as an academic discipline independent from its older siblings, mathematics and physics, and with it the problem of defining its epistemological status as influenced by mathematical, empirical, and engineering methods (Tedre and Sutien 2008, Tedre 2011, Tedre 2015, Primiero 2020). A debate is still in place today concerning whether computer science has to be *mostly* considered as a mathematical discipline, a branch of engineering, or as a scientific discipline.

8.1 Computer Science as a Mathematical Discipline

Any epistemological characterization of computer science is based on ontological, methodological, and epistemological commitments, namely on assumptions about the nature of computational systems, the methods guiding the software development process, and the kind of reasoning thereby involved, whether deductive, inductive, or a combination of both (Eden 2007).

The origin of the analysis of computation as a mathematical notion came notoriously from logic, with Hilbert's question concerning the decidability of predicate calculus, known as the *Entschiedungsproblem* (Hilbert and Ackermann 1950): could there be a mechanical procedure for deciding of an arbitrary sentence of logic whether it is provable? To address this question, a rigorous model of the informal concept of an effective or mechanical method in logic and mathematics was required. This is first and foremost a mathematical endeavor: one has to develop a mathematical analogue of the informal notion. Supporters of the view that computer science is mathematical in nature assume that a computer program can be seen as a physical realization of such a mathematical entity and that one can reason about programs deductively through the formal methods of theoretical computer science. Dijkstra (1974) and Hoare (1986) were very explicit in considering programs' instructions as mathematical sentences, and considering a formal semantics for programming languages in terms of an axiomatic system (Hoare 1969). Provided that program specifications and instructions are advanced in the same formal language, formal semantics provide the means to prove correctness. Accordingly, knowledge about the behaviors of computational systems is acquired by the deductive reasoning involved in mathematical proofs of correctness. The reason at the basis of such a rationalist optimism (Eden 2007) about what can be known about computational systems is that they are artifacts, that is, *human-made* systems and, as such, one can predict their behaviors with certainty (Knuth 1974).

Although a central concern of theoretical computer science, the topics of computability and complexity are covered in existing entries on the [Church-Turing thesis](), [computational complexity theory](), and [recursive functions]().

8.2 Computer Science as an Engineering Discipline

In the late 1970s, the increasing number of applications of computational systems in everyday contexts, and the consequent booming of market demands caused a deviation of interests for computer scientists in Academia and in Industry: from focusing on methods of proving programs' correctness, they turned to

methods for managing complexity and evaluating the reliability of those system (Wegner 1976). Indeed, expressing formally the specifications, structure, and input of highly complex programs embedded in larger systems and interacting with users is practically impossible, and hence providing mathematical proofs of their correctness becomes mostly unfeasible. Computer science research developed in the direction of testing techniques able to provide a statistical evaluation of correctness, often called reliability (Littlewood and Strigini 2000), in terms of estimation of error distributions in a program's code.

In line with this engineering account of computer science is the thesis that reliability of computational systems is evaluated in the same way that civil engineering does for bridges and aerospace engineering for airplanes (DeMillo *et al*. 1979). In particular, whereas empirical sciences examine what exists, computer science focuses on what *can* exist, i.e., on how to produce artifacts, and it should be therefore acknowledged as an "engineering of mathematics" (Hartmanis 1981). Similarly, whereas scientific inquiries are involved in discovering laws concerning the phenomena under observation, one cannot identify proper laws in computer science practice, insofar as the latter is rather involved in the production of phenomena concerning computational artifacts (Brooks 1996).

8.3 Computer Science as a Scientific Discipline

As examined in [§6](#), because software testing and reliability measuring techniques are known for their incapability of assuring for the absence of code faults (Dijkstra 1970), in many cases, and especially for the evaluation of the so-called safety-critical systems (such as controllers of airplanes, rockets, nuclear plants etc..), a combination of formal methods and empirical testing is used to evaluate correctness and dependability. Computer science can accordingly be understood as a scientific discipline, in that it makes use of both deductive and inductive probabilistic reasoning to examine computational systems (Denning et al. 1981, 2005, 2007; Tichy 1998; Colburn 2000).

The thesis that computer science is, from a methodological viewpoint, on a par with empirical sciences traces back to Newell, Perlis, and Simon's 1967 letter to Science (Newell et al. 1967) and dominated all the 1980's (Wegner 1976). In the 1975 Turing Award lecture, Newell and Simon argued:

> Computer science is an empirical discipline. We would have called it an experimental science, but like astronomy, economics, and geology, some of its unique forms of observation and experience do not fit a narrow stereotype of the experimental method. Nonetheless, they are experiments. Each new machine that is built is an experiment. Actually constructing the machine poses a question to nature; and we listen for the answer by observing the machine in operation and analyzing it by all analytical and measurement means available (Newell and Simon 1975, p. 114)

Since Newell and Simon's Turing award lecture, it has been clear that computer science can be understood as an empirical science but of a special sort, and this is related to the nature of experiments in computing. Indeed, much current debate on the epistemological status of computer science concerns the problem of defining what kind of science it is (Tedre 2011, Tedre 2015) and, in particular, the nature of experiments in computer science (Schiaffonati and Verdicchio 2014), the nature, if any, of laws and theorems in computing (Hartmanis 1993; Rombach and Seelish 2008), and the methodological relation between computer science and software engineering (Gruner 2011).

Bibliography

Abramsky, Samson & Guy McCusker, 1995, "Games and Full Abstraction for the Lazy λ-Calculus", in D. Kozen (ed.), *Tenth Annual IEEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, pp. 234–43. doi:10.1109/LICS.1995.523259

Abramsky, Samson, Pasquale Malacaria, & Radha Jagadeesan, 1994, "Full Abstraction for PCF", in M. Hagiya & J.C. Mitchell (eds.), *Theoretical Aspects of Computer Software: International Symposium TACS '94*, Sendai, Japan, April 19–22, 1994, Springer-Verlag, pp. 1–15.

Abrial, Jean-Raymond, 1996, *The B-Book: Assigning Programs to Meanings*, Cambridge: Cambridge University Press.

Alama, Jesse, 2015, "The Lambda Calculus", *The Stanford Encyclopedia of Philosophy* (Spring 2015 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/spr2015/entries/lambda-calculus/>.

