On the ontology of the computing process and the epistemology of the computed.

Abstract: Software Intensive Science (SIS) challenges in many ways our current scientific methods. This affects significantly our notion of science and scientific interpretation of the world, driving at the same time the philosophical debate. We consider some issues prompted by SIS in the light of the philosophical categories of ontology and epistemology.

Keywords: complexity, computational errors, software design, epistemic trust.

Introduction

The introduction of software in scientific research has fundamentally changed our way of understanding and categorizing the world. The distinction between software-intensive (SIS) and non-software-intensive science (NSIS) considered in (Symons, Horner 2014) suggests two considerations: firstly, software has become an essential part of our scientific world-view and science done with little or no help of computational techniques is becoming obsolete, at the risk of offering limited or partial results; secondly, scientific knowledge that is produced or aided by an intensive use of software is generated in ways that make it impossible for its content and its results to be articulated otherwise. For these reasons, computational methods of producing scientific knowledge are non-neutral, both with respect to the description of the world and to our knowledge thereof. (Symons, Horner 2014) propose an important quantitative assessment of SIS in terms of the path complexity of software code. This highly stimulating debate can be further approached by considering some open problems for the Philosophy of Computing directly deriving from the SIS vs. NSIS distinction on the basis of the traditional categories of ontology and epistemology.

Complex ontologies threatened by errors.

The ontology of software intensive science can be analysed in terms of highly complex programs and of the complexity of the systems generated or simulated by such programs. These two aspects are clearly intertwined, but distinct. Let us refer to the former simply as the 'ontology of the computing process' and to the latter as the 'ontology of the computed'.

Conditionality statements and branching paths represent a crucial way of measuring uncertainty in the ontology of the computing process. By this, we mean the number of possible outcomes of program execution, as analysed in (Symons, Horner 2014): the more complex the program, the more uncertain[1] its outcome. This property of the underlying algorithm maps the complexity of what is computed in the exploring possibilities of the program. It is tempting to compare it to the complexity of the brain and the logical processes involved in NSIS research, but this would easily lead to arguing that neural/cognitive complexity in the latter context is still unmatched by any program and possibly will remain so on the long run. It is instead more interesting to focus on the high variability of methods and problem definitions offered by algorithm design and the specific language chosen: the same scientific problem can be modelled (and eventually solved) by differently designed algorithms and the complexity of such designs can be affected inherently by the choice of language. To offer an example limited to programming languages, *if-else* statements are more or less crucial to the structure of the program, depending on the language used: in the case of a functional language like Haskell, where inductive constructions are more common, conditional statements are sensibly less relevant compared to other languages,

1 Notoriously, the level of complexity of algorithmic systems of rules can be systematized in terms of their organizational behaviour with respect to output: fixed-point stabilizing, oscillating, random/chaotic, universally computing. See e.g. the explication in terms of cellular automata offered in (Wolfram 2002).

such as Java or C and derivatives. This is certainly to be taken into account in view of the Turing-completeness of all these languages, as duly mentioned in (Symons, Horner 2014). But the question of principle remains: how much freedom is allowed by the computational construction of a scientific result? This question reflects the well-known problem of characterizing programs in terms of their specification.[2] A program specification should establish what the program is supposed to do. On the one hand, such description will necessarily be incomplete, as the designer will never be able to take into account all the contextual elements valid or even admissible to the correct execution of the program. On the other-hand, avoiding over-specification of the program seems to offer more space for the *de facto* standards of good programming, i.e. the "natural" way the programming practice manages to create well-behaving programs.[3] This latter approach means allowing less structured design methods, favouring a more direct problem-fixing attitude, which clearly induces a different sort of complexity. Such change of perspective is becoming even more pressing in view of disciplines such as machine learning, where deductive methods are replaced by statistical correlations to extract significant patterns. It is therefore becoming essential to understand the difference – and account for a good balance – between structured complexity of algorithm design and the unstructured complexity of code-writing practices and data mining procedures. This crucial issue is still largely ignored: the way programs and computational practices are designed affects directly the result of scientific inquiry in SIS and in particular what is accounted as scientifically valid.

