Humanities & Social Sciences Communications

ARTICLE

https://doi.org/10.1057/s41599-020-00565-0 OPEN

Towards a hermeneutic definition of software

Luca M. Possati[1]✉

The paper intends to establish a comprehensive definition of software from a post-phenomenological and hermeneutic point of view. It intends to show the contribution of continental philosophy to the study of new technologies. In section "Introduction: why do we need a comprehensive definition of software?," I underline the need for a philosophical analysis that can highlight the multifaceted and paradoxical nature of software. In section "Engineering in written form: the five criteria," starting from some remarks on the history of programming languages, I define a list of minimal requirements (five criteria) that something needs to meet to be qualified as software. All these requirements share two essential features: the written form and the effectiveness, that is, the need to be executed by a physical machine. In section "Software as text: a hermeneutic model," I focus on software as form of writing. I develop this idea by using Ricoeur's hermeneutic model. I claim that software is a type of text. In section "The grammatology of microprocessor," I focus on the second aforementioned feature: the effectiveness of software. I claim that this effectiveness is based on the analogy between electric circuitries and Boolean logic. Software is a writing and re-writing process that implies an interpretation on two levels, epistemological and ontological.

[1] University of Porto, Porto, Portugal. ✉email: lupossati@gmail.com

Introduction: why do we need a comprehensive definition of software?

> […] the program is hard to understand from the outside: It's a black box (Voosen, 2017, p. 22).

> […] programming languages establish the programmer as a sovereign subject, for whom there is no difference between command given and command completed. As a lawgiver more powerful than a playwright or emperor, the programmer can "say" "let there be light," and there is light (Chun, 2013, p. 47).

The paper intends to establish a comprehensive definition of software and to show why simply regarding software as a mere algorithm, or a set of algorithms, is inadequate. I argue for the need for a philosophical analysis that can highlight the multifaceted and paradoxical nature of software. In my opinion, one of the tasks of philosophy is to shed light on the concepts we use. Therefore, the fundamental aim of this paper is to circumscribe some essential criteria for establishing what is software. This paper addresses in particular two fields: media studies and philosophy of technology.

Why do we need a comprehensive definition of software? In what way would a comprehensive definition of software be useful and to whom? Am I looking for a definition that is helpful also to technicians and software professionals (non-academic contexts)?

The first answer is that we need a comprehensive definition of software because software is pervasive, and it is the foundation of digital technologies and media widely used in society. As Marc Andreessen, the founder of Andreessen Horowitz (AH Capital Management), writes in 2011, "software is eating the world"[1], and this in economic, social, and cultural ways—everything today is software-mediated.

According to Chun (2013), a comprehensive definition of software is impossible. Software—she says—has so many different forms that it is impossible to reach a single definition. "Software is, or should be, a notoriously difficult concept. Historically unforeseen, barely a thing, software's ghostly presence produces and defies apprehension, allowing us to grasp the world through its ungraspable mediation" (Chun, 2013, p. 3). There is no single approach to software. Software is both a tangible and intangible thing, "a paradoxical combination of visibility and invisibility, of rational causality and profound ignorance" (Chun, 2013, p. 59). Hence, there is not an essence of software. But why? How can Chun argue that?

Chun's position is symptomatic of a broader issue within media studies. Two of the most important scholars (Kittler and Manovich) enter a paradoxical debate, in which they assert that there is no software and there is only software, respectively.

In a 1995 paper, Kittler has declared: "There is no software." If I opened my laptop, I would find neither programs nor numbers. Software is visible and invisible at the same time. Within the laptop, there are only electrical circuits and voltages: a certain arrangement of electrons. Indeed, what I could see if I looked inside my laptop would not tell me anything about how the machine works. This is why software is an illusion—the illusion of immateriality—and everything is reducible to hardware. According to Kittler, if the computer were not immersed in a human linguistic context dominated by natural language, there would be no need for HL (high level) programming, but everything would happen at the machine level, as a simple flow of electrical impulses. From this point of view, "the signifiers [could] be considered as voltages […] ultimately everything in a digital computer [can be reduced] to changes in voltages" (Hayles, 2005, p. 45).

Kittler writes:

> Programming languages have eroded the monopoly of ordinary language and grown into a new hierarchy of their own. This postmodern tower of Babel reaches from simple operation codes whose linguistic extension is still a hardware configuration passing through an assembler whose extension is that very assembler. As a consequence, far reaching chains of self-similarities in the sense defined by fractal theory organize software as well as hardware of every writing. What remains a problem is only the realization of these layers which, just as modern media technologies in general, have been explicitly contrived in order to evade all perception. We simply do not know what our writing does.[2]

According to Kittler, the software illusion is caused especially by the strange trend in computer culture to obscure the role and weight of hardware. The physical machine is forgotten to make room for programming languages, but without the former, the latter would make no sense. For this reason, Kittler suggests the creation of a "non-programmable system," i.e., a form of hardware so advanced that it does not need a language for organizing operations, "a physical device working amidst physical devices and subjected to the same bounded resources." In such a device, "software as an ever-feasible abstraction would not exist anymore."

Kittler's thesis was broadly criticized. In *Software Takes Command*, Manovich claims that "There is only software." According to Manovich, the development of the computer as a meta-medium, i.e., a medium capable of connecting, reproducing and transforming all previous media, is not due to the transition from analog to digital, but to the diffusion of programming languages. Software is more powerful than hardware because of its ubiquity. The same software can run on several different machines, devices, and platforms. "There is no such thing as 'digital media'. There is only software—as applied to media data" says Manovich. But what is software? Software is just "a combination of data structure and set of algorithms" (Manovich, 2013, pp. 207–208). "We can compare data structures to nouns and algorithm to verbs. To make an analogy with logic, we can compare [data and algorithms] to subject and predicates" (Manovich, 2013, p. 211; see Manovich, 2001, pp. 218–225). However, how can the simple combination of data and algorithm produce so much culture and imagination? How can this combination "eating the world"?

In my view, many media studies scholars often are "victims" of their own approach. This approach focuses too much on the role of software in communication and its effects on the sociological and cultural level, without tackling seriously the technical aspects and underlying philosophical questions, such as: How does our concept of truth change through software? How does language change through software?, etc. Many arguments are too rhetorical and useless. Furthermore, these works are "often written without engagement with the history of technology literature" (Haigh, 2019, p. 15). Although his theses have a remarkable originality, Kittler remains too tied to the anti-hermeneutic controversy and his objectives are not clear—his approach is defined "post-hermeneutics" (Sebastian and Geerke, 1990). In Chun and Manovich, I see two inverse processes. Chun, on the one hand, stresses too much the paradoxical aspect of software, often giving marginal aspects an excessive problematic nature. Manovich, on the other hand, remains too tied to a definition of software that is flattened on the notion of algorithm, without seeing its intrinsic

complexity and multistability. The essence of software is lost in the multiplicity of media.