Allen, Robert J., 1997, *A Formal Approach to Software Architecture*, Ph.D. Thesis, Computer Science, Carnegie Mellon University. Issued as CMU Technical Report CMU-CS-97-144. Allen 1997 available on line

Ammann, Paul & Jeff Offutt, 2008, *Introduction to Software Testing*, Cambridge: Cambridge University Press.

Angius, Nicola, 2013a, "Abstraction and Idealization in the Formal Verification of Software", *Minds and Machines*, 23(2): 211–226. doi:10.1007/s11023-012-9289-8

–––, 2013b, "Model-Based Abductive Reasoning in Automated Software Testing", *Logic Journal of IGPL*, 21(6): 931–942. doi:10.1093/jigpal/jzt006

–––, 2014, "The Problem of Justification of Empirical Hypotheses in Software Testing", *Philosophy & Technology*, 27(3): 423–439. doi:10.1007/s13347-014-0159-6

Angius, N., & Primiero, G., 2018, "The logic of identity and copy for computational artefacts", *Journal of Logic and Computation*, 28(6): 1293–1322.

Angius, Nicola & Guglielmo Tamburrini, 2011, "Scientific Theories of Computational Systems in Model Checking", *Minds and Machines*, 21(2): 323–336. doi:10.1007/s11023-011-9231-5

–––, 2017, "Explaining engineered computing systems' behaviour: the role of abstraction and idealization", *Philosophy & Technology*, 30(2): 239–258.

Anscombe, G. E. M., 1963, *Intention*, second edition, Oxford: Blackwell.

Arkoudas, Konstantine & Selmer Bringsjord, 2007, "Computers, Justification, and Mathematical Knowledge", *Minds and Machines*, 17(2): 185–202. doi:10.1007/s11023-007-9063-5

Arif, R. Mori, E., and Primiero, G, 2018, "Validity and Correctness before the OS: the case of LEOI and LEOII", in Liesbeth de Mol, Giuseppe Primiero (eds.), *Reflections on Programmings Systems - Historical and Philosophical Aspects*, Philosophical Studies Series, Cham: Springer, pp. 15–47.

Ashenhurst, Robert L. (ed.), 1989, "Letters in the ACM Forum", *Communications of the ACM*, 32(3): 287. doi:10.1145/62065.315925

Baier, A., 1970, "Act and Intent", *Journal of Philosophy*, 67: 648–658.

Baier, Christel & Joost-Pieter Katoen, 2008, *Principles of Model Checking*, Cambridge, MA: The MIT Press.

Bass, Len, Paul C. Clements, & Rick Kazman, 2003 [1997], *Software Architecture in Practice*, second edition, Reading, MA: Addison-Wesley; first edition 1997; third edition, 2012.

Bechtel, William & Adele Abrahamsen, 2005, "Explanation: A Mechanist Alternative", *Studies in History and Philosophy of Science Part C: Studies in History and Philosophy of Biological and Biomedical Sciences*, 36(2): 421–441. doi:10.1016/j.shpsc.2005.03.010

Boghossian, Paul A., 1989, "The Rule-following Considerations", *Mind*, 98(392): 507–549. doi:10.1093/mind/XCVIII.392.507

Bourbaki, Nicolas, 1968, *Theory of Sets*, Ettore Majorana International Science Series, Paris: Hermann.

Bratman, M. E., 1987, *Intention, Plans, and Practical Reason*, Cambridge, MA: Harvard University Press.

Bridges, Douglas & Palmgren Erik, 2013, "Constructive Mathematics", *The Stanford Encyclopedia of Philosophy* (Winter 2013 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/win2013/entries/mathematics-constructive/>.

Brooks, Frederick P. Jr., 1995, *The Mythical Man Month: Essays on Software Engineering*, Anniversary Edition, Reading, MA: Addison-Wesley.

–––, 1996, "The Computer Scientist as Toolsmith II", *Communications of the ACM*, 39(3): 61–68. doi:10.1145/227234.227243

Burge, Tyler, 1998, "Computer Proof, Apriori Knowledge, and Other Minds", *Noûs*, 32(S12): 1–37. doi:10.1111/0029-4624.32.s12.1

Bynum, Terrell Ward, 2008, "Milestones in the History of Information and Computer Ethics", in Himma and Tavani 2008: 25–48. doi:10.1002/9780470281819.ch2

Callahan, John, Francis Schneider, & Steve Easterbrook, 1996, "Automated Software Testing Using Model-Checking", in *Proceeding Spin Workshop*, J.C. Gregoire, G.J. Holzmann and D. Peled (eds.), New Brunswick, NJ: Rutgers University, pp. 118–127.

Cardelli, Luca & Peter Wegner, 1985, "On Understanding Types, Data Abstraction, and Polymorphism", 17(4): 471–522. [Cardelli and Wegner 1985 available online]

Carnap, R., 1966, *Philosophical foundations of physics* (Vol. 966), New York: Basic Books.

Carrara, M., Gaio, S., and Soavi, M., 2014, "Artifact kinds, identity criteria, and logical adequacy", in M. Franssen, P. Kroes, T. Reydon and P. E. Vermaas (eds.), *Artefact Kinds: Ontology and The Human-made*

World, New York: Springer, pp. 85–101.

Chalmers, A. F., 1999, *What is this thing called Science?*, Maidenhead: Open University Press

Chalmers, David J., 1996, "Does a Rock Implement Every Finite-State Automaton?" *Synthese*, 108(3): 309–33. [D.J. Chalmers 1996 available online] doi:10.1007/BF00413692

Clarke, Edmund M. Jr., Orna Grumberg, & Doron A. Peled, 1999, *Model Checking*, Cambridge, MA: The MIT Press.

Colburn, Timothy R., 1999, "Software, Abstraction, and Ontology", *The Monist*, 82(1): 3–19. doi:10.5840/monist19998215

–––, 2000, *Philosophy and Computer Science*, Armonk, NY: M.E. Sharp.

Colburn, T. R., Fetzer, J. H. , and Rankin T. L., 1993, *Program Verification: Fundamental Issues in Computer Science*, Dordrecht: Kluwer Academic Publishers.

