Central Themes and Open Questions in the Philosophy of Computer Science

Nicola Angius\* and John Symons°

\*Department of Cognitive Science,

University of Messina, Messina, Italy

nicola.angius@unime.it

°Department of Philosophy,

University of Kansas, Lawrence, KS

johnsymons@ku.edu

**Abstract**

This paper introduces the *Global Philosophy* symposium on Giuseppe Primiero’s book *On the Foundations of Computing* (2020). The collection gathers commentaries and responses of the author with the aim of engaging with some open questions in the philosophy of computer science. Firstly, this paper introduces the central themes addressed in Primiero’s book; secondly, it highlights some of the main critiques from commentators in order to, finally, pinpoint some conceptual challenges indicating future directions for the philosophy of computer science.

**Keywords:** Philosophy of Computer Science, foundations of computing, implementation, software correctness, miscomputation, unconventional computing.

1. **Introduction**

This *Global Philosophy* collection gathers commentaries and responses to Giuseppe Primiero’s book *On the Foundations of Computing* (2020). The book is an important milestone in the development of the philosophy of computer science (Angius *et al.* 2021): its content encompasses most of the issues that have been examined in the relevant literature to date, while simultaneously attempting to provide a foundational analysis of the discipline of computer science. We understand the philosophy of computer science to be a subdiscipline of the philosophy of science, focusing on computer science as a special science, in the same way that, for example, philosophy of biology or philosophy of chemistry treats their respective sciences. Computer science has its own set of characteristic ontological, methodological, and epistemological problems and these form the core focus of the philosophy of computer science. In his book, Primiero aims to address these central problems while recognizing that there are many open questions and, for this reason, it is useful to take the opportunity in this book symposium to engage with some of these open questions and to provide a venue for further discussion of this young field.

Philosophy of computer science is in a relatively early stage of development, initially this work has tended to blend elements from philosophy of mind, formal epistemology, logic, and general philosophy of science; more recently philosophers have paid attention to the history and socio-cultural context of computer science and this has enriched treatment of the conceptual issues considerably. Computer science itself is a living discipline, subject to continuous change as a result of technological innovation as well as market and cultural pressures. The contributions to this collection sketch some central themes and future directions for the philosophy of computer science. To that end, in our introduction, we will delve into the topics tackled in Primero's book (section 2), highlighting some of the critiques from commentators (section 3). Our goal is twofold: to paint a picture of the current state of the philosophy of computer science, and to pinpoint the conceptual challenges and questions on the horizon (section 4).

1. **Central issues in the philosophy of computer science**

Philosophy and computer science have been linked since the very beginning of the modern computing era. Early computability theory is intertwined with logical and philosophical investigations into the foundations of mathematics and the definition of an *effective procedure*,and basic philosophical questions around the nature of mind and rationality have always loomed in the background of even the most mundane engineering and software projects. As such, the philosophy of computer science is as old as computer science itself. Nevertheless, beginning in the 2000’s philosophy of computer science came to be recognized as an independent branch of philosophy of science. The first “Philosophy of Computer Science” entry in the *Stanford Encyclopedia of Philosophy* appeared in 2008 (Turner, Eden 2008), and this was preceded by papers that explicitly characterized their projects as part of the philosophy of computer science. These started to appear only in the early 2000s.[[1]](#footnote-1) By the 2020s a series of monographs on the philosophy of computer science were published. At this point, the sub-discipline was showing signs of maturity and acceptance in the academic community.[[2]](#footnote-2)

The philosophy of computer science has covered a list of topics so far that include the ontology of computational systems, the nature of algorithms, the relations between models and specifications, or between theories and programs, the notion of implementation, the verification problem, inductive and experimental methods in computer science, the notion of miscomputation and error, and the epistemological status of computer science. Primiero (2020) addresses all of them with an overarching purpose in mind, that of identifying the foundations of the science of computing.[[3]](#footnote-3)

Understanding the ontology of a computational system amounts to determining what makes the system what it is. An ontological account will usually explain the kind of being that these systems possess in terms of their components and their relationships. Computational systems have been traditionally understood via a *dual ontology*, consisting of an abstract level, the software, and a physical level, the hardware. The distinction between hardware and software, when understood as an ontological dichotomy, has been explicitly criticized since the work of Moor (1978). Nevertheless, the distinction features persistently in popular discussions and continues to capture the interest of both philosophers and computer scientists (see for instance Duncan 2011 and Irmak 2012). The main difficulty is the characterization of software as a pure abstract entity. Philosophers of computer science observe that software always comes encoded into a programming language corresponding to some physical realization in a memory device. Software is tethered to material reality, human interests, and practical resource trade-offs.