Scholars in computer science make the same errors, but from a different perspective. They reduce software to a set of algorithms, namely a mathematical structure. The predominance of a technical approach and the excessive formalization produces three remarkable issues: (a) an excessive intellectualization of software, that is reduced to a type of formal language based on mechanical rules; (b) the difficulty of explaining the relationship between the program and its material execution (parallelism? causality? analogy?); (c) the difficulty of explaining the phenomenon of miscomputation (bugs, malware, etc.), that is, the errors that are intrinsic to the programming—in itself the program, as pure mathematical structure, could not be wrong; (d) an underestimation of philosophical issues.

I want to analyze here only the second point, that concerning the ontological status of software. Colburn (1999) defines software by coining the expression "concrete abstraction," underlining its dual, almost contradictory nature. Abstractness and concreteness are essential properties of software. There is a deep distance between the binary language and the physical states of the corresponding machine; "the characterization of physical state in machine language as zeros and ones is *itself* an abstraction; the kinds of physical state that this language 'abstracts' are boundless. They may be the electronic states of semiconductors, or the state of polarization of optical media, or the states of punched paper tape, or the position of gears in Babbage's 18th century analytical engine." According to Colburn, we cannot reduce a side to the other either: a monistic approach is useless in this case. However, a dualistic approach cannot be causal: how can a string of numbers and operations give rise to a physical state? Colburn prefers to speak of "pre-established harmony" (Colburn, 1999, p. 17), namely a parallelism between code and machine established not by God, but by the programmer.

This conclusion is criticized by Irmak (2012) who prefers to define software as an "abstract artifact," that is an abstract object, built by the human mind for a precise purpose. Software has no spatial characteristics, but it is placed in a time, as well as in a historical period—it is created and can be destroyed, unlike Platonic ideas. Irmak develops his thesis through a comparison with music: "I think that both software and musical works are abstract artifacts which are created by software engineers or composers with certain intentions," and therefore "computer programs are not types and thus the relation between computer programs and their physical copies cannot be understood in terms of the type/token distinction" (Irmak, 2012, p. 70). Nevertheless, Irmak's conclusion also seems problematic or incomplete. How can the following statement "musical works and software come into existence by some human being's act of creation" (Irmak, 2012, p. 71) help us understand the ontological status of software? Computer science fails to deal coherently with the philosophical problem of software ontological status.

I claim that we need a philosophical comprehensive definition of software because of the lack of clarity on this concept especially in media studies and computer science. There are too many different theories about software; a philosophical approach to software has to analyze the limits of each of them and looking for a possible synthesis. In this paper, I argue that a philosophical comprehensive definition of software is possible. Philosophy can help us to clarify the analysis of media studies and computer science and propose new solutions to better understand the role of digital technologies in today's society. My philosophical approach will be manly continental, but open to discussing with analytical tradition.

The paper has the following structure. In section "Engineering in written form: the five criteria," starting from some remarks on the history of programming languages, I will try to define a list of minimal requirements (five criteria) that something needs to meet to be qualified as software. All these requirements share two characteristics: the written form and the effectiveness, that is, the need to be executed by a machine. In section "Software as text: a hermeneutic model," I focus on software as form of writing. I develop this idea by using Ricoeur's hermeneutic model. I claim that software is a type of text. In section "The grammatology of microprocessor," I focus on the second aforementioned feature: the effectiveness of software, i.e., the fact that software is a form of writing that becomes what it says. I claim that the effectiveness of software is based on the analogy between electric circuitries and Boolean logic. This point is underlined also by Sack (2019). I want to improve this view by claiming that this analogy is the condition of possibility of the software hermeneutic process. The main result of this paper is that software cannot be considered as an object nor as a form of language. Software is a complex hermeneutic process.

Engineering in written form: the five criteria

The OED defines software as "Programs and other operating information used by a computer." The Cambridge dictionary takes the approach that software is "The instructions that control what a computer does; computer programs." Encyclopedia Britannica offers a more detailed explanation by stating:

> Software comprises the entire set of programs, procedures, and routines associated with the operations of a computer system. The term was coined to differentiate these instructions from hardware—i.e., the physical components of a computer system. A set of instructions that directs a computer's hardware to perform a task is called a program, or software program.[3]

What does "program" mean? How was the concept of software born? In order to reply to these questions, we have to overcome the common definitions and go back to Turing:

> A simple form of logical control would be a list of operations to be carried out in the order in which they are given. Such a scheme […] lacks flexibility. We wish to be able to arrange that the sequence of orders can divide at various points, continuing in different ways according to the outcome of the calculations to date. We also wish to be able to arrange for the splitting up of operations into subsidiary operations (Turing, 1946, p. 43).

Turing is completely aware that the program cannot be a mere list of instructions because this scheme "lacks flexibility." Program has to be flexible. Program has to be able to transform its own instructions and data storage. Program has to do all these things as fast as possible. As Priestley (2011, p. 1) says,

> Computers possess this flexibility because of the great range of *programs* that can be run on them. A program is a set of instructions telling a computer how to perform a particular task: the flexibility of the computer is therefore limited only by the ingenuity of its programmers in describing complex activities in a way that can be interpreted by the machine. In fact, a better way of looking at the situation is to notice that computers perform only the single task of carrying out the instructions in a program: universality is not an intrinsic property of computers but is derived from the range of programs that can be written for them. Programs are often referred to as *software*, as opposed to the electronic *hardware* that makes up the computer itself. This terminology marks a basic distinction: whereas computers are physical devices, programs are linguistic,

> or logical, entities. A program can be thought of as a text, and the conventions governing how program instructions should be expressed so as to be interpretable by a computer are thought of as defining a programming *language*.

The word "program" in programming context was first used by the ENIAC team (Haigh, 2019, p. 6). The history of programming has to be seen as a part of a complex history in which logic and technology ignored each other for a long time. "The history of programming is seen to stand at the intersection of the two fields of *machinery and language*, and in particular to be closely related to attempts to give a mechanical account of language on the one hand, and a linguistic account of machines on the other" (Priestley, 2011, p. 3; emphasis added). This is a very important remark. A programming language tries to put together two elements: machinery and language. Logic and mathematics are only tools that are used to realize this synthesis. They are only functional for the synthesis between machinery and language.

The search for this synthesis dates from the beginning of the scientific revolution with Francis Bacon and François Viète (see Priestley, 2011, pp. 17–53). Various elements composed the historical evolution of programming language: (a) the progressive mechanization of data processing with Babbage's machines (1822, 1830) or the Hollerith Tabulating System (1890); (b) the logic and mathematical investigations on the concept of "effective computability"; (c) the theory of formal languages in the works of Carnap, Tarski and Church; (d) the creation and development of ENIAC (1946) and EDVAC (1949), the first machines based on the von Neumann architecture and the logic of stored-program computers (von Neumann's *Draft Report*, 1945). In stored-program machines the instructions are held in internal memory (see Haigh and Priestley, 2019, pp. 153–158; Campbell-Kelly et al., 2018, pp. 56–65; Mahoney, 2011). This approach was motivated by the need to make instructions available at high speed, but it also allowed two new coding techniques to be introduced. "Furthermore, programs could modify data in the store, a feature that made possible *the writing of programs that could modify their own instructions to a potentially unlimited extent*" (Priestley, 2011, p. 157; emphasis added).