Colburn, Timothy & Gary Shute, 2007, "Abstraction in Computer Science", *Minds and Machines*, 17(2): 169–184. doi:10.1007/s11023-007-9061-7

Copeland, B. Jack, 1993, *Artificial Intelligence: A Philosophical Introduction*, San Francisco: John Wiley & Sons.

–––, 1996, "What is Computation?" *Synthese*, 108(3): 335–359. doi:10.1007/BF00413693

–––, 2015, "The Church-Turing Thesis", *The Stanford Encyclopedia of Philosophy* (Summer 2015 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/sum2015/entries/church-turing/>.

Copeland, B. Jack & Oron Shagrir, 2007, "Physical Computation: How General are Gandy's Principles for Mechanisms?" *Minds and Machines*, 17(2): 217–231. doi:10.1007/s11023-007-9058-2

–––, 2011, "Do Accelerating Turing Machines Compute the Uncomputable?" *Minds and Machines*, 21(2): 221–239. doi:10.1007/s11023-011-9238-y

Cummins, Robert, 1975, "Functional Analysis", *The Journal of Philosophy*, 72(20): 741–765. doi:10.2307/2024640

Davidson, D., 1963, "Actions, Reasons, and Causes," reprinted in *Essays on Actions and Events*, Oxford: Oxford University Press, 1980, pp. 3–20.

–––, 1978, "Intending", reprinted in *Essays on Actions and Events*, Oxford: Oxford University Press, 1980, pp. 83–102.

De Millo, Richard A., Richard J. Lipton, & Alan J. Perlis, 1979, "Social Processes and Proofs of Theorems and Programs", *Communications of the ACM*, 22(5): 271–281. doi:10.1145/359104.359106

Denning, Peter J., 2005, "Is Computer Science Science?", *Communications of the ACM*, 48(4): 27–31. doi:10.1145/1053291.1053309

–––, 2007, "Computing is a Natural Science", *Communications of the ACM*, 50(7): 13–18. doi:10.1145/1272516.1272529

Denning, Peter J., Edward A. Feigenbaum, Paul Gilmore, Anthony C. Hearn, Robert W. Ritchie, & Joseph F. Traub, 1981, "A Discipline in Crisis", *Communications of the ACM*, 24(6): 370–374. doi:10.1145/358669.358682

Devlin, Keith, 1994, *Mathematics: The Science of Patterns: The Search for Order in Life, Mind, and the Universe*, New York: Henry Holt.

Dijkstra, Edsger W., 1970, *Notes on Structured Programming*, T.H.-Report 70-WSK-03, Mathematics Technological University Eindhoven, The Netherlands. [Dijkstra 1970 available online]

–––, 1974, "Programming as a Discipline of Mathematical Nature", *American Mathematical Monthly*, 81(6): 608–612. [Dijkstra 1974 available online

Distributed Software Engineering, 1997, *The Darwin Language*, Department of Computing, Imperial College of Science, Technology and Medicine, London. [Darwin language 1997 available online]

Duhem, P., 1954, *The Aim and Structure of Physical Theory*, Princeton: Princeton University Press.

Duijf, H., Broersen, J., and Meyer, J. J. C., 2019, "Conflicting intentions: rectifying the consistency requirements", *Philosophical Studies*, 176(4): 1097–1118.

Dummett, Michael A.E., 2006, *Thought and Reality*, Oxford: Oxford University Press.

Duncan, William, 2011, "Using Ontological Dependence to Distinguish between Hardware and Software", *Proceedings of the Society for the Study of Artificial Intelligence and Simulation of Behavior Conference: Computing and Philosophy*, University of York, York, UK. [Duncan 2011 available online (zip file)]

–––, 2017, "Ontological Distinctions between Hardware and Software", *Applied Ontology*, 12(1): 5–32.

Eden, Amnon H., 2007, "Three Paradigms of Computer Science", *Minds and Machines*, 17(2): 135–167. doi:10.1007/s11023-007-9060-8

Egan, Frances, 1992, "Individualism, Computation, and Perceptual Content", *Mind*, 101(403): 443–59. doi:10.1093/mind/101.403.443

Edgar, Stacey L., 2003 [1997], *Morality and Machines: Perspectives on Computer Ethics*, Sudbury, MA: Jones & Bartlett Learning.

Ferrero, L., 2017, "Intending, Acting, and Doing," *Philosophical Explorations*, 20 (Supplement 2): 13–39.

Fernández, Maribel, 2004, *Programming Languages and Operational Semantics: An Introduction*, London: King's College Publications.

Fetzer, James H., 1988, "Program Verification: The Very Idea", *Communications of the ACM*, 31(9): 1048–1063. doi:10.1145/48529.48530

–––, 1990, *Artificial Intelligence: Its Scope and Limits*, Dordrecht: Springer Netherlands.

Feynman, Richard P., 1984–1986, *Feynman Lectures on Computation*, Cambridge, MA: Westview Press, 2000.

Flanagan, Mary, Daniel C. Howe, & Helen Nissenbaum, 2008, "Embodying Values in Technology: Theory and Practice", in *Information Technology and Moral Philosophy*, Jeroen van den Hoven and John Weckert (eds.), Cambridge: Cambridge University Press, pp. 322–353.

Floridi, Luciano, 2008, "The Method of Levels of Abstraction", *Minds and Machines*, 18(3): 303–329. doi:10.1007/s11023-008-9113-7

Floridi, Luciano, Nir Fresco, & Giuseppe Primiero, 2015, "On Malfunctioning Software", *Synthese*, 192(4): 1199–1220. doi:10.1007/s11229-014-0610-3

Floyd, Robert W., 1979, "The Paradigms of Programming", *Communications of the ACM*, 22(8): 455–460. doi:10.1145/1283920.1283934

Fowler, Martin, 2003, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edition, Reading, MA: Addison-Wesley.

Franssen, Maarten, Gert-Jan Lokhorst, & Ibio van de Poel, 2013, "Philosophy of Technology", *The Stanford Encyclopedia of Philosophy* (Winter 2013 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/win2013/entries/technology/>.

Frege, Gottlob, 1914, "Letter to Jourdain", reprinted in Frege 1980: 78–80.

–––, 1980, Gottlob Frege: *Philosophical and Mathematical Correspondence*, G. Gabriel, H. Hermes, F. Kambartel, C. Thiel, and A. Veraart (eds.), Oxford: Blackwell Publishers.