This issue is not treated in detail in Primiero (2020), but the ontology of computational systems adopted by the author influences all his foundational analysis. Instead of separating an abstract level from a physical, material, level, he relies on the notion of *Level of Abstraction* (LoA) initially advanced in (Floridi 2008) to define a computational system on the basis of a hierarchy of LoAs, identified in a functional specification level, a design specification level, an algorithm design level, an algorithm implementation level, and an algorithm execution level.[[4]](#footnote-4) Upper levels in the hierarchy are more abstract and functional, lower levels are more physical and procedural. This allows Primiero to avoid some of the difficulties of the software/hardware dichotomy.

Another ontological concern involves the nature of algorithms. The problem in its modern form arose while defining what an effective procedure is. Defining an effective procedure was, of course, a central feature of early 20th Century debates in the foundations of mathematics. The problem of the ontology of algorithms is still far from settled (Vardi 2012). While Knuth (1973) provided the classic starting point for this discussion, more recently the debate has been influenced by the axiomatic approach of Gurevich (2000), understanding algorithms in terms of abstract machines, or by the recursive equations approach defended by Moschovakis (2001). Others, including Rapaport (2023) and Hill (2016), argue that formal definitions are pointless in that non-formal properties are those making an algorithm an effective procedure and they thus try to provide informal accounts for algorithms. We regard this as a central open problem in the philosophy of computer science.

Primiero (2020) takes the LoA approach to provide a tripartite answer to the problem of defining what an algorithm is. At a functional specification level, algorithms can be understood as informal specifications, that is, as informal descriptions of a procedure. At a design specification level, an algorithm corresponds to a formal description in a given formal language of how to execute a procedure. To specify many important properties of algorithms, such as complexity classes, a lower-level definition of algorithm is needed. At the algorithm design level, algorithms can be understood as abstract machines implementing procedures and formally describing executions of programs implementing those abstract machines.

Careful attention to software specification has occupied philosophers from a variety of perspectives.[[5]](#footnote-5) Raymond Turner proposed an extensive work on the ontology of software specifications (see for instance 2011; 2018; 2020). His main contribution involves the recognition that specifications in software development should not be confused with computational models, even though they may well coincide. For instance, a state transition system can be both a model and a specification. If models and specifications can be put on a par at an ontological level, they differ at an epistemological and methodological level. When a state transition system is taken as a specification, it is aimed at *requiring* the behaviors of the specified program, in terms of the paths in the state transition system. When it is a model, paths are taken to *represent* actual behaviors of an implemented program. For Turner (2011) whereas specifications are *normative,* models are *descriptive*.

Primiero (2020) embraces Turner’s analysis on models and computation to embed it in his definition of *formal computational validity*. Formal validity is given at higher LoAs, namely at the design specification level and algorithm design level wherein formal verification can be in principle applied to prove whether an algorithm complies with the requested specifications. Both models and specifications are required to define formal computational validity. A program is said to formally validate a specification if and only if there is no model such that program expressions are true in the model but the specification is false. This is tantamount to saying that formal computational validity obtains in case models of the program comply with specifications, that is, when prescribed and observed program behaviors coincide.

Also, for the notion of *implementation* Primiero (2020) is able to incorporate the significant work carried out within the philosophy of computer science in his ontological framework. As we discuss in the next section, implementation is a very crucial notion that draws on debates far beyond the scope of the philosophy of computer science: in the philosophy of computing broadly conceived, it is the problem of defining *when* a system realizes a computation. As such it goes back to debates among philosophical functionalists in philosophy of mind and philosophy of biology.[[6]](#footnote-6) In the restricted context of computational artifacts, with respect to which one already knows that they realize a computation, it is the problem of determining *in virtue of what* an artifact implements a computation; the answer, in turn, is used to define when an implementation is a *correct* implementation.

Rapaport (1999; 2005b; see also 2023) defined implementation as a kind of *semantic interpretation* wherein an abstract, and syntactic, domain is interpreted, i.e. is given meaning, in a concrete, or semantic, domain, by means of an intermediary domain called the medium of interpretation, which can be either abstract or concrete. So, for instance, a *java* program is an implementation of an algorithm in case it provides meaning to the syntactic constructs of the algorithm, having the Java programming language as its medium of implementation.

Turner (2012; 2014; see also 2018) highlights that one thing that distinguishes implementation in computer science from implementation of non-engineered computational systems is that the relation of interpretation cannot be given between any two domains, otherwise one could not distinguish an incorrect from a correct implementation. In order to be able of doing this, one should consider implementation as the relation specification-artifact: a computational artifact is an implementation in that there is a specification that provides correctness jurisdiction over the artifact, that is, an artifact is an implementation, and a correct one, in case it realizes the computational paths prescribed by a specification.