The Draft Report represents a moment of *closure* as much as a moment of invention, a point where the efforts of many people over the preceding decade to design machines capable of large-scale automatic calculation reached a widely accepted conclusion. The *Draft Report* creates a concrete paradigm which, as the response to it at the Moore School course showed, enabled workers in the field to agree on the basis of the computer design and focus in a concentrated and collaborative way on its implementation. Outside the world of computer builders, however, the stored-program principle attracted little immediate attention (Priestley, 2011, p. 154).

Looking for a philosophical comprehensive definition of software also means trying to better understand this complex historical process. This definition must provide us with a set of characteristics that belong to the object to be defined (X) and to it alone. Such a definition should include the necessary and/or sufficient conditions. Necessary conditions are those that something must possess in order to be defined as X. Sufficient conditions are those which it is sufficient for something to possess in order to be defined as X. In the literature, a necessary and sufficient condition is indicated by the expression "if and only if." For example, a prime number is such if and only if its divisors are 1 and itself, and no other. The expression "if and only if" is called "biconditional."

Let us try to create a biconditional definition of software based on the remarks above. I would say that software is:

1. a form of engineering, thus it is an artifact and has functions;
2. this engineering produces sets of instructions linked to operations, tasks, and data, i.e., the programs;
3. these programs are connected to a physical machine, namely the computers that execute them;
4. these programs can be of different types.

Any object that can be called software if and only if it can satisfy all these conditions.

Let us try to test each of these four criteria. A set of instructions without any connection to the physical functioning of specific machines such as computers would make no sense; it would be just a useless set of rules, but not software. Software has to be executed by a computer. However, a simple set of instructions connected to a computer but without the application of relevant engineering skills cannot be software. In order to produce complex software, a long circular process of analysis, design, development, testing, and maintenance is needed, and skilled people to carry out all these steps. Once tested or released, the user is also involved in the software process. The user's knowledge of software is very different from that of those who have produced it. Developers also differ in terms of the roles they occupy within the software lifecycle, and their experiences differ as a result. Software is a different thing and a different experience to each of the people who play a part in its design or use, and one might wonder if there really is such a thing as an exhaustive knowledge of a particular software.

Is our biconditional definition satisfactory? It is but only to an extent. There is a feature that links all the criteria. Software is engineering *in written form*; it is a form of writing; designing and producing software means above all writing, using a written language. Any programmer recognizes this point. "Software's specificity as a technology resides precisely in the relationship it holds with writing, or, to be more precise, with historically specific forms of writing" (Frabetti, 2015, p. 68). In software, writing gains full autonomy with respect to any writer or reader. Software is a form of writing that is not intended to be read as such, in fact, "for a computer, to read is to write elsewhere" (Chun, 2013, p. 91). "Software is a special kind of text, and the production of software is a special kind of writing" (Sack, 2019, p. 35). Moreover, software not only does what it says; it *also becomes what it says*: this a crucial idea that the present paper wants to analyze. Software is not only performative in the sense of natural language. It is a form of writing that transforms into what it says.

All the criteria we have distinguished are forms of writing: the programs are writing, the operations performed are forms of writing, the imagination of the programmer develops into a writing. Therefore, I propose to add a fifth criterion to our list. I claim that software is:

1. a form of engineering, thus it is an artifact and has a function;
2. this form of engineering is realized in a form of writing not made to be read;
3. this engineering produces sets of written instructions linked to operations, tasks and data, i.e., the programs to be executed,
4. connected to the functioning of a physical machine, namely the computer;
5. and these programs can be of different types.

An objector could reply by saying that our approach is too superficial, and that writing is only a marginal aspect of software. According to our potential objector, focusing too much on writing would be like trying to explain how a car works by saying that it is made of metal. However, this objection is not appropriate. The

written form of software is not comparable to the metal which a car is made of. An algorithm, a list, a procedure or a formula would not even be thinkable without writing. This is the point. Writing is not simply a material phenomenon; it is above all a cognitive structure which is abstract and concrete at the same time.

Another possible objection is that our thesis (the essence of software is writing) can also be valid for other forms of human activity, such as recipes, games, travel plans, etc. However, this objection does not capture a decisive point. In a recipe, writing is a means; it is meant to be read and send a message. In software, *writing is the subject*. Writing becomes completely independent of any form of reading. In a laptop, "reading" software means re-writing it elsewhere. In digital technologies every form of acting is writing; every action must be written in order to be done. Even the programming language would make no sense if it were not written.

Writing has transformed the human mind more than any other invention. In *Phaedrus*, Plato criticized writing because of its inhuman pretense at recreating what is only in the mind, outside of the mind. This is why writing is also illusory, destroys memory and weakens the mind, according to Plato. However, as Havelock (1963) has shown, Plato's epistemology is based on a rejection of the old oral culture—after all, Plato *writes*. In writing, sound is reduced to space and the concept is separated from the living present. Thanks to its autonomy, writing improves the level of awareness and analytical thinking and creates completely new cognitive structures (see Ong, 1982; Goody, 1977; Serfati, 2005). In the *Origins of Geometry*, Husserl (1978) claims that the appearance of writing allows a new level of development of meaning through the inscription process, which can be read repeatedly. The connection between writing and computation has been emphasized by Bachimont (2010) who develops a "transcendental deduction" of the concept of computation from that of writing.

Writing has profoundly transformed the structure of our brain, as Wolf (2008) shows. "Human beings invented reading only a few thousand years ago. And with this invention, we rearranged the very organization of our brain, which in turn expanded the ways we were able to think, which altered the intellectual evolution of our species. Reading is one of the single most remarkable inventions in history; the ability to record history is one of its consequences" (Wolf, 2008, p. 3). The ability of reading is based on "new connections among structures and circuits originally devoted to other more basic brain processes that have enjoyed a longer existence in human evolution, such as vision and spoken language" (Wolf, 2008, p. 5).

How can writing help us better understand software? My argument is that writing constitutes the fundamental mediation between the two elements that a programming language tries to put together: machinery and language. The history of computers and the history of computing must be considered as parts of a much broader history, which is that of writing.

In the following sections, I analyze the two crucial aspects of the five requirements that I have distinguished: the written form and effectiveness (software does not only do with it says but becomes what it says). I use a specific philosophical model: Ricoeur's hermeneutics. I claim that software is a type of text. This is a way of testing our five requirements. I want to emphasize the unity of my intent: writing and effectiveness are two fundamental characteristics of the aforementioned five criteria of the software definition. Therefore, in the next sections I discuss these two features in a philosophical perspective (Ricoeur's hermeneutics) in order to better clarify the five criteria.

Software as text: a hermeneutic model

Ricoeur and the philosophy of technology. I have distinguished five criteria that a good definition of software must respect. I have then underlined the fact that all these criteria have two characteristics in common: the written form and the effectiveness. In the present section I further clarify the meaning of the term "writing" for software by using Ricoeur's hermeneutic model. In the next section, I will analyze the other aspect of this particular form of writing that is software: its effectiveness, or performativity, i.e., the fact that software becomes what it says.

