Design, Malfunction, Validity: Three more tasks for the Philosophy of Computing

Giuseppe Primiero
Department of Philosophy
University of Milan
giuseppe.primiero@unimi.it

Abstract

We present a review of Raymond Turner's book *Computational Artifacts – Towards a Philosophy of Computer Science* (2018), focusing on three main topics: Design, Malfunction, and Validity.

1 Computational Artifacts

A philosophy of computing should be able to guide a society currently thriving with technological evolution. To this aim, the epistemology and ontology of Computer Science object of Raymond Turner's *Computational Artifacts – Towards a Philosophy of Computer Science* [11] are undoubtedly the first and most important steps, offering a philosophical understanding of the notions of program, computation, programming language, machine, along with associated methodologies for their construction and development.

Turner's book does a fantastic job of illustrating the complexity of this task, at the same time providing a well-grounded and substantially solid philosophical theory, based on the interpretation of computational artifacts as technical ones in the sense of the Philosophy of Technology. Specifically, computational artifacts are characterized by a layered ontology and a related epistemology, combining a physical, material nature with an abstract one. The first realization of this duality is known in the literature as the relation between specification and implementation, which has been object of early research [10, 9]. Turner's more refined analysis considers computational artifacts as construed by: function, design, structure and implementation ([11, p.27]). For programs, these elements are instantiated respectively by: specification, design, symbolic program, implementation (see [11, p.27]). Each layer can be associated with a corresponding notion of information, see [6].

Three aspects related to the layered ontology and epistemology of computational artifacts, and which are of major relevance for the continuation of a profitable and impactful research agenda, are:

1. *design*: what is the role that a philosophical and formal analysis of design can play in the context of the philosophy of computing?

2. *malfunction*: how can a theory of computational malfunction be developed and applied to the benefit of both technological development and society?

3. *validity*: what is the appropriate notion of validity for computational artifacts?

These topics emerge in Turner's book, although their analysis is not fully developed. This commentary will highlight their importance, and argue for a research agenda in their direction, also in the light of the social role that any philosophy of computing should play.

2 Design

Designs as objects of formal and philosophical analysis date back to the 1960s and should be understood as methods to rationalise (software) systems in terms of problem-solving processes and logics. Several methodological approaches to design are available in the literature, and the logics of design answer the need for a generalisable and systematic approach. For computational systems, designs are crucial. In Chapter 15 [11, p.129], Turner overviews the problem from a dual perspective:

> *"we introduce programming and software design from the perspective of the philosophy of design. [...] One fundamental question concerns the nature of design:*
>
> - *What is a design, what kinds of things are designs, and what is a good one and how do we judge?*
>
> *[...] This raises basic metaphysical, epistemological, and aesthetic questions about design. [...] Of course, these questions cannot be completely separated from the methodological ones:*
>
> - *Which methods are used in practice to obtain good designs? How do we evaluate them?"*

Here Turner hits a major issue in the philosophical debate on computing: the centrality of design is primarily due to the fact that the result of design is at its higher abstraction level the specification, and the latter is normative for the implementation (see also [11, pp.79-89 and 217-218]). This also means that there is a correctness requirement for designs themselves, along with other desirable properties, like simplicity [11, ch.16], modularity [11, ch.16], and the use of a methodology which guarantees a balance with the speed of development [11, ch.18]. For designs (of technical artifacts at large) as objects and the process of obtaining them, Turner tells us the following [11, p.26]:

> *"The output of the design process is a structural description of an artifact, and there might be many structural forms that fulfill a given function. [...] The design process may itself be divided into several stages, where more informative structural descriptions emerge at each stage. Even so, the final description does not have to be a complete structural description in the sense that all the physical properties need to be included."*

In the case of software (see [11, p.54]), the design process needs to follow requirements specification through a structural description of the intended system by appropriate methods (e.g. UML diagrams or any specification language). Essential properties specific for the good and successful design of programming languages are (see [11, Ch.19]):

- Simplicity, requiring an understanding of minimality and maximality conditions of correctness, which Turner distinguishes in syntactic and ontological terms.
- Expressive power, formulated as a parameter of Turing completeness, while its ontological counterpart can be formulated as the domain of reference.
- Security, enforced through semantic properties of typing, and in combination with uniformity represents a solution to an optimization problem.

The possible conflict between simplicity and expressive power is solved by a balance between principles of (see [11, Ch.20]):

- Correspondence: for each form of definition or declaration there exists a corresponding parameter mechanism, and vice versa.
- Type completeness: every term is typed, types may be contextual independent, no empty types are defined, functions may use parameters, and provide outputs of any type.
- Abstraction: all major syntactic categories should have abstractions defined over them, i.e. phrases of semantically meaningful syntactic classes may be named.

This analysis provides a clear and useful understanding of properties of designs for programming languages.