Primiero (2020) extends the analyses of Rapaport (1999; 2005b) and Turner (2012; 2014) by arguing that the implementation relation should not be limited to the specification/program domains in that it implicitly embraces a dual ontology for computational systems. Rather, by considering all LoAs, the relation should be relaxed so as to hold between any two such levels. For instance, an algorithm can be considered an implementation of a specification, and an executable code an implementation of a high-level language program. Implementation is then defined as an *instantiation relation* that holds between any LoA and any other one upper in the LoA hierarchy. This latter requisite is justified by the fact that any LoA is to be considered a specification for any other lower LoA.

The proposed account of implementation allows Primiero (2020), among other things, to provide an innovative analysis of the phenomenon of *miscomputation*.[[7]](#footnote-7) Following previous work (Fresco and Primiero 2013; Floridi*et al.* 2015), he distinguishes three main families of miscomputations, namely, *conceptual, material,* and *performable* errors. The latter occur only at the algorithm execution level and amount to errors due to some faulty hardware implementation of a correct software. Conceptual errors, called *mistakes,* refer to the logical inconsistencies a developer may introduce at any LoA. A typical mistake at the design specification level is a logically inconsistent specification expressed in a formal language. Material errors, called *failures*, can occur at any LoA as well and amount to failing in successfully implementing a specification or intentional requirement. Here the common case is given by an algorithm which turns out to be incorrect with respect to its specification set. To these the author adds *slips*, which are both conceptual and material errors occurring at the algorithm implementation level depending on whether the code violates respectively syntactic or semantic rules of the implementing programming language.[[8]](#footnote-8)

Evaluating whether an implementation is a correct implementation or not is known as the *verification problem*. The debate on program verification is more tightly related to whether formal methods conceived in theoretical computer science are indeed able to prove program correctness. The two main protagonists in the classical debate on program verification have been Tony Hoare and James Fetzer. Hoare (1969) famously argued that computer science is an “exact science” in that the behaviors of a program can be derived by an axiomatic representation of the program itself, through a logic he introduced and hence known as Hoare’s logic. Accordingly, program correctness can be proved by means of mathematical reasoning. Fetzer (1988) equally famously argued that what one is proving here is not program correctness, but rather algorithm correctness. Programs contain implementation details such that their correctness also depends on the correctness of the physical structure of the implementation. Therefore, program verification also involves empirical reasoning.

Once again, the stratified ontology of computational systems by LoAs allows Primiero (2020) to address the program verification problem from a different perspective. Computational systems can be verified, following Hoare’s program, at the algorithm design level but not at the algorithm implementation level or algorithm execution level, as implicitly pointed out by Fetzer (1988). At the algorithm design LoA, algorithms *qua* abstract entities can be verified against their specification pursuing, as stated above, formal computational validity. The correctness of computational systems at lower LoAs involves different forms of computational validity. *Physical computational validity* obtains when the computational system is shown to be correct with any LoA defining it. Clearly, this form of correctness does not involve formal reasoning only, but also empirical one, achieved through the engineering approach of software testing. As Symons and Horner show, there is no general method for software verification in this sense (2020).

*Experimental computational validity* only considers experimental correctness, attained through simulations, here considered instances of scientific experiments. In particular, the algorithmic method consisting in addressing a computational problem by advancing an algorithm as a hypothetical solution to such a problem, implementing the algorithm and executing it to test whether the computational processes actually provides a correct solution to the problem, is compared by the author to the hypothetical-deductive method in science.[[9]](#footnote-9)

Following a well-established tradition (see for instance Eden 2007), the verification debate, and the broader issue of program correctness, is linked to the problem of defining the *epistemological status* of computer science. In a few words, the latter is the problem of determining whether computer science is to be acknowledged as a mathematical discipline, an engineering discipline, or a scientific discipline. Hoare (1969) has been among the first to argue, as recalled above, that computer science is an exact, mathematical, discipline; others, such as Brooks (1996), stressed the similarities of computer science with other branches of engineering, all involved in the development of artifacts and not of abstract entities; and the idea that computer science is the science of computers, involving experimental knowledge, finds in Newell, Perlis, and Simon (1967) one of its former formulations. Primiero (2020) argues that foundational inquiry into computer science requires three distinct approaches: a *mathematical foundation* which is based on formal computational validity, an *engineering foundation* provided by physical computational validity, and an *experimental foundation* obtained through experimental computational validity. Again, the three, compatible, foundations of computer science are rooted in his general approach to the LoAs defining computational systems, and on his view, different foundations are required at different levels.