How can we use Ricoeur's hermeneutic phenomenology to understand technology? What do text and software have in common? As Kaplan (2006) points out, Ricoeur has never been directly interested in technology; in his work we do not find a specific analysis of the technological revolution that marked the twentieth century. In few passages disseminated in his books, Ricoeur seems to mainly share the Heideggerian vision of technology which is pessimistic and apocaliptic. Nonetheless, Ricoeur's hermeneutics can offer us powerful tools for understanding technology, especially digital technology. In fact, technology creates new cultural and social meanings. "The empirical approach to technology studies understands it hermeneutically and contextually: technology must be interpreted against a cultural horizon of meaning, like any other social reality" (Kaplan, 2006, p. 49). From this point of view, "Ricoeur's work becomes extremely helpful for understanding it philosophically […]. Ricoeur's hermeneutic philosophy provides a model for interpreting the meaning of technological practices" (Kaplan, 2006, p. 49). This means that Ricoeur's hermeneutics provides a model for analyzing the cultural practical and social meanings of technology. Furthermore, as other authors point out (Romele, 2015, 2017; Romele and Severo, 2016; Coeckelberg and Reijers, 2016; Gransche, 2017), technology has an intrinsic narrative dimension and contributes to public and social narratives. From this point of view, Ricoeur gives us a hermeneutic model to analyze the narrative and imaginative dimensions of digital technology.

I share these readings of the Ricoeurian hermeneutics. I think that Ricoeur can make a very important contribution to one of the most interesting perspectives in the contemporary philosophy of technology: post-phenomenology.

Postphenomenology is "a particular mode of science-technology interpretation" Ihde writes (Rosenberg and Verbeek, 2017, p. 1). This approach combines the style of Husserl's phenomenological investigation and the tradition of American pragmatism. Science and technology are conceived as a fundamental mediation between the human being and world. They shape our way of relating to and thinking of the world. For example, in Ihde, the main exponent of post-phenomenology, the relationship between technology and the human being is conceived in four ways: embodiment relations (when a technology is "embodied," a user's experience is reshaped through the device), hermeneutic relations (the use of a device involves an interpretation), and alterity relations (we relate to the devices in a manner somewhat similar to how we interact with other human beings)[4] and background relations (the devices define the environment in which the subject lives). Therefore, technologies have to be understood in terms of the relations human beings have with them, not as entities "in themselves."

By focusing on mediation, postphenomenology reconceptualizes the intentional relation in two distinct ways. "First, it investigates its fundamentally mediated character. There is no direct relation between subject and object, but only an 'indirect' one, and technologies often function as mediators. The human–world relation typically is a human–technology–world relation" (Rosenberg and Verbeek, 2017, p. 12). Secondly, postphenomenology "does away with the idea that there is a pre-given subject in a pre-given world of objects, with a mediating entity between them. Rather, the mediation is the *source* of the

specific shape that human subjectivity and the objectivity of the world can take in this specific situation. Subject and object are *constituted* in their mediated relation" (Rosenberg and Verbeek, 2017, p. 12). Hence, intentionality is not a bridge between subject and object, but rather a fountain from with the two of them emerge. Starting from this rereading of intentionality, postphenomenology develops a relational ontology.

Now, Ricoeur's hermeneutics arises properly from a critique of the classical concept of intentionality in Husserl (Ricoeur, 1986). Language is the fundamental mediation that allows us to get in touch with ourselves, with others and the world. But language is first of all symbol, metaphor, and text (Ricoeur, 1975, 1986b); it is polysemic, ambiguous, and must be interpreted. For Ricoeur, language is not merely a set of fixed rules, but mainly a product of the imagination. However imagination is not a subjective phenomenon; it is an extremely complex social, cultural, and ontological process, which develops through sedimentation and innovation (Ricoeur, 1983). This idea can be applied to Ihde's notion of mediation: technology is an imaginative process whose stratigraphy has to be reconstructed. The Ricoeurian theory of imagination, which is mainly inspired by Kant, is a useful tool for understanding how digital technology and the "datafication" of society mediates between humans and the world. It reinforces and unifies the five criteria we have described above.

Software as text. The purpose of this section is to apply Ricoeur's text model to the notion of software. I claim that software itself is hermeneutic in its structure.

The core concept of Ricoeur's hermeneutics of text is that of distancing or "distanciation." In the case of the text, the distancing refers to how meaning gains autonomy from (1) the intention of the original author, (2) the original world of circumstances in which the author wrote or which s/he wrote about, and (3) the original readers of the text when it was first produced (for instance, the Greek community who listened to or read Homer's *Odyssey*). Writing is the condition of possibility of distancing.

For Ricoeur, writing is not simply a technical fact, but *an essential hermeneutical factor*. Ricoeur defines the text as "written discourse." Through writing, the language becomes autonomous and thus opens up to endless interpretations: This process is the text. However, in this process "distanciation" is always connected to what Ricoeur calls belonging, or, following Gadamer's terminology, "appropriation." The autonomy of the text is the condition for the text to be read and understood by several readers, who re-read their own experiences through the text and transform its meaning. From this point of view, following Heidegger and Gadamer, Ricoeur talks of a "dialectics between distanciation and appropriation." He overcomes the structuralist point of view on text and literature by arguing that the discourse can never be completely reduced to its syntactical and grammatical structures. "Structuralism was correct that texts have a structure. But this structure varies depending on the kind of discourse inscribed in the text, so discerning that structure and how it contributes to shaping that discourse helps one identify the discourse as being of a certain type or genre" (Pellauer, 2016). The language "goes beyond itself" because it is essentially mediation between the subject and herself/himself, between the subject and other subjects and between the subject and the world. Therefore, language is not only a set of symbolic structures that are "internal" to the text, but also a movement that refers to the external world and the reader's praxis (see Ricoeur, 1965, 1969, 1986b).

In his most influential works on hermeneutics, Ricoeur mentions two concepts: "explaining" and "understanding." The first is taken up by Dilthey and Weber and indicates the method of exact sciences, while the second is taken up by Heidegger and Gadamer and has an ontological and existential sense. The aim of "explaining" is objectification and causal explanation. The "understanding" concerns meanings and the relation between meanings and subject's existence. Ricoeur tries to overcome a merely dualistic view of these two concepts. He proposes to articulate them into a single hermeneutical model: it is necessary to "explain more in order to understand better." This means that the methods of disciplines such as linguistics and literary criticism, which treat the text as an object by analyzing its structures, must be integrated into a broader ontological understanding. The text is written and thus it is an object with specific structures. This justifies the possibility of the objectifying approaches. However, the text is not just an object. Thanks to its structures, the text "projects a world," says Ricoeur. In this world, the subject recognizes himself/herself and his/her way of being. Imagination plays an essential role in this process (see Ricoeur, 1983–1985). The imaginative work of the reader responds to the imaginative work of the text. As we read in Ricoeur (1975), the text is defined as "a heuristic model" given to the reader, i.e., a tool that allows the reader to discover new aspects of her/his experience and praxis. Imagination makes the reader able to translate what the text tells him/her into his/her existence and praxis. As Kearney (2006, p. 16) claims, Ricoeur "argue[s] that the meaning of Being is always mediated through an endless process of interpretations—cultural, religious, political, historical, and scientific. Hence Ricoeur's basic definition of hermeneutics is the art of deciphering indirect meaning".