This leads us directly to the second aspect relevant to the ontology of the program: complex programs which often make use of non well-specified design methodologies or not always deductive procedures, are inevitably threatened by computational errors,[4] whose detection and resolution represents a necessary methodological step in establishing syntactic and semantic correctness of scientific results. There exists an extensive branch of current research in software engineering and formal methods dedicated to testing and proving software correct. Most well-known examples are model-checking, verification, certified programming. Each of these has its own foundational approach: to make only some examples, model checking can be seen as an *a posteriori* method of testing programs by looking for output instances that are no models of the specification; certification, instead, is an *a priori* method of asserting the principled correctness of any future program execution. Both are crucially affected by the above mentioned problem of balancing between under- and over-specification. The ability of detecting failure at either compile or runtime (and subsequently solving it, or at least handling it in the sense of preserving functionality) in highly articulated algorithmic patterns reflects their complex structure, in terms of semantic specification satisfaction, syntactical data structure definition and accessibility.[5] The more complex the program, the more essential is to assess correctness and prevent failure. This bears great effects for the 'ontology of the computed'.

It seems, by the above considerations, that a (complex) program proven correct in terms of (complex, sophisticated) techniques could deliver a more in-depth representation of the object of computation; accordingly, our relying on computational methods to know and to prove that we know highlights the complexity of the scientific method underlying SIS. This offers a bridge to consider the epistemology of the computed.

A trust-based, procedural epistemology.

The complex ontology of the computing process leads to an analysis of the epistemology underlying its design: how does scientific knowledge change in view of software development and use? How does its complexity affect scientific comprehension and certainty in the results?

2 See e.g. (Turner 2014), especially sec.2.

3 For a practical view on the problem of program design and standards, see for example the interview to Douglas Crockford in (Seibel 2009), pp.125-127.

4 See (Symons, Horner 2014), especially section 4.

5 For an overview of errors in the design and production cycle of computational systems, see (Fresco, Primiero 2013).

Symons and Horner (2014, section 3) consider the relation between epistemic confidence and software reliability. This is a new way of approaching well-known problems of expertise from social epistemology.[6] Software reliability, which is substituting expert credibility, is arguably a matter of both software design and practice, considered during the whole production and life-cycle of the software. To make an easy example, consider the IBM AI *Watson,* winner of the TV-show *Jeopardy* against two human champions and now programmed to help medical doctors diagnose and treat patients in real life:[7] the standard relation between patient and doctor is bound to change in a crucial way, and security and trust are the relevant areas of software engineering in this context. The reliability of the machine is at stake, but the epistemic attitude of the user is involved as well. The important question for the Philosophy of Computing is not just how we design trustworthy and secure systems, but also what it means for a user that a system and its output are trustworthy and secure. Practices such as debugging, failure detection, static analysis, among others, increase system reliability in objective ways, and the problem of fixing standards for computational experiments and their reproducibility is still open. But this only identifies reliability with the practical nature of systems' performance and consistency. From the user's point of view, system's reliability is also an issue of expectations. These considerations lead to the analysis of reliability as trustworthiness: trust in software offers an effective, although partial, measure of program usability and epistemic certainty it generates, but it also represents a risk. The more trusted a computational process is, the less transparent its use becomes. One way to avoid this risk is reducing trust as delegation and maximising trust as accounting of resources use and origin. This area presents plenty of little explored topics, for example the problem of defining malfunctioning programs and certifying them untrustworthy.[8] The design of computational processes, especially when involved in scientific research, strongly relies on the interpretation of validity as reliability and on the fact that the latter is, at least in some measure, a function of the designer's and user's trust in the system. In this analysis, there is a clear shift towards the practical nature of SIS and an underlying engineering understanding of validity, where the latter has been redesigned in terms of weak and eventual consistency for distributed systems.[9] This means, in other words, that validity and availability of computational processes have become locally constrained, agent-based notions.