Emerging discussions in the philosophy of computer science emphasize the trade-offs involved in computer science in the context of questions of verification, trustworthiness, and related epistemic questions. An excellent recent example of the tendency to take computers seriously as instruments in science is Alvarado (2023). Alvarado’s work emphasizes the importance of careful examination of the characteristics of the technology itself as part of understanding its influence on the practice of simulation in science. It is an example of the influence that philosophy of computer science is likely to have on general philosophy of science.

**3. Some open debates**

One of the most vexing issues in the philosophy of computer science is the status of computer science itself. Philosophers have taken a range of positions on whether computer science is a branch of engineering, an experimental science, a purely formal discipline, some blend of these, of something entirely different. This issue is related to another, challenging question, namely whether testing activities in software engineering should be regarded as scientific experiments (see on this Schiaffonati 2020; Schiaffonati and Verdicchio 2014; Tedre 2014). Angius (2022) in this book symposium goes even further by highlighting how a pure experimental foundation, not involving formal reasoning, is not achievable when software testing is considered.[[10]](#footnote-10)

The reason is that testing, *qua* pure empirical method, is affected by a known limitation famously formulated by Dijkstra: “program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra 1970, p.7). Testing is the practice of launching a program and observing its executions to evaluate whether they conform or not with behaviors prescribed by some specification (Ammann and Offutt 2016). Unless one is dealing with trivial programs, program inputs are potentially infinite (think about reactive systems interacting with users and environments). Consequently, a program cannot be tested using all inputs; inputs are to be chosen by looking at the specification under scrutiny. Angius (2022) highlights how formal verification methods are often used to isolate the inputs with which to test a program by algorithmically checking a model of the program against the required specification. This is taken by the author to be another instance of the hypothetical-deductive method in computer science beyond the algorithmic method. It follows that engineering and experimental methods are combined together in the actual computer science practice.

With respect to the mathematical foundation, and the related notion of formal computational validity, Primiero proposes a comprehensive historical reconstruction of the mathematical roots of computer science. So, for instance, while sketching out the debate on program verification, the author recalls that one may distinguish the *syntactic approach* of Hoare (1969) and Djkstra (1975), from *semantic approaches* and *resource-based analysis* (pp 88-100). The semantic approaches, as the name suggests, link the correctness of programs to the meaning associated to program instructions, either in terms of a *denotational semantics,* associating mathematical objects to program constructs, or in terms of *operational semantics*, associating operations of an abstract machine to the program.

Cardone (2023) questions the thesis that the denotational traditional approach of Scott and Strachey (1971) was initially conceived to provide correctness criteria for program verification. The author proposes an interesting historical reconstruction of the development of denotational semantics by Scott and Strachey to emphasize that the association of mathematical objects, namely domains, to program constructs was mainly conceived to define properties of those constructs without having a normative role. It is argued that the notions of implementation and miscomputation have to be reformulated accordingly.

The commentaries from AbuMusab (2023) and Curtis-Trudel (2023) focus on the fundamental notion of implementation, which is critical to the engineering foundation of the discipline. As highlighted above in Section Two, Primiero’s account of implementation as an instantiation relation was motivated by the need to provide a single notion of implementation capable of capturing the relation at any LoA, and not just between the design specification level and the algorithm implementation level. This is a deficiency Primiero identifies in both Rapaport’s (1999) and Turner’s (2012) accounts. The merit of Primero’s approach is indeed that of providing a *unique* notion of implementation applicable to any twoLoA. This may, for instance, account for the fact that a program, or an algorithm, is often said to fail to implement the client's intentions, despite being correct with respect to its specifications.

By contrast, AbuMusab (2023) takes the uniqueness character of Primiero’s notion of implementation to be its most compelling shortcoming. AbuMusab’s argument goes as follows. First, a notion of implementation should not be detached from a notion of correctness: what one is interested in here is correct implementation. A structural level implements a functional level when the former implements the latter correctly, otherwise it is not called an implementation. At the same time, Primiero (2020, ch 11), distinguishes as many as three notions of correctness. *Functional correctness*, occurring when a computational artifact displays the functionality expressed by its specifications; *procedural correctness*, holding when the artifact displays the functionality expressed, this time, by a (set of) algorithm(s); finally, *executional correctness* characterizes computational artifacts that are executed on a well-functioning physical devise, when no hardware failures take place. One may notice here that the three correctness notions refer to three distinct LoAs. AbuMusab (2023) has it that, since implementation comes along with correctness, one should define one distinct notion of implementation for each notion of correctness. Once a computational system is defined by many LoAs, different notions of implementations are needed to catch the instantiation relations between different LoAs.