By using Ricoeur's hermeneutical model, I underline three points:

(a) *Software can be interpreted as a higher degree of "distanciation"*: In software, a language (a set of characters and rules) becomes autonomous with respect to its author, the circumstances and the original readers. Hence it becomes an autonomous subject capable of acting in the world and establishing relationships with human beings or other machines. In software, writing is independent of any possible reading. I have already quoted Chun (2013, p. 91), but it is useful to repeat that "for a computer, to read is to write elsewhere." Software is then the realm of pure writing.

(b) *Software behaves like a text*: In software the dialectics between "distanciation" and "appropriation" is realized through different layers. Some layers are material, others immaterial, some visible, others invisible, some imply the participation of the human being, others do not. I propose an overview in the Fig. 1. A movement of increasing abstraction is realized through these levels. The language becomes less human and closer to the machine, until it becomes pure machine code, that is, binary language. The binary code is included and executed by the machine, the CPU. Through these levels a translation process takes place: the user gives a command, this command is defined by an interface and then expressed in HL language. The code string in HL language is translated into a compiler, that is, in another language, which completely restructures the code in order to make another translation possible, the one in to the machine code. These steps are purely "internal," syntactic and structural—as computer science shows (see Turner, 2018).

The lowest level, "Users," is the level of all those—humans or machines—who have not designed and built software, but only use it. It is a visible level as it involves all the social, psychological, economic, and political effects of using software. "Code costs money and labor to produce, and once it is written requires continual inputs of energy, maintenance, and labor to keep functioning. […] the political economy of software cannot be ignored" (Berry, 2011, p. 61). Software is a social reality deeply connected to the last decades of capitalist economy.

The next level is that of design, i.e., the iterative loop that involves software creation (the planning, analysis, design,

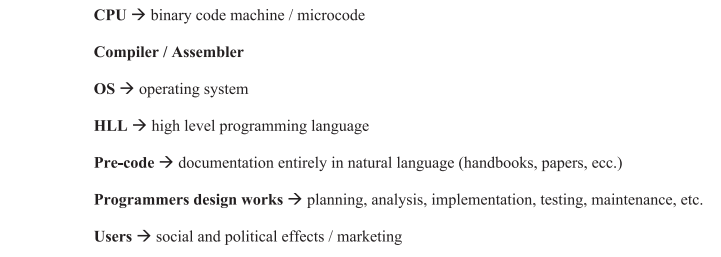

Fig. 1 The different layers that compose a software system. Some layers are material, others immaterial; some are visible, others invisible. From top to bottom the abstraction increases.

development, testing, etc.) and marketing. Here I refer to Latour's "trial of strength" (Latour, 1987) to emphasize how essential the testing phase is in particular. Software is never the product of an individual, but always a collective and collaborative endeavor, which involves different kinds of activities and actors. These documents translate the design phase into a written form.

At the next level we have the code itself, consisting of a formal language for the description and the realization of the software tasks. The HL language is written in a specific code and its strings still contain the interpolation of the comments of programmers in natural language that explain in detail how the code works.

There are two intermediate levels between the HL language and the machine core, the CPU. The first one is the operating system, which "forms the host machine on which a high-level programming language is implemented" (Gabbrielli and Martini, 2010, p. 22). The second is the set of languages that translate—re-write—the HL language into machine language, or machine code. Indeed, the machine code must be binary because the CPU can understand only strings of 1s and 0s. Therefore the HL programs must be "translated" into machine code to be implemented. Then the machine code is translated into electrical voltages. "Programmers use high-level language to develop application program; in order for the program to become an executable form, it must be converted in machine code (binary)" (Elahi, 2018, p. 161).

As I said above, re-writing the HL program into machine code is the task of another language, the compiler, which restructures and re-write the entire program.

Through a complex procedure of lexical, syntactic and semantic analysis, the compiler produces an intermediate code, the assembly language, which is subjected to an optimization process. The assembly is called a low-level language".[5] This is not an exact translation: the compiler integrates the program by modifying it—for example, by identifying errors or making interpolations. The compiler/assembler allows the translation of the HL program into machine code. These strings correspond to the operations to be performed by the CPU. At this point software is actually realized; it performs its function. "Each of these instructions tells the computer to undertake a simple task, whether to move a certain piece of data from A to B in the memory, or to add one number to another. This is the simplest processing level of the machine, and it is remarkable that on such simple foundations complex computer systems can be built to operate at the level of our everyday lives" (Berry, 2011, p. 96).

The process is summarized in Fig. 2.

(c) As we can see, software is a network of writing and re-wrting: pre-code → HL language → compiler/assembler → code machine. From praxis (design loop) to praxis (actions implemented by a machine) through a series of translations. Now, this movement of internal structuring of software that proceeds from the the user to the code, from the world to the machine, is connected to another movement, another level of translation, which goes in the opposite direction: from the code to the user, from the machine to the world. The machine code is translated into electrical impulses and, therefore, into actions in the world. This is a hermeneutical circle. Software distances itself from the world (HLL-assembly-machine code), but only in order to return to the world and transform the concrete experience of the subject, i.e., the user.

Like a novelist, the programmer defines a sequence of actions and interactions and, by using writing, gives a certain autonomy to her/his history/program. The program is then translated into a series of actions. It becomes alive and acts like any other agent in the social world. In the translation of the machine code into electrical impulses, software returns to the user and therefore to the world. This is the "software appropriation." The modality of this appropriation is not reading: The user does not read the code. The user *interacts with the machine*. Appropriation is interaction. Software gives us new model of praxis by acting. In other words, I would say that software realizes the deep desire that animates every type of text.

Now, as Ricoeur says, the wider the distancing, the more intense the movement of belonging. The distancing movement of software corresponds to new forms of appropriation, much more complex than simply reading the text. Software "projects" an imaginative world in front of us simply "doing it," i.e., "building it." The subject interacts with software and this mutual shaping transforms her/his experience of the world, namely, her/his way of acting and thinking of- and in-this world.

Hence, I claim that software is a hermeneutical process which is organized in two phases: (1) the explaining, i.e., the internal structuring in several layers of writing; (2) the understanding, i.e., the execution, that is the interaction with the subject-user. I explore this second layer in the next section.

The grammatology of microprocessor

Let us examine now the connection between software and machine, and therefore the enigma of the effectiveness of software. I said above that software "projects" an imaginative world in front of us simply "doing it," i.e., "building it." In order to do that, software has to intervene in the world. But how? How can a text produce concrete actions by itself and interact with human subjects? The enigma lies in the relationship between symbolic and reality: How can a series of symbols produce real effects? How can a symbol, in itself, change the world? As I said above, software is not only performative in the sense of natural language. Software does not say what it does, *but it becomes what it says*. This aspect is fundamental to explain the application of the "software-text" to the real life, namely, the understanding, the execution, and the interaction with the subject-user.

Let us look inside the CPU and see what happens. All the CPU activities can be described as a series of translation, namely, writing and re-writing.