Fresco, Nir & Giuseppe Primiero, 2013, "Miscomputation", *Philosophy & Technology*, 26(3): 253–272. doi:10.1007/s13347-013-0112-0

Friedman, Batya & Helen Nissenbaum, 1996, "Bias in Computer Systems", *ACM Transactions on Information Systems (TOIS)*, 14(3): 330–347. doi:10.1145/230538.230561

Frigg, Roman & Stephan Hartmann, 2012, "Models in Science", *The Stanford Encyclopedia of Philosophy* (Fall 2012 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/fall2012/entries/models-science/>.

Gagliardi, Francesco, 2007, "Epistemological Justification of Test Driven Development in Agile Processes", *Agile Processes in Software Engineering and Extreme Programming: Proceedings of the 8th International Conference, XP 2007*, Como, Italy, June 18–22, 2007, Berlin: Springer Berlin Heidelberg, pp. 253–256. doi:10.1007/978-3-540-73101-6_48

Gamma, Erich, Richard Helm, Ralph Johnson, & John Vlissides, 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley.

Glennan, Stuart S., 1996, "Mechanisms and the Nature of Causation", *Erkenntnis*, 44(1): 49–71. doi:10.1007/BF00172853

Glüer, Kathrin & Åsa Wikforss, 2015, "The Normativity of Meaning and Content", *The Stanford Encyclopedia of Philosophy* (Summer 2015 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/sum2015/entries/meaning-normativity/>.

Goguen, Joseph A. & Rod M. Burstall, 1985, "Institutions: Abstract Model Theory for Computer Science", *Report CSLI-85-30*, Center for the Study of Language and Information at Stanford University.

–––, 1992, "Institutions: Abstract Model Theory for Specification and Programming", *Journal of the ACM (JACM)*, 39(1): 95–146. doi:10.1145/147508.147524

Gordon, Michael J.C., 1979, *The Denotational Description of Programming Languages*, New York: Springer-Verlag.

Gotterbarn, Donald, 1991, "Computer Ethics: Responsibility Regained", *National Forum: The Phi Beta Kappa Journal*, 71(3): 26–31.

–––, 2001, "Informatics and Professional Responsibility", *Science and Engineering Ethics*, 7(2): 221–230. doi:10.1007/s11948-001-0043-5

Gotterbarn, Donald, Keith Miller, & Simon Rogerson, 1997, "Software Engineering Code of Ethics", *Information Society*, 40(11): 110–118. doi:10.1145/265684.265699

Gotterbarn, Donald & Keith W. Miller, 2009, "The Public is the Priority: Making Decisions Using the Software Engineering Code of Ethics", *IEEE Computer*, 42(6): 66–73. doi:10.1109/MC.2009.204

Gruner, Stefan, 2011, "Problems for a Philosophy of Software Engineering", *Minds and Machines*, 21(2): 275–299. doi:10.1007/s11023-011-9234-2

Gunter, Carl A., 1992, *Semantics of Programming Languages: Structures and Techniques*, Cambridge, MA: MIT Press.

Gupta, Anil, 2014, "Definitions", *The Stanford Encyclopedia of Philosophy* (Fall 2014 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/fall2014/entries/definitions/>.

Gurevich, Y., 2000, "Sequential Abstract-State Machines Capture Sequential Algorithms", *ACM Transactions on Computational Logic (TOCL)*, 1(1): 77-111.

–––, 2012, "What is an algorithm?", in *International conference on current trends in theory and practice of computer science*, Heidelberg, Berlin: Springer, pp. 31–42.

Hacking, I., 2014, *Why is there a Philosophy of Mathematics at all?*, Cambridge: Cambridge University Press.

Hagar, Amit, 2007, "Quantum Algorithms: Philosophical Lessons", *Minds and Machines*, 17(2): 233–247. doi:10.1007/s11023-007-9057-3

Hale, Bob, 1987, *Abstract Objects*, Oxford: Basil Blackwell.

Hales, Thomas C., 2008, "Formal Proof", *Notices of the American Mathematical Society*, 55(11): 1370–1380.

Hankin, Chris, 2004, *An Introduction to Lambda Calculi for Computer Scientists*, London: King's College Publications.

Harrison, John, 2008, "Formal Proof—Theory and Practice", *Notices of the American Mathematical Society*, 55(11): 1395–1406.

Hartmanis, Juris, 1981, "Nature of Computer Science and Its Paradigms", pp. 353–354 (in Section 1) of "Quo Vadimus: Computer Science in a Decade", J.F. Traub (ed.), *Communications of the ACM*, 24(6): 351–369. doi:10.1145/358669.358677

–––, 1993, "Some Observations About the Nature of Computer Science", in *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer Berlin Heidelberg, pp. 1–12. doi:10.1007/3-540-57529-4_39

Hayes, P. J., 1997, "What is a Computer?", *The Monist*, 80(3): 389–404.

Hempel, C. G., 1970, "On the 'standard conception' of scientific theories", *Minnesota Studies in the Philosophy of Science*, 4: 142–163.

Henson, Martin C., 1987, *Elements of Functional Programming*, Oxford: Blackwell.

Hilbert, David, 1931, "The Grounding of Elementary Number Theory", reprinted in P. Mancosu (ed.), 1998, *From Brouwer to Hilbert: the Debate on the Foundations of Mathematics in the 1920s*, New York: Oxford University Press, pp. 266–273.

Hilbert, David & Wilhelm Ackermann, 1928, *Grundzüge Der Theoretischen Logik*, translated as *Principles of Mathematical Logic*, Lewis M. Hammond, George G. Leckie, and F. Steinhardt (trans.), New York: Chelsea, 1950.

Hill, R.K., 2016, "What an algorithm is", *Philosophy & Technology*, 29(1): 35–59.

–––, 2018, "Elegance in Software", in Liesbeth de Mol, Giuseppe Primiero (eds.), *Reflections on Programmings Systems - Historical and Philosophical Aspects (Philosophical Studies Series)*, Cham: Springer, pp. 273–286.

Hoare, C.A.R., 1969, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, 12(10): 576–580. doi:10.1145/363235.363259

–––, 1973, "Notes on Data Structuring", in O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (eds.), *Structured Programming*, London: Academic Press, pp. 83–174.