According to Curtis-Trudel (2023) the proposed hierarchy of LoAs is too coarse and does not allow one to categorize all kinds of miscomputations. The author points out that Primiero’s ontology is based on the software engineer perspective, one involved in the development of a system by defining the desired functions, realizing those functions into an algorithm, instantiating the algorithm in a high-level language program and compiling the program into an executable code. The above mentioned tripartition among conceptual, material, and performable error is, according to Curtis-Trudel (2023), based on the software engineer perspective and does not take into account errors that come from lower, fine-grained, LoAs. Software engineers are not interested in the development of hardware systems which, nonetheless, are themselves artifacts and can be defined at proper LoAs, including the macroarchitecture level, the microarchitecture level, the logic design level, or the circuit design level.

Not only are the proposed LoAs insufficient, they are even unnecessary if taken as a whole. Curtis-Trudel (2023) points out that many computational artifacts do not go through all the LoAs; think about many special purpose systems, wherein programs are hard-wired directly as executable code, thereby not going through the algorithm design level and algorithm implementation level. And even more difficult is applying the LoA ontology to natural computing systems, such as the brain or cell systems. It is straightforward, for instance, that no natural computational system is characterized by what Primiero calls intention or functional specification level. The reason is that they are not engineered systems.

This latter one is also the main point of the last commentary by Dodig-Crnkovic (2023), who indeed complains of the lack of a fourth, important, foundation of computing, namely *natural science*. Computational logic, computability theory, and software engineering are not the only roots of computing, physics is an important pillar of the science of computing as well. The author refers here to Denning and Rosenbloom’s (2009) idea of computing as the fourth domain of science, together with natural, formal, and social sciences. As Curtis-Trudel (2023), Dodig-Crnkovic (2023) points out that the ontological and epistemological framework elaborated by Primiero (2020) does not apply to many natural computing phenomena, often going under the name of *unconventional computing,* covering many chemical and organic systems not following Turing’s model of computation (Adamatzky 2018). Dodig-Crnkovic remarks that Primero’s contribution is not a foundational analysis of computing but rather of conventional computing or of computational artifacts.

**4. Future directions for the philosophy of computer science**

For his response to the criticisms described in the previous section, the reader is referred to Primiero (2023) in this collection. Primiero regards these remarks as hints for the future direction of the philosophy of computer science. Primiero’s book provides a historical and foundational analysis of computer science as it developed from the early computing era to the present date. The ontological, epistemological, and methodological examinations brought forward are those that have been characteristic of computer science from its origins. Computer science broadly conceived is still changing and developing, and a question now arises as to what the philosophical analyses that urge to be addressed in the next future are.

In Primiero (2020) the science of computing is conceived as the academic discipline of computer science with its branches including computational logic, theoretical computer science, and software engineering. Dodig-Crnkovic (2023) emphasizes that computing is far more than this and there are many, especially natural, computing systems which ontology and epistemology differ from those of computing artifacts. Indeed, the present introduction preferred using the expression ‘computer science’ throughout the paper, instead of ‘computing’, precisely to be clear that the object of investigations here are software systems. And Curtis-Trudel (2023) asks why developing a foundational analysis for computational artifacts only and not one that also extends to natural computing systems.

One reason for this is certainly that the ontology of natural computing systems and of computational artifacts is different. This is one of the main arguments contained in Rymond Turner’s book (2018) on the philosophy of computer science. Both natural and artificial computing systems are defined by computable *functions* and physical *structures* that implement those functions. However, implementation differs dramatically. Natural computing systems obey a *causal theory of function* (Cummins 1975), according to which the physical structural properties of the system cause and fix the functions (typically as adaptations in an evolutionary process). By contrast, artifacts obey an *intentional theory of function (*McLaughlin 2001, Symons 2010), according to which functions are designed into artifacts by an agent.

Nonetheless, natural computing is becoming more and more a source of inspiration for engineered computational systems; DNA computing, systems biology, agent-based modeling, and machine learning being cases in point. So even though the philosophical analyses of physical computing on one hand (Piccinini and Maley 2021), and of computer science on the other, have been carried out independently, there may well be reasons to elaborate a unified foundational analysis of the two fields.

A further reason to do so is related to the resurgence and success of contemporary machine learning techniques, in both technology and science. *Deep learning* networks, in spite of seemingly simple improvements of classical neural networks (more hidden layers and a slightly modified back-propagation algorithm, to mention two) are surprisingly successful in many technological applications, including artificial vision, language generation, or autonomous systems (Plebe and Grasso 2019). Computational systems implementing deep learning models differ in their ontology and epistemology from traditionally developed computational systems as long as they do not involve all the LoAs presented in Primiero (2020). Functions are not defined at a functional specification level, formalized at a design specification level and implemented at the algorithm design level; functions rather arise from the model during training and depend much more on the training dataset than on the algorithm itself (Angius and Plebe 2023). It follows that a new foundational analysis is required for these systems.