The assembly language is used to re-write the HL program faster. A string of assembly includes a series of fixed operations (ADD, addition, SUB, subtraction, etc.) and the positions in CPU registers (R1, R2, etc.). For instance, the string "ADD R1, R2, R3" means adding up the contents of R1 and R2 and then storing the result in R3. In more technical terms, assembly language contains three main elements: (a) the command, (b) the position in the registers (more on this below), and (c) the position of the data in the random access memory (RAM). Furthermore, the assembly language is based on the nature of the processor in use: "Each CPU has a known instruction set that a programmer can use to write an assembly language program. Instruction sets are specific to each type of processor" (Elahi, 2018, p. 162).

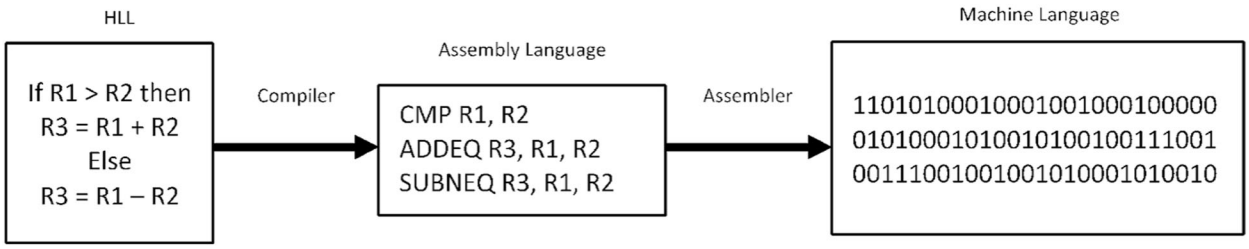

Fig. 2 The software translation process. The diagram represents the transition from a HL program to its restructuring in terms of a compiler language and then of machine code.

Only when the program is re-written by the assembly language, the CPU can understand it. But what does "understand" mean for the CPU?

Let us consider the fundamental activity of the CPU, the so-called FDE cycle (Fetch, Decode, Execute; see Gabrielli and Martini, 2010, pp. 8–9; Elahi, 2018, p. 204; Bindal, 2017, p. 156). The PC register (Program counter, instruction pointer) receives the memory address that allows the CPU to access, through the Address Buses, to the main memory and find the data, which is then transferred to the CPU through the Data Buses. The data and the relative memory address are then stored in CPU memories, i.e., the data memory register (MDR) and the memory address register (MAR), which are the interfaces between the main memory and the CPU. MAR and MDR are memories that CPU can access quickly and directly. The instruction is stored in the instruction register (IR) until it is decoded by the control unit (CU). The CU controls the clock in relation to which all the CPU parts work, checks the status of the CPU by means of the so-called flag signals, and reads the signals passing through the control buses (interrupts, acknowledgments). The CU decodes the instruction and translate it into a sequence of operations that the arithmetic-logic unit (ALU) can perform. In many cases we also talk about "microprogramming," or firmware, microcode, or circuit-level operations. "Microcode is generally not visible to a 'normal' programmer, not even to programmers who deal with assembler code. It is also strictly associated with the particular electronic circuitry for which is designed—in fact, it is an inherent part of that circuitry" (Frabetti, 2015, p. 164). The microcode allows CPU to perform more complex actions. Each instruction corresponds to a sequence of micro-operations performed directly by the machine. Thus, the same machine code string can refer to one or more sets of micro-operations.[6]

We can summarize the entire FDE process in Fig. 3.

The string of machine code does not "tell" the CPU what to do; the FDE is not a communication process. The CPU is not the "interlocutor" to whom the programmer speaks by using software. The CPU does not "answer" to the code machine string. Once a string is received, the CPU re-writes that string: this is its way of understanding. For the CPU, understanding means writing elsewhere. Every operation is the manipulation and re-combination of data stored in the memory, where "stored" means "written in the hard drive." Storage is another form of inscription (Kirschenbaum 2004). Manipulating and moving data mean writing them elsewhere. All the main CPU logical operations (ADD, SUB, MUL, Or, Exclusive Or, Nand, Nor, Exclusive Nor, Shift Right, Shift Left, etc., see Bindal, 2017, pp. 280–300) are only a re-writing of data stored in the CPU memories.

This process is possible because the CPU is already "written," it is an *Ur-writing*. For this reason, I mention Derrida's concept of "grammatology" which means the importance of writing in human experience (Derrida, 1967). From this point of view, Ricoeur and Derrida are not so far. In both, the concepts of trace and inscription have an essential role. Here I use the concept of grammatology in order to highlight the ontological nature of writing in software.

The fact of "being already written" is precisely what allows the CPU to write and rewrite the code, and therefore to carry out all its fundamental operations. But what does it mean "to be already written"? The microprocessor is the *extreme inscription* because in it writing reaches the depth of matter, that is, its atomic structure. The microprocessor is in fact a set of millions of integrated circuits: the transistors. The basic material of integrated circuits is silicon. To become a transistor, silicon must undergo a long purification process. After the purification process, silicon is melted to obtain pure crystals. The crystals obtained from the fusion of the raw silicon are cut into thin disks, the so-called "wafers." The wafers are then cleaned until they are completely free of defects. At this point, silicon disks are covered by photosensitive material and exposed to ultraviolet rays. The material is poured onto the wafers while they are rotating in order to guarantee uniform arrangements. The reaction that occurs during the exposure of the wafers to ultraviolet rays is like what happens to film when taking a photograph. The exposition is realized using a filter thanks to which it is possible to impress precise shapes on the silicon—this is the inscription process. The impressed shapes correspond to the project, to the design of the microprocessor. A lens projects the image of the filter onto the wafer in such a way that the projected and imprinted image is at least four times smaller than the original. The exposed but not imprinted parts of the wafer are then dissolved and eliminated using a special solvent. The exposed parts of the wafer are then bombarded with ions. After the bombardment, the photosensitive material is eliminated. The wafers are placed in a copper solution. The last phase is the application of metal layers to connect the transistors according to the design.

What allows this physical reality to act in a computational manner, that is, to perform increasingly complex operations? What allows the relationship between physical structure and abstract structure in any digital device?

I want to emphasize only one essential aspects of the CPU making process, namely, the regulative analogy that lies at the origin of the microprocessor. I call this analogy "regulative" following Kant's terminology. I claim that this analogy gives a meaning to everything that happens in the microprocessor. This is the analogy between Boolean logic and electrical circuits, as theorized first by Peirce (1993). This point is underlined also by Sack (2019). I want to improve this view by claiming that this analogy is the condition of possibility of the hermeneutic appropriation of software.