–––, 1981, "The Emperor's Old Clothes", *Communications of the ACM*, 24(2): 75–83. doi:10.1145/1283920.1283936

–––, 1985, *Communicating Sequential Processes*, Englewood Cliffs, NJ: Prentice Hall. [Hoare 1985 available online]

–––, 1986, *The Mathematics of Programming: An Inaugural Lecture Delivered Before the University of Oxford on Oct. 17, 1985*, Oxford: Oxford University Press.

Hodges, Andrews, 2011, "Alan Turing", *The Stanford Encyclopedia of Philosophy* (Summer 2011 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/sum2011/entries/turing/>.

Hodges, Wilfrid, 2013, "Model Theory", *The Stanford Encyclopedia of Philosophy* (Fall 2013 Edition), Edward N. Zalta (ed.), forthcoming URL = <https://plato.stanford.edu/archives/fall2013/entries/model-theory/>.

Hopcroft, John E. & Jeffrey D. Ullman, 1969, *Formal Languages and their Relation to Automata*, Reading, MA: Addison-Wesley.

Hughes, Justin, 1988, "The Philosophy of Intellectual Property", *Georgetown Law Journal*, 77: 287.

Irmak, Nurbay, 2012, "Software is an Abstract Artifact", *Grazer Philosophische Studien*, 86(1): 55–72.

Johnson, Christopher W., 2006, "What are Emergent Properties and How Do They Affect the Engineering of Complex Systems", *Reliability Engineering and System Safety*, 91(12): 1475–1481. [Johnson 2006 available online]

Johnson-Laird, P. N., 1988, *The Computer and the Mind: An Introduction to Cognitive Science*, Cambridge, MA: Harvard University Press.

Jones, Cliff B., 1990 [1986], *Systematic Software Development Using VDM*, second edition, Englewood Cliffs, NJ: Prentice Hall. [Jones 1990 available online]

Kimppa, Kai, 2005, "Intellectual Property Rights in Software—Justifiable from a Liberalist Position? Free Software Foundation's Position in Comparison to John Locke's Concept of Property", in R.A. Spinello & H.T. Tavani (eds.), *Intellectual Property Rights in a Networked World: Theory and Practice*, Hershey, PA: Idea, pp. 67–82.

Kinsella, N. Stephan, 2001, "Against Intellectual Property", *Journal of Libertarian Studies*, 15(2): 1–53.

Kleene, S. C., 1967, *Mathematical Logic*, New York: Wiley.

Knuth, D. E., 1973, *The Art of Computer Programming*, second edition, Reading, MA: Addison-Wesley.

–––, 1974a, "Computer Programming as an Art", *Communications of the ACM*, 17(12): 667–673. doi:10.1145/1283920.1283929

–––, 1974b, "Computer Science and Its Relation to Mathematics", *The American Mathematical Monthly*, 81(4): 323–343.

–––, 1977, "Algorithms", *Scientifc American*, 236(4): 63–80.

Kripke, Saul, 1982, *Wittgenstein on Rules and Private Language*, Cambridge, MA: Harvard University Press.

Kroes, Peter, 2010, "Engineering and the Dual Nature of Technical Artefacts", *Cambridge Journal of Economics*, 34(1): 51–62. doi:10.1093/cje/bep019

–––, 2012, *Technical Artefacts: Creations of Mind and Matter: A Philosophy of Engineering Design*, Dordrecht: Springer.

Kroes, Peter & Anthonie Meijers, 2006, "The Dual Nature of Technical Artefacts", *Studies in History and Philosophy of Science*, 37(1): 1–4. doi:10.1016/j.shpsa.2005.12.001

Kröger, Fred & Stephan Merz, 2008, *Temporal Logics and State Systems*, Berlin: Springer.

Ladd, John, 1988, "Computers and Moral Responsibility: a Framework for An Ethical Analysis", in Carol C. Gould, (ed.), *The Information Web: Ethical & Social Implications of Computer Networking*, Boulder, CO: Westview Press, pp. 207–228.

Landin, P.J., 1964, "The Mechanical Evaluation of Expressions", *The Computer Journal*, 6(4): 308–320. doi:10.1093/comjnl/6.4.308

Littlewood, Bev & Lorenzo Strigini, 2000, "Software Reliability and Dependability: a Roadmap", *ICSE '00 Proceedings of the Conference on the Future of Software Engineering*, pp. 175–188. doi:10.1145/336512.336551

Locke, John, 1690, *The Second Treatise of Government*. [Locke 1690 available online]

Loewenheim, Ulrich, 1989, "Legal Protection for Computer Programs in West Germany", *Berkeley Technology Law Journal*, 4(2): 187–215. [Loewenheim 1989 available online] doi:10.15779/Z38Q67F

Long, Roderick T., 1995, "The Libertarian Case Against Intellectual Property Rights", *Formulations*, Autumn, Free Nation Foundation.

Loui, Michael C. & Keith W. Miller, 2008, "Ethics and Professional Responsibility in Computing", *Wiley Encyclopedia of Computer Science and Engineering*, Benjamin Wah (ed.), John Wiley & Sons. [Loui and Miller 2008 available online]

Lowe, E. J., 1998, *The Possibility of Metaphysics: Substance, Identity, and Time*, Oxford: Clarendon Press.

Luckham, David C., 1998, "Rapide: A Language and Toolset for Causal Event Modeling of Distributed System Architectures", in Y. Masunaga, T. Katayama, and M. Tsukamoto (eds.), *Worldwide Computing and its Applications*, WWCA'98, Berlin: Springer, pp. 88–96. doi:10.1007/3-540-64216-1_42

Machamer, Peter K., Lindley Darden, & Carl F. Craver, 2000, "Thinking About Mechanisms", *Philosophy of Science*, 67(1): 1–25. doi:10.1086/392759

Magee, Jeff, Naranker Dulay, Susan Eisenbach, & Jeff Kramer, 1995, "Specifying Distributed Software Architectures", *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, Berlin: Springer-Verlag, pp. 137–153.

Markov, A., 1954, "Theory of algorithms", Tr. Mat. Inst. Steklov 42, pp. 1–14. trans. by Edwin Hewitt in *American Mathematical Society Translations*, Series 2, Vol. 15 (1960).