Primiero (2023) states that this is particularly true for the experimental foundation of computing. Indeed, deep learning modeling is being used in many scientific and simulative contexts, such as physics (Monk 2018), meteorology (Espeholt *et al.* 2021), neuroscience (Caucheteux*et al.* 2023), or most impressively in biology with the recent achievements of the Alphafold team in the domain of protein folding (Jumper 2021). Traditional, equation-based, simulative science is characterized by the formulation of a dynamical system, the mathematical model, used to simulate some target empirical system. Simulation consists in the implementation and execution of a computational model approximating the differential equations involved in the mathematical model. Experimental computational validity is defined by analyzing the formal relations entertained by the target system, the mathematical model, the computational model, and the artifact carrying out the executions. Those relations include the simulation relations between mathematical and computational models, the verification relation between the mathematical model and the artifact, and the validation relation between the mathematical model and the target system. This logical-epistemological picture must be fundamentally rethought in the context of machine learning driven versions of software intensive science, wherein deep learning models based on transformer approaches need not be approximated, can be directly executed, and are not conceived as representations of the target simulated systems. There is a great deal of philosophical work to be done to understand how we should begin to evaluate these remarkable new scientific instruments.

Another epistemological challenge associated with AI systems powered by deep neural networks concerns the definition of their formal computational validity. The debate on program verification will reasonably enter a third stage of renewed interest, insofar as deep learning systems cannot be said to be verifiable, but for reasons that are far from those advanced by Fetzer (1988). Firstly, desired functions, as already stated, are not advanced at any functional or design specification level; secondly, deep learning models are known for being opaque and uninterpretable (Lipton 2018), thereby making algorithmic exploration not directly performable. Primiero (2023) in his replies suggests that one way formal verification may be applied here is through the constructions of formal, probabilistic, models of trained networks, with formal proofs or algorithmic exploration being applied to the surrogate model instead. He claims that in the case of AI systems, formal computational validity cannot be but reduced to experimental validity, an additional logical and philosophical challenge for the upcoming research.

A final remark about the proposed LoAs, together with its hierarchies of ontological domains and epistemological structures (see ch. 10 of Primiero 2020), is that they are conceived considering classical software development methods. The traditional waterfall model, especially in the context of formal development methods, aims indeed at developing a computational system starting from a functional specification level and going through the design specification, the algorithm, the algorithm implementation, and finally execution levels. The spiral model goes through all LoAs several times; and agile methods go only through to some of them, trying to find a balance between velocity of development and correctness of the implemented system (Sommerville 2016).

Programming activities have been changing as well in the last two decades; object-oriented programming languages favored a way of programming, informally called cowboy-programming (see Scott 2016), wherein companies demand the development of programs and applications to freelance programmers, who work without coordination, without a shared design or specification set, by putting together software objects to build the final code. Briefly speaking, in object-oriented programming languages an object is a set of procedures, called methods, of constants and of variables, proper of that object, that perform a given function and that can be “called” by the code containing the object. Needless to say, in cowboy-programming software is not verified, and if eventually tested, errors are managed by adding correcting code lines, instead of debugging the program.

This, and other unconventional ways of programming not going through the proposed LoAs,[[11]](#footnote-11) induce new kinds of errors and miscomputations not fully captured by the taxonomy proposed in Primiero (2020). Curtis-Trudel (2023) suggests that different error types may come from the relation among source code, macro- and micro-architecture of the implementing hardware; Primiero (2023) calls for new formal frameworks enabling to account for malware, adversarial attacks, or bias-based miscomputations.

**5. Conclusions**

There are many other aspects of Primero’s (2020) book that may be remarked upon, or challenged, beyond those discussed here. And there are undoubtedly going to be a range of issues that occupy the philosophy of computer science in the next few years which we have not considered. The aim of this book symposium is to make a first step in that direction and to help consolidate the sub-disciplinary identity of the philosophy of computer science. Our initial idea was that these goals are helpfully achieved by inviting philosophers to comment on Primiero’s book, insofar as it clearly reflects the core themes and problems of contemporary philosophy of computer science.

**References**

AbuMusab, S. (2023) Implementing a Computing System: A Pluralistic Approach. *Global Philosophy* 33(20).