The analogy seems obvious, but it is not the case. The computer is built by this analogy. We are used to thinking of the computer as a mathematical machine, but it is not at all. The computer is just a collection of electrical material and voltages. We interpret this series of voltages as logical operations. Through this primitive hermeneutic act, the set of silicon and voltages becomes a computer. This analogy works well, but it is only an

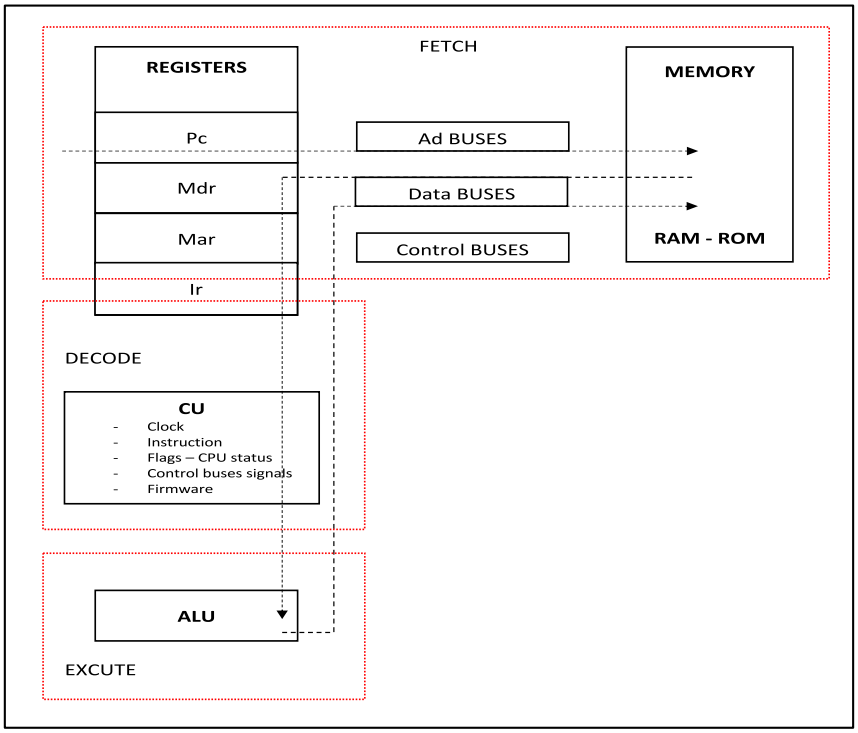

Fig. 3 The FDE cycle, the fundamental activity of the CPU. Once a string of code is received, the CPU re-writes that string elsewhere.

analogy—and therefore leaves open the space of possible errors—and is not explicable.

In the *Critique of Pure Reason*, the distinction between regulative and constitutive concepts marks the distinction between the faculty of reason and the faculty of understanding. The understanding—with all its pure concepts and categories—constitute the possibility of experience. Every experience must necessarily conform to the categories and forms of pure intuition. Reason, on the other hand, has a purely regulative role: the concepts of reason are not categories to which each datum of experience must conform. The ideas of reason do not have any correspondence in the world of experience. However, these concepts may have another function for knowledge, namely, a regulative and methodological function. In other words, the reason gives us a horizon of reference, which cannot be experienced, in relation to which we can give meaning to our knowledge, to all our experience (Friedman, 1992).[7]

The analogy between electrical circuits and Boolean logic gives a meaning to all computer operations, but cannot be experienced. This analogy makes the effectiveness of software possible. Through this analogy the software not only does what it says, but also becomes what it says. The voltages are nothing more than the last translation of the code, its final re-writing.

Conclusions

This paper claims that a comprehensive definition of software is possible from a hermeneutic point of view. I have chosen Ricoeur's model of text in order to understand how software work and interact with human subjects. I claim that software is a writing and re-writing process that implies an interpretation on two levels, epistemological and ontological. The main conclusion of the paper is that continental philosophy can help us understand aspects of software that are often not considered by a merely logical-mathematical approach. The hermeneutic process involves two moments: (1) the explaining, i.e., the internal structuring of software in its various layers; (2) the understanding, i.e., the execution, the interaction with the subject-user. The first dimension is epistemological, the second ontological. Software redefines our knowledge and our being-in-the-world.

The hermeneutic analysis of software opens the way to new research on the cultural and existential dimension of digital technology. From a Ricoeurian point of view, an important line of research would be to understand how the narrative identity transforms in contact with software and artificial intelligence. The concept of narrative identity has a crucial importance in Ricoeur (1990) because it represents the synthesis between the two fundamental senses of identity, namely the selfhood (ipse) and the sameness (idem). Taking advantage of the potential of the narrative function, the narrative identity puts selfhood and sameness together. Personal identity is therefore based on the narrative, on the constant reconfiguration of the experience of time and memory. It is within the framework of the narrative theory elaborated in Ricoeur (1983–1985) that the "concrete dialectics" of selfhood and sameness reaches its full development. Selfhood is the temporal identity that admits changes, while sameness is identity in the sense of permanence, of fixity.

Now, digital technologies are deeply narrative, in the sense that, thanks to their pervasiveness, they bring about narratives about humankind and the world. Narratives and above all self-narratives, i.e., narratives that the subject must tell himself/herself to give continuity and meaning to his/her existence, have expanded and complicated. This is why digital technologies generate new conflicts of the self.

The narrative function of new technologies is demonstrated in particular by Coeckelbergh and Reijers (2016) that investigates how blockchain technologies such as cryptocurrencies can transform social world. The blockchain produces plots through financial transactions. "Like texts, technologies have the capacity to configure our narrative understanding by organizing events into a meaningful whole: a plot that encompasses both humans and technologies" (Coeckelbergh and Reijers, 2016, p. 11). The blockchain is an example that can be applied to many other different things: machine learning, Big Data, self-driving cars, chatbots, etc. But how do the narrative resources of new

technologies affect the self? My hypothesis is that digital technologies build a new form of self, which is precisely an "algorithmic self" (Elliott 2020, chapter 6). Today the construction of personal identity is based on multiple narratives conveyed by large AI systems that define habits, dispositions and preferences. The problem is that these large AI systems entail also forms of surveillance and limitation of human freedom, as Zuboff (2018) has shown. Does this mean that the risk of narrative technologies is to produce an "algorithmic self" based more on the sameness (stasis) than on the selfhood (change), thus blocking the Ricoeurian dialectic? I am not convinced of this. As Elliott (2020) and Lupton (2019) show, Zuboff's theses can be criticized: AI (even its forms of surveillance) does not always have the effect of limiting human freedom. The "algorithmic self" is not the end of the self, but a challenge to the self. Today narrative and self-narrative—in the Ricoeurian sense of the synthesis of heterogeneous—are much more complex tasks. I claim that the construction of the self in the digital world passes through narratives and self-narratives that are capable of revealing and challenging new forms of surveillance, and thereby creating new spaces of autonomy and freedom.

Data availability

All data generated or analyzed during this study are included in this published article.