Martin-Löf, Per, 1982, "Constructive Mathematics and Computer Programming", in *Logic, Methodology and Philosophy of Science* (Volume VI: 1979), Amsterdam: North-Holland, pp. 153–175.

McGettrick, Andrew, 1980, *The Definition of Programming Languages*, Cambridge: Cambridge University Press.

McLaughlin, Peter, 2001, *What Functions Explain: Functional Explanation and Self-Reproducing Systems*, Cambridge: Cambridge University Press.

Meijers, A.W.M., 2001, "The Relational Ontology of Technical Artifacts", in P.A. Kroes and A.W.M. Meijers (eds.), *The Empirical Turn in the Philosophy of Technology*, Amsterdam: Elsevier, pp. 81–96.

Mitchelmore, Michael & Paul White, 2004, "Abstraction in Mathematics and Mathematics Learning", in M.J. Høines and A.B. Fuglestad (eds.), *Proceedings of the 28th Conference of the International Group for the Psychology of Mathematics Education (Volume 3)*, Bergen: Programm Committee, pp. 329–336. [Mitchelmore and White 2004 available online]

Miller, Alexander & Crispin Wright (eds), 2002, *Rule Following and Meaning*, Montreal/Ithaca: McGill-Queen's University Press.

Milne, Robert & Christopher Strachey, 1976, *A Theory of Programming Language Semantics*, London: Chapman and Hall.

Milner, R., 1971, "An algebraic definition of simulation between programs", Technical Report, *CS-205*, pp. 481–489, Department of Computer Science, Stanford University.

Mitchell, John C., 2003, *Concepts in Programming Languages*, Cambridge: Cambridge University Press.

Monin, Jean François, 2003, *Understanding Formal Methods*, Michael G. Hinchey (ed.), London: Springer (this is Monin's translation of his own Introduction aux Méthodes Formelles, Hermes, 1996, first edition; 2000, second edition), doi:10.1007/978-1-4471-0043-0

Mooers, Calvin N., 1975, "Computer Software and Copyright", *ACM Computing Surveys*, 7(1): 45–72. doi:10.1145/356643.356647

Moor, James H., 1978, "Three Myths of Computer Science", *The British Journal for the Philosophy of Science*, 29(3): 213–222.

Morgan, C., 1994, *Programming From Specifications*, Englewood Cliffs: Prentice Hall. [Morgan 1994 available online]

Moschovakis, Y. N., 2001, "What is an algorithm?", in *Mathematics Unlimited—2001 and Beyond*, Heidelberg, Berlin: Springer, pp. 919–936.

Naur, P., 1985, "Programming as theory building", *Microprocessing and microprogramming*, 15(5): 253–261.

Newell, A., and Simon, H. A., 1961, "Computer simulation of human thinking" *Science*, 134(3495): 2011–2017.

–––– 1972, *Human Problem Solving*, Englewood Cliffs, NJ: Prentice-Hall.

–––, 1976, "Computer Science as Empirical Inquiry: Symbols and Search", *Communications of the ACM*, 19(3): 113–126. doi:10.1145/1283920.1283930

Newell, Allen, Alan J. Perlis, & Herbert A. Simon, 1967, "Computer Science", *Science*, 157(3795): 1373–1374. doi:10.1126/science.157.3795.1373-b

Nissenbaum, Helen, 1998, "Values in the Design of Computer Systems", *Computers and Society*, 28(1): 38–39.

Northover, Mandy, Derrick G. Kourie, Andrew Boake, Stefan Gruner, & Alan Northover, 2008, "Towards a Philosophy of Software Development: 40 Years After the Birth of Software Engineering", *Journal for General Philosophy of Science*, 39(1): 85–113. doi:10.1007/s10838-008-9068-7

Pears, David Francis, 2006, *Paradox and Platitude in Wittgenstein's Philosophy*, Oxford: Oxford University Press. doi:10.1093/acprof:oso/9780199247707.001.0001

Piccinini, Gualtiero, 2007, "Computing Mechanisms", *Philosophy of Science*, 74(4): 501–526. doi:10.1086/522851

–––, 2008, "Computation without Representation", *Philosophical Studies*, 137(2): 206–241. [Piccinini 2008 available online] doi:10.1007/s11098-005-5385-4

–––, 2008, "Computers", *Pacific Philosophical Quarterly*, 89: 32–73.

–––, 2015, *Physical Computation: A Mechanistic Account*, Oxford: Oxford University Press. doi:10.1093/acprof:oso/9780199658855.001.0001

Piccinini, Gualtiero & Carl Craver, 2011, "Integrating Psychology and Neuroscience: Functional Analyses as Mechanism Sketches", *Synthese*, 183(3): 283–311. doi:10.1007/s11229-011-9898-4

Popper, Karl R., 1959, *The Logic of Scientific Discovery*, London: Hutchinson.

Primiero, G., 2016, "Information in the philosophy of computer science", in Floridi L. (ed.), *The Routledge Handbook of Philosophy of Information*, London: Routledge, pp. 90–106.

–––, 2020, *On the Foundations of Computing*. New York: Oxford University Press.

Primiero, G., D.F. Solheim & J.M. Spring, 2019 "On Malfunction, Mechanisms and Malware Classification", *Philos. Technol.* 32: 339–362. https://doi.org/10.1007/s13347-018-0334-2

Pylyshyn, Z. W., 1984, *Computation and Cognition: Towards a Foundation for Cognitive Science*, Cambridge, MA: MIT Press.

Pym, D., J.M. Spring, & P. O'Hearn, 2019, "Why Separation Logic Works", *Philosophy & Technology*, 32: 483–516.

Rapaport, William J., 1995, "Understanding Understanding: Syntactic Semantics and Computational Cognition", in Tomberlin (ed.), *Philosophical Perspectives*, Vol. 9: AI, Connectionism, and Philosophical Psychology, Atascadero, CA: Ridgeview, pp. 49–88. [Rapaport 1995 available online] doi:10.2307/2214212

–––, 1999, "Implementation Is Semantic Interpretation", *The Monist*, 82(1): 109–30. [Rapaport 1999 available online]

–––, 2005, "Implementation as Semantic Interpretation: Further Thoughts", *Journal of Experimental& Theoretical Artificial Intelligence*, 17(4): 385–417. [Rapaport 2005 available online]

–––, 2012, "Semiotic systems, computers, and the mind: how cognition could be computing", *International Journal of Signs and Semiotic Systems*, 2(1): 32–71

–––, 2018, "What is a Computer? A Survey", *Minds and Machines*, 28(3): 385–426.