Adamatzky, A. (2018) *Unconventional Computing. A Volume in the Encyclopedia of Complexity and Systems Science, Second Edition*. Encycloped. New York, NY, USA: Springer Nature 2018

Alvarado, R. (2023). *Simulating Science: Computer Simulations as Scientific Instruments* (Vol. 479). Springer Nature.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Angius, N. (2022). On the Experimental Foundation of Computing. *Axiomathes*3(32): 1221–1236

Angius, N., Primiero, G. and Turner R. (2021). The Philosophy of Computer Science. *The Stanford Encyclopedia of Philosophy* (Spring 2021 Edition), Edward N. Zalta (ed.), URL = <<https://plato.stanford.edu/archives/spr2021/entries/computer-science/>>.

Brooks FP Jr (1996) The computer scientist as toolsmith ii. *Commun ACM* 39(3):61–68

Cardone, F. (2023). On Giving Meanings to Programs. *Global Philosophy*, *33*(1), 13.

Caucheteux, C., Gramfort, A.,& King, J. R. (2023). Evidence of a predictive coding hierarchy in the human brain listening to speech. *Nature human behaviour*, 7(3), 430-441.

Colburn, T. (2000), *Philosophy and Computer Science*, Armonk, NY: M.E. Sharp

Cummins, R. (1975). Functional analysis. Journal of Philosophy, 72, 741–764.

Curtis-Trudel, A. (2023) Primiero on Physical Computation. *glob. Philosophy* 33(2).

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

Denning, P., Rosenbloom, P. (2009) The fourth great domain of science. *ACM Commun*52(9):27–29

Dijkstra, E. W. (1975). Correctness concerns and, among other things, why they are resented. *ACM SIGPLAN Notices*, *10*(6), 546.

Dodig-Crnkovic, G. (2023) On the Foundations of Computing. Computing as the Fourth Great Domain of Science. *glob. Philosophy* 33(16).

Duncan, W. (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

Eden, A. H. (2007). Three paradigms of computer science. *Minds and machines*, *17*, 135-167.

Espeholt, L., Agrawal, S., Słonderby, C., Kumar, M., Heek, J., Bromberg, C., Gazen, C., Hickey, J., Bell, A., &Kalchbrenner, N. (2021). Skillful twelve hour precipitation forecasts using large context neural networks. *abs/2111.07472*.

Fetzer, J. H. (1988) Program Verification: The Very Idea. *Communications of the ACM*, 31(9): 1048–1063.

Floridi, L. (2008). The method of levels of abstraction. *Minds and machines*, *18*, 303-329.

Floridi, L., Fresco, N., &Primiero G. (2015) On Malfunctioning Software. *Synthese*, 192(4): 1199–1220.

Fresco, N., & Primiero G. (2013) Miscomputation. *Philosophy & Technology*, 26(3): 253–272.

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

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

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

Horner, J. K., & Symons, J. (2019). Understanding error rates in software engineering: Conceptual, empirical, and experimental approaches. *Philosophy & Technology*, *32*, 363-378.

Jumper, John, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool et al. (2021). "Highly accurate protein structure prediction with AlphaFold." *Nature* 596, no. 7873 583-589.

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

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

Lipton, Z. C. (2018). The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, 16(3), 31–57.

McLaughlin, P. (2001). *What functions explain: Functional explanation and self-reproducing systems*. Cambridge University Press.

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

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

Newell A, Perlis AJ, Simon HA (1967) Computer science. Science 157(3795):1373–1374

Monk, J. W. (2018). Deep learning as a parton shower. *Journal of High Energy Physics*, 2018, 21.

Piccinini, G., &Maley C. (2021) Computation in Physical Systems *The Stanford Encyclopedia of Philosophy* (Summer 2021 Edition), Edward N. Zalta (ed.)

Plebe, A., & Grasso, G. (2019). The unbearable shallow understanding of deep learning. *Minds and Machines*, *29*, 515-553.

Primiero, G. (2020). *On the foundations of computing*. Oxford University Press.

Primiero, G. (2023) On the Foundations of Computing: Limits and Open Issues. Forthcoming in *Global Philosophy*. <https://doi.org/10.1007/s10516-023-09688-w>

Putnam, H. (1987). Representation and Reality. MIT Press.

Rapaport, W. J. (1999) Implementation Is Semantic Interpretation. *The Monist*, 82(1): 109–30.

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

Rapaport, W. J. (2005) Implementation as Semantic Interpretation: Further Thoughts. *Journal of Experimental & Theoretical Artificial Intelligence*, 17(4): 385–417.

Rapaport, W. J. (2023). *Philosophy of Computer Science: An Introduction to the Issues and the Literature*. John Wiley & Sons.

Schiaffonati, V. (2020). Experimentation. In *The Routledge Handbook of the Philosophy of Engineering* (pp. 421-434). Routledge.