Received: 10 December 2019; Accepted: 30 July 2020;
Published online: 21 August 2020

Notes

1 https://a16z.com/2011/08/20/why-software-is-eating-the-world/
2 https://journals.uvic.ca/index.php/ctheory/article/view/14655/5522.
3 https://www.britannica.com/technology/software
4 "One common form is computer interface schemes that pose direct questions to the user, such as the ATM machine that displays questions on its screen ('Would you like to make a withdrawal?'), or the 'dialog box' that opens on a computer screen to provide program installation instructions. This is not to claim that we mistake these devices for actual people, but simply that the interface modes take an analogous form" (Rosenberg and Verbeek, 2017, p. 18).
5 Let us therefore call *low-level*, those languages whose abstract machines are very close to, or coincide with, the physical machine. "Starting at the end of the 1940s, these languages were used to program the first computers, but they turned out to be extremely awkward to use. Because the instructions in these languages had to take into account the physical characteristics of the machine, matters that were completely irrelevant to the algorithm had to be considered while writing programs, or in coding algorithms. It must be remembered that often when we speak generically about 'machine language,' we mean the language (a low-level one) of a physical machine. A particular low-level language for a physical machine is its *assembly language*, which is a symbolic version of the physical machine (that is, which uses symbols such as ADD, MUL, etc., instead of their associated hardware binary codes). *Programs in assembly language are translated into machine code using a program called an assembler*" (Gabbrielli and Martini, 2010, p. 5).
6 Microprogramming "is at an extremely low level and consists of *microinstructions* which specify simple operations for the transfer of data between registers, to and from main memory and perhaps also passage through the logic circuits that implement arithmetic operations. Each instruction in the language which is to be implemented (that is, in the machine language that the user of the machine sees) is simulated using a specific set of microinstructions. These microinstructions, which encode the operation, together with a particular set of microinstructions implementing the interpretation cycle, constitute a *microprogram* which is stored in special read-only memory (which requires special equipment to write). This microprogram implements the interpreter for the (assembly) language common to different computers, each of which has different hardware" (Gabbrielli and Martini, 2010, p. 10).
7 The reference to Kant is not an inconsistency. I do not see the contradiction between the hermeneutic position that I have defined above and the use of some Kantian concepts. The concept of analogy in Kant is very complex and presents numerous nuances. However, I do not use here the Kantian concept of analogy, but the Kantian concept of "regulative use," in the sense of a non-constitutive use of concepts for our knowledge. Hence, in this context my use of the term "analogy" is very general: "An *analogy* is a comparison between two objects, or systems of objects, that highlights respects in which they are thought to be similar. *Analogical reasoning* is any type of thinking that relies upon an analogy" (Bartha, 2019).

References

Bachimont B (2010) Le sens de la technique. Les Belles Lettres, Paris
Bartha P (2019) Analogy and analogical reason. In: Stanford Encyclopedia of Philosophy. Stanford University Press
Berry D (2011) The philosophy of software. Palgrave McMillan, New York
Bindal A (2017) Fundamentals for computer architecture design. Springer, Berlin
Campbell-Kelly M, Aspray W, Ensmenger N, Jeffrey RY (2018) Computer: a history of information machine. Westview Press, Boulder
Chun W (2013) Programmed visions. Software and memory. MIT Press, Cambridge
Coeckelbergh M, Reijers W (2016) Narrative technologies: a philosophical investigation of the narrative capacities of technologies. Hum Stud 39:325–346
Colburn T (1999) Software, abstraction, and ontology. Monist 82:3–19
Derrida J (1967) De la grammatologie. Seuil, Paris
Elahi A (2018) Computer systems. Digital design, fundamentals of computer architecture and assembly language. Springer, Berlin
Elliott A (2020) Concepts of the self, 4th edn. Polity Press, Cambridge
Frabetti F (2015) Software theory. Rowman&Littlefield (Media Philosophy), London-New York
Friedman M (1992) Regulative and constitutive. South J Philos 30:73–102
Gabbrielli M, Martini S (2010) Programming languages: principles and paradigms. Springer, Berlin
Goody J (1977) The domestication of the savage mind. Cambridge University Press
Gransche B (2017) The art of staging simulations: Mise-en-scène, social impact, and simulation literacy. In: Resch M, Kaminski A, Gehring P (eds) The science and art of simulation. Vol. I. Springer, Berlin
Haigh T (2019) Exploring the early digital. Springer, Berlin
Haigh T, Priestley M (2019) The media of programming. In: Haigh T (ed.), Exploring the early digital. Springer, Berlin, p. 135–158
Havelock E (1963) Preface to plato. Cambridge University Press
Hayles C (2005) My mother was a computer. Digital subjects and literary texts. University of Chicago Press
Husserl E (1978) Origin of geometry. An introduction by Jacques Derrida. University of Nebraska Press
Kaplan D (2006) Paul Ricoeur and the philosophy of technology. J French Philos 16:42–56
Kearney R (2006) Introduction: Ricoeur's philosophy of translation. In: Ricoeur P (ed.) On translation. Routledge, London
Kirschenbaum M (2004) Extreme inscription: toward the grammatology of the hard drive. Text. Technology 2:91–125
Kittler F (1995) There is no software. ctheory.net. https://journals.uvic.ca/index.php/ctheory/article/view/14655/5522
Irmak N (2012) Software is an abstract artifact. Grazer Philos Stud 86:55–72
Latour B (1987) Science in action. Harvard University Press
Lupton D (2019) Data selves. Polity Press, Cambridge
Mahoney M (2011) Histories of computing. Harvard University Press
Manovich L (2001) The language of new media. MIT press, Cambridge
Manovich L (2013) Software takes command. Bloomsbury, London
Ong W (1982) Orality and literacy: the technologizing of the word. Routledge, London
Peirce C (1993) Writings of Charles S. Peirce: A Chronological Edition: 1884-1886. Indiana University Press
Pellauer D (2016) Paul Ricoeur. In: Stanford encyclopedia of philosophy. Stanford University Press
Priestley M (2011) A science operations. Machines. Logic and the invention of programming. Springer, Berlin
Ricoeur P (1965) De l'interprétation. Essai sur Freud. Seuil, Paris
Ricoeur P (1969) Le conflit des interprétations. Seuil, Paris
Ricoeur P (1975) La métaphore vive. Seuil, Paris
Ricoeur P (1983–1985) Temps et récit. 3 vol. Seuil, Paris
Ricoeur P (1986) A l'école de la phénoménologie. Vrin, Paris
Ricoeur P (1986b) Du texte à l'action. Seuil, Paris
Ricoeur P (1990) Soi-même comme un autre. Seuil, Paris
Romele A(2015) Digital memory and the right to be forgotten. Ricoeurian perspectives Tropos 8(2):105–118
Romele A (2017) Imaginative machines. Techné 22:98–125
Romele A, Severo M (2016) From philosopher to network. Using digital traces for understanding Paul Ricoeur's legacy Azimuth 4(7):113–128
Rosenberg R, Verbeek P (2017) Postphenomenological investigations. Routledge, London
Sack W (2019) The software arts. MIT Press, Cambridge
Sebastian T, Geerke J (1990) Technology romaticized: Friedrich Kittler's discourse networks 1800/1900. MLN 105(3):583–595

Serfati M (2005) La révolution symbolique. La constitution de l'écriture symbolique mathématique. Editions Petra, Paris
Turing A (1946) Proposal for development in the Mathematics Department of an Automatic Computing Engine (ACE). Technical report. National Physical Laboratory, Teddington
Turner R (2018) Computational artifacts. Towards a philosophy of computer science. Springer, Berlin
Voosen P (2017) The AI detectives. As neural nets push into sciences, researchers probe back. Science 357:22–27
Wolf M (2008) Proust and the squid. The story and science of reading mind. Icon Books, London
Zuboff S (2018) The age of the surveillance capitalism. Profile Books, London

Competing interests
The author declares no competing interests.

Additional information
Correspondence and requests for materials should be addressed to L.M.P.

Reprints and permission information is available at http://www.nature.com/reprints

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.