Reynolds, J.C., 2002, "Separation Logic: a logic for shared mutable data structures", in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, IEEE, pp. 55–74.

Rombach, Dieter & Frank Seelisch, 2008, "Formalisms in Software Engineering: Myths Versus Empirical Facts", in *Balancing Agility and Formalism in Software Engineering*, Springer Berlin Heidelberg, pp. 13–25. doi:10.1007/978-3-540-85279-7_2

Rosenberg, A., 2012, *The Philosophy of Science*, London: Routledge.

Ryle G., 1949 [2009], *The Concept of Mind*, Abingdon: Routledge

Schiaffonati, Viola, 2015, "Stretching the Traditional Notion of Experiment in Computing: Explorative Experiments", *Science and Engineering Ethics*, 22(3): 1–19. doi:10.1007/s11948-015-9655-z

Schiaffonati, Viola & Mario Verdicchio, 2014, "Computing and Experiments", *Philosophy & Technology*, 27(3): 359–376. doi:10.1007/s13347-013-0126-7

Searle, J. R., 1990, "Is the brain a digital computer?" *Proceedings and Addresses of the American Philosophical Association*, 64(3): 21–37.

Searle, John R., 1995, *The Construction of Social Reality*, New York: Free Press.

Setiya, K., "Intention", *The Stanford Encyclopedia of Philosophy* (Fall 2018 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/fall2018/entries/intention/>.

Shanker, S.G., 1987, "Wittgenstein versus Turing on the Nature of Church's Thesis", *Notre Dame Journal of Formal Logic*, 28(4): 615–649. [Shanker 1987 available online] doi:10.1305/ndjfl/1093637650

Shavell, Steven & Tanguy van Ypersele, 2001, "Rewards Versus Intellectual Property Rights", *Journal of Law and Economics*, 44: 525–547

Skemp, Richard R., 1987, *The Psychology of Learning Mathematics*, Hillsdale, NJ: Lawrence Erlbaum Associates.

Smith, Brian Cantwell, 1985, "The Limits of Correctness in Computers", *ACM SIGCAS Computers and Society*, 14–15(1–4): 18–26. doi:10.1145/379486.379512

Snelting, Gregor, 1998, "Paul Feyerabend and Software Technology", *Software Tools for Technology Transfer*, 2(1): 1–5. doi:10.1007/s100090050013

Sommerville, Ian, 2016 [1982], *Software Engineering*, Reading, MA: Addison-Wesley; first edition, 1982.

Sprevak, M., 2010, "Computation, individuation, and the received view on representation", *Studies in History and Philosophy of Science*, 41(3): 260–270.

–––, 2012, "Three Challenges to Chalmers on Computational Implementation", *Journal of Cognitive Science*, 13(2): 107–143.

Stoy, Joseph E., 1977, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, Cambridge, MA: MIT Press.

Strachey, Christopher, 2000, "Fundamental Concepts in Programming Languages", *Higher-Order and Symbolic Computation*, 13(1–2): 11–49. doi:10.1023/A:1010000313106

Suber, Peter, 1988, "What Is Software?" *Journal of Speculative Philosophy*, 2(2): 89–119. [Suber 1988 available online]

Summerville, I., 2012, *Software Engineering*, Reading, MA: Addison-Wesley; first edition, 1982.

Suppe, Frederick, 1989, *The Semantic Conception of Theories and Scientific Realism*, Chicago: University of Illinois Press.

Suppes, Patrick, 1960, "A Comparison of the Meaning and Uses of Models in Mathematics and the Empirical Sciences", *Synthese*, 12(2): 287–301. doi:10.1007/BF00485107

–––, 1969, "Models of Data", in *Studies in the Methodology and Foundations of Science*, Dordrecht: Springer Netherlands, pp. 24–35.

Technical Correspondence, Corporate, 1989, *Communications of the ACM*, 32(3): 374–381. Letters from James C. Pleasant, Lawrence Paulson/Avra Cohen/Michael Gordon, William Bevier/Michael Smith/William Young, Thomas Clune, Stephen Savitzky, James Fetzer. doi:10.1145/62065.315927

Tedre, Matti, 2011, "Computing as a Science: A Survey of Competing Viewpoints", *Minds and Machines*, 21(3): 361–387. doi:10.1007/s11023-011-9240-4

–––, 2015, *The Science of Computing: Shaping a Discipline*, Boca Raton: CRC Press, Taylor and Francis Group.

Tedre, Matti & Ekki Sutinen, 2008, "Three Traditions of Computing: What Educators Should Know", *Computer Science Education*, 18(3): 153–170. doi:10.1080/08993400802332332

Thagard, P., 1984, "Computer programs as psychological theories", *Mind, Language and Society*, Vienna: Conceptus-Studien, pp. 77–84.

Thomasson, Amie, 2007, "Artifacts and Human Concepts", in Eric Margolis and Stephen Laurence (eds.), *Creations of the Mind: Essays on Artifacts and Their Representations*, Oxford: Oxford University Press, pp. 52–73.

Thompson, Simon, 2011, *Haskell: The Craft of Functional Programming*, third edition, Reading, MA: Addison-Wesley; first edition, 1996.

Tichy, Walter F., 1998, "Should Computer Scientists Experiment More?", *IEEE Computer*, 31(5): 32–40. doi:10.1109/2.675631

Turing, A.M., 1936, "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society (Series 2)*, 42: 230–65. doi:10.1112/plms/s2-42.1.230

–––, 1950, "Computing Machinery and Intelligence", *Mind*, 59(236): 433–460. doi:10.1093/mind/LIX.236.433

Turner, Raymond, 2007, "Understanding Programming Languages", *Minds and Machines*, 17(2): 203–216. doi:10.1007/s11023-007-9062-6

–––, 2009a, *Computable Models*, Berlin: Springer. doi:10.1007/978-1-84882-052-4

–––, 2009b, "The Meaning of Programming Languages", *APA Newsletters*, 9(1): 2–7. (This APA Newsletter is available online; see the Other Internet Resources.)