Schiaffonati, V., &Verdicchio, M. (2014). Computing and experiments: a methodological view on the debate on the scientific nature of computing. *Philosophy & Technology*, *27*, 359-376.

Scott, D. S., Strachey, C. (1971) Toward a mathematical semantics for computer languages. In: Fox J (ed) *Proceedings of the symposium on computers and automata*, Polytechnic Institute of Brooklyn Press, New York, pp 19–46

Scott, W. (2016) Cowboy coding. *Search software quality*. *TechTarget*. available at <https://www.techtarget.com/whatis/definition/software-development> Retrieved August 24, 2023.

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

Sommerville, I. (2016) *Software Engineering.* 10th Edition. Pearson.

Symons, J. (2010). The individuality of artifacts and organisms. *History and philosophy of the life sciences*, 233-246.

Symons, J., & Abumusab, S. (2021). Giuseppe Primiero's On the Foundations of Computing. *BJPS Review of Books*.

Symons, J & Horner, J. (2014) “Software Intensive Science” *Philosophy and Technology*, Volume 27, Issue 3, pp 461-477.

Symons, J & Horner, J. (2017) “Software Error as a Limit to Inquiry for Finite Agents: Challenges for the Post-human Scientist” In T. Powers, ed.  *Philosophy and Computing: Essays in Epistemology, Philosophy of Mind, Logic, and Ethics*.  Springer. 2017 pp. 85-99

Symons, J. & Horner, J. (2020) “Why there is no general solution to the problem of software verification” *Foundations of Science*. **25**, 541–557

Tedre, M. (2014). *The science of computing: shaping a discipline*. CRC Press.

Turing, A. M. 1950 Computing Machinery and Intelligence *Mind*, 59(236): 433–460.

Turner, R. Specification. *Minds & Machines* 21, 135–152 (2011).

Turner, R. (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.

Turner, R. (2014) Programming Languages as Technical Artefacts. *Philosophy and Technology*, 27(3): 377–397; first published online 2013.

Turner, R., & Turner, R. (2018). *Computational artifacts*. Springer Berlin Heidelberg.

Turner, R. (2020). Computational intention. *Studies in Logic, Grammar and Rhetoric*, 63(1), 19–30.

Turner, R., Eden, A. (2008) The Philosophy of Computer Science. *The Stanford Encyclopedia of Philosophy* (Winter 2008 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/win2008/entries/computer-science/>

Vallverdú, J. (Ed.). (2010). *Thinking machines and the philosophy of computer science: concepts and principles*. Hershey: IGI Global Group

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

1. An early example is Rapaport’s (2005a).However, there were clearly many papers concerned with the philosophical aspects of computer science that were published before. Notable cases are Moor (1978) or Fetzer (1988), just to give a couple of examples. But it was only in the 2000s that philosophers understood their engagement with computer science as part of a distinct subdiscipline. [↑](#footnote-ref-1)
2. See, for example, (Tedre 2014), followed by (Turner 2018), (Primero 2020), and (Rapaport 2023). Two previous books are Colburn (2000) and Vallverdú (2010), but they are more focused on the philosophy of artificial intelligence and only marginally on the philosophy of computer science. [↑](#footnote-ref-2)
3. This paper is not intended to be an introduction of Primero’s (2020) book; for an overview of the contents of the book, the interested reader may refer to Symons and Abumusab (2021) or to Angius (2022). [↑](#footnote-ref-3)
4. The reader interested in details should refer to Primero (2020) itself or to the useful recap in (Primiero 2023) included in this collection. [↑](#footnote-ref-4)
5. See Symons and Horner (2014; 2020) for detailed discussions of the role of software specification in the characterization of error. [↑](#footnote-ref-5)
6. For explicit applications to computer science see Searle (1990) and Putnam’s responses to the triviality objection in computational theories of mind (1987). [↑](#footnote-ref-6)
7. One should go back to some side-considerations of Turing (1950) to find a previous examination of the issue of miscomputation in computer science. [↑](#footnote-ref-7)
8. The issue of software error has been developed in detail by other authors, see for instance (Horner and Symons 2019; 2020). [↑](#footnote-ref-8)
9. Symons and Horner have a contrasting account of some of the implications of the verification problem for software intensive science (2014). [↑](#footnote-ref-9)
10. For the sake of precision, Primiero (2020) discusses simulative methods, rather than testing, when formulating his experimental foundation. Angius (2022) should be understood as an extension of Primiero’s analysis. [↑](#footnote-ref-10)
11. An additional novel programming paradigm comes from automatic code generation, for instance using generative AI. [↑](#footnote-ref-11)