This appears to be in seemingly strong contrast with the nature of processes in non-software based science: on the one hand, scientific laws are compared to conditional statements in branching paths of software code; on the other hand, the former seem to have a normative value in describing reality independently of our experience, while the latter seem to be accountable for a procedural, user-dependent characterization. Laws of nature tell us what can be known of the world, given some general conditions; code instead seems to tell us what the user, in her local conditions, can do with the world. Moreover, the scientific description of nature is not affected by performance criteria,[10] while algorithms are by definition evaluated just in terms of those criteria: speed, consistency, scalability, availability, to name a few, and their results are evaluated in view of the user's trust on the non-transparent parts of algorithm execution. This is another reason why experiments by computational means seem to offer an entirely different set of epistemological conditions than those in the natural sciences.[11] This has suggested that computation, in terms of programmability, offers its own model of *nature*, and that natural systems (as known from standard sciences) are comparable, distinguishable, or even conform to such a model.[12]

This intense debate, which can be summarized as the quest to discern algorithmic structures

6 See e.g (Thagard, 2001).

7 http://www.ibm.com/smarterplanet/us/en/ibmwatson/work.html

8 The focus is currently on trustworthiness certification. For an analysis of software malfunctioning, see (Floridi, Fresco, Primiero, 201x). For a first definition of the semantics of untrustworthiness, see e.g. (Primiero, Kosolosky 2013)

9 See e.g. (Lynch, Gilbert 2002).

10 We strictly consider the description of nature by laws, while certainly performance criteria affect the way those laws work.

11 See (Angius 2014) and (Schiaffonati, Verdicchio 2013) for recent debates on experiments and hypothesis testing in computational settings.

12 For such an approach, see e.g. (Zenil 2013).

from rules to create a model of nature, is crucial for SIS in order to understand how the algorithmic representation of law-like phenomena affect our interpretation of the world.

Concluding remarks

The notion of software intensive science starts with determining the complexity of computational processes, but it offers much more in terms of conceptual and technical problems for both scientists and philosophers. The kind and number of questions arising in this context are large and diverse. We have listed some of these problems, reflected in epistemological and ontological terms: from the methods underlying algorithm design and data mining to their effect on scientific results; from the way computational errors (by design or execution) affect our understanding and confidence in the science that computational systems help generate, to the resulting relation between scientific statements and code-based models. These elements are of philosophical interest and have already consequences on the way scientific research is done. SIS is here to stay, and we better have the conceptual and formal tools to understand it.

References

Angius, N. (2014), "The Problem of Justification of Empirical Hypotheses in Software Testing", Philosophy & Technology, DOI:10.1007/s13347-014-0159-6.

Fresco, N. and Primiero, G. (2013), “Miscomputation”, Philosophy & Technology, volume 26, issue 3, pp. 253-272.

Floridi, L., Fresco, N. and Primiero, G. (201x), “On Malfunctioning Software”, submitted for publication.

Lynch, N. and Gilbert, S. (2002), “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, vol. 33(2), pp,51-59.

Primiero, G. and Kosolosky, L. (2013), "The semantics of untrustworthiness", Topoi, DOI: 10.1007/s11245-013-9227-2.

Schiaffonati, V. and Verdicchio M. (2014), "Computing and Experiments", Philosophy & Technology, DOI:10.1007/s13347-013-0126-7.

Seibel, P. (2009), Coders at Work, APress, US.

Symons, J. and Horner J. (2014), "Software Intensive Science", Philosophy & Technology, DOI: 10.1007/s13347-014-0163-x.

Thagard, Paul (2001), "Internet Epistemology: Contributions of New Information Technologies to Scientific Research." In *Designing for Science*. Edited by Kevin Crawley, Christian Schunn and Takeshi Okada. Mawah, NJ: Erlbaum.

Turner, Raymond (2014), *The Philosophy of Computer Science*, The Stanford Encyclopedia of Philosophy (Summer 2014 Edition), Edward N. Zalta (ed.), forthcoming URL = <http://plato.stanford.edu/archives/sum2014/entries/computer-science/>.

Wolfram, S. (2002), *A new kind of Science,* Wolfram Media.

Zenil, H. (2013), “What is Nature-Like Computation? A behavioural approach and a notion of programmability”, Philosophy & Technology, DOI: 10.1007/s13347-012-0095-2.