–––, 2010, "Programming Languages as Mathematical Theories", in J. Vallverdú (ed.), *Thinking Machines and the Philosophy of Computer Science: Concepts and Principles*, Hershey, PA: IGI Global, pp. 66–82.

–––, 2011, "Specification", *Minds and Machines*, 21(2): 135–152. doi:10.1007/s11023-011-9239-x

–––, 2012, "Machines", in H. Zenil (ed.), *A Computable Universe: Understanding and Exploring Nature as Computation*, London: World Scientific Publishing Company/Imperial College Press, pp. 63–76.

–––, 2014, "Programming Languages as Technical Artefacts", *Philosophy and Technology*, 27(3): 377–397; first published online 2013. doi:10.1007/s13347–012–0098-z

–––, 2018, *Computational artifacts: Towards a philosophy of computer science*, Berlin Heidelberg: Springer.

Tymoczko, Thomas, 1979, "The Four Color Problem and Its Philosophical Significance", *The Journal of Philosophy*, 76(2): 57–83. doi:10.2307/2025976

–––, 1980, "Computers, Proofs and Mathematicians: A Philosophical Investigation of the Four-Color Proof", *Mathematics Magazine*, 53(3): 131–138.

Van Fraassen, Bas C., 1980, *The Scientific Image*, Oxford: Oxford University Press. doi:10.1093/0198244274.001.0001

–––, 1989, *Laws and Symmetry*, Oxford: Oxford University Press. doi:10.1093/0198248601.001.0001

Van Leeuwen, Jan (ed.), 1990, *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*, Amsterdam: Elsevier and Cambridge, MA: MIT Press.

Vardi, M., 2012, "What is an algorithm?", *Communications of the ACM*, 55(3): 5. doi:10.1145/2093548.2093549

Vermaas, Pieter E. & Wybo Houkes, 2003, "Ascribing Functions to Technical Artifacts: A Challenge to Etiological Accounts of Function", *British Journal of the Philosophy of Science*, 54: 261–289. [Vermaas and Houkes 2003 available online]

Vliet, Hans van, 2008, *Software Engineering: Principles and Practice*, 3rd edition, Hoboken, NJ: Wiley. (First edition, 1993)

von Neumann, J. (1945). "First draft report on the EDVAC", *IEEE Annals of the History of Computing*, 15(4): 27–75.

Wang, Hao, 1974, *From Mathematics to Philosophy*, London: Routledge, Kegan & Paul.

Wegner, Peter, 1976, "Research Paradigms in Computer Science", in *Proceedings of the 2nd international Conference on Software Engineering*, Los Alamitos, CA: IEEE Computer Society Press, pp. 322–330.

White, Graham, 2003, "The Philosophy of Computer Languages", in Luciano Floridi (ed.), *The Blackwell Guide to the Philosophy of Computing and Information*, Malden: Wiley-Blackwell, pp. 318–326. doi:10.1111/b.9780631229193.2003.00020.x

Wiener, Norbert, 1948, *Cybernetics: Control and Communication in the Animal and the Machine*, New York: Wiley & Sons.

–––, 1964, *God and Golem, Inc.: A Comment on Certain Points Where Cybernetics Impinges on Religion*, Cambridge, MA: MIT press.

Wittgenstein, Ludwig, 1953 [2001], *Philosophical Investigations*, translated by G.E.M. Anscombe, 3rd Edition, Oxford: Blackwell Publishing.

–––, 1956 [1978], *Remarks of the Foundations of Mathematics*, G.H. von Wright, R. Rhees, and G.E.M. Anscombe (eds.), translated by G.E.M. Anscombe, revised edition, Oxford: Basil Blackwell.

–––, 1939 [1975], *Wittgenstein's Lectures on the Foundations of Mathematics, Cambridge 1939*, C. Diamond (ed.), Cambridge: Cambridge University Press.

Woodcock, Jim & Jim Davies, 1996, *Using Z: Specification, Refinement, and Proof*, Englewood Cliffs, NJ: Prentice Hall.

Wright, Crispin 1983, *Frege's Conception of Numbers as Objects*, Aberdeen: Aberdeen University Press.

Academic Tools

How to cite this entry.

Preview the PDF version of this entry at the Friends of the SEP Society.

Look up topics and thinkers related to this entry at the Internet Philosophy Ontology Project (InPhO).

Enhanced bibliography for this entry at PhilPapers, with links to its database.

Other Internet Resources

- ACM (ed.), 2013, ACM Turing Award Lectures.
- APA Newsletter on Philosophy and Computers, 9(1): Fall 2009.
- Groklaw, 2012a, "What Does 'Software is Mathematics' Mean?" Part 1, by PoIR.
- Groklaw, 2012b, "What Does 'Software is Mathematics' Mean?" Part 2, by PoIR.
- Huss, Eric, 1997, The C Library Reference Guide, at Fortran 90+ (www.fortran-2000.com).
- Rapaport, William J., 2020, "Philosophy of Computer Science". DRAFT © 2004–2020 by William J. Rapaport. Available at Philosophy of Computer Science, manuscript.
- Smith, Barry, 2012, "Logic and Formal Ontology". A revised version of the paper which appeared in J. N. Mohanty and W. McKenna (eds), 1989, *Husserl's Phenomenology: A Textbook*, Lanham: University Press of America.
- Turner, Ray and Amon Eden, 2011, "The Philosophy of Computer Science", *Stanford Encyclopedia of Philosophy* (Winter 2011 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/win2011/entries/computer-science/>. [This was the previous entry on the philosophy of computer science in the *Stanford Encyclopedia of Philosophy*—see the version history.]
- Center for Philosophy of Computer Science

Related Entries

artificial intelligence: logic and | Church-Turing Thesis | computability and complexity | computation: in physical systems | computational complexity theory | information | information: semantic conceptions of | intention | mathematics, philosophy of | recursive functions | technology, philosophy of | Turing machines

Giuseppe Primiero <*giuseppe.primiero@unimi.it*>
Raymond Turner <*turnr@essex.ac.uk*>

Open access to the SEP is made possible by a world-wide funding initiative.
Please Read How You Can Help Keep the Encyclopedia Free