But the initial ontological and methodological aims of providing a definition of designs for computational artifacts at large, and to characterize their good properties, remain to be addressed. In particular, it is a crucial task for the philosophy of computing to establish characteristics of designs as computational objects, their properties, and how we model them. A logic of design of a system as a specific kind of conceptual logic of the design of the model of a system, that is, the blueprint that provides information about the system to be created

has been recently considered in [1], a task reconsidered further in [7], where a blueprint is a formal object composed by a tuple <process, resources, output>, required to instantiate a system. For such a formal object one can study resource access properties and thereby establish the logic of how computational artefacts are made, but also the logic that validates the design. In the latter case, a number of properties can be studied, for example minimality and maximality conditions on resources satisfying intended requirements; other properties (like those identified by Turner for the design of programming languages, but also others related to: explanation, external and internal correctness, security) should become a major objective of the philosophy of computing.

From a theoretical perspective, designs have the potential to be for the philosophy of computing the formal object of investigation, similar to what has been the case for proofs for over two centuries in logic and the philosophy of mathematics. From a practical viewpoint, designs are the tool necessary to improve the methodology of development and evolution of computational systems.

3 Malfunction

The requirement of correctness for designs impacts another central issue: malfunction. On the one hand, correct designs are the criterion to establish when a computational artifact malfunctions (see [11, p.46]):

> *"specifications are normative in the sense that they fix the criterion of correctness and malfunction for any proposed program."*

To establish taxonomies of computational malfunctions, and of malfunctioning software in particular, has been the task of a number of recent proposals in the literature ([3, 5, 2]), relying on the essential multi-layered structure of computational artifacts. This aspect is also briefly highlighted by Turner [11, p.57], although he does not report on current research in the area:

> *"Each level of abstraction offers different possibilities for malfunction. Once the requirements are employed as a specification for the design stage, there is a possibility for malfunction. Once accepted, the design acts as a functional specification for the design of individual programs. This provides yet another possibility for malfunction. Finally, the actual physical devices may malfunction. Moreover, the conceptual natures of correctness and malfunction at each stage are, as we shall see much later, rather different."*

The philosophy of computing (and of Computer Science in particular) should endorse the problem of defining miscomputation as among the most relevant ones in its agenda in terms of formal correctness criteria on designs.

In view of the impact that malfunctioning computational artifacts have on everyday life and especially in safety-critical systems, there is another aspect

of correctness of designs that we urgently need to account for: the malicious modification of the design of a system to induce malfunctioning. This issue surfaces today mostly in the enormous distribution of malware, known since the 1980s and object of continuous attempts at identification and categorization. From a conceptual and formal point of view, analyses are starting emerging in the literature ([4, 8]) which go beyond the classificatory approach typical of software engineering.

But a more extensive effort is needed to connect the issue of induced malfunctioning with the above presented formal and conceptual description of designs for computational artifacts. A problematic theoretical aspect is that the modification of designs implies that technically an artifact affected my malware is compliant to its (modified) design, and hence not strictly speaking malfunctioning, as it by-passes the principle of correctness as a match to an intended specification. This means to recognize the limits of existing behavioral analyses. We need to investigate thoroughly properties of security and resilience; to provide formal models of behaviour prevision under changing environmental execution condition; and to define designs resistant to undesired modifications. To these aims, *trustworthy designs* should be defined and their properties investigated.

This task can significantly increase the relevance of our research area for the practitioners, in particular software developers and system administrators, and for the legislators, who are to be given clear conditions under which systems can be identified as under threats and actions characterised as culpable.

4 Validity

The aims of defining correct and trustworthy designs and of obtaining a clear conceptual definition of well-functioning, secure computational artifacts, preserving essential properties of evolution and resilience, lead us to a third, overarching main issue for the philosophy of computing: what is an appropriate notion of validity for computational artifacts? Also in this case, the layered ontology and epistemology of computational artifacts play a crucial role.

Correctness is the property usually referred to for computational artifacts since the formal approaches started in the 1960s, and with the modelling practices advanced by Software Engineering since the second half of the 1970s. The term *correctness* refers to the matching relation between implementation and intended specification, and thus it is essentially expressed by a syntactically defined, behavioural property of the physical program. Levels of abstraction, though, are not exhausted by the duality of specification and implementation. A more refined description of the epistemology and ontology of computational systems is offered in [6], including:

- Intention: at this level one reflects on and expresses the computational problem to be solved;
- Specification: at this level the set of requirements needed by the solution of the problem is stated;

- Algorithm: at this level a procedure is formulated which satisfies the requirements and provides a solution to the problem;
- High-level programming language instructions: at this level the task resolution provided by the algorithm is implemented in linguistic constructs for the chosen language;
- Assembly/machine code operations: at this level the operations which need to be performed for the realization of the programming language instructions are translated to the low-level constructs required by the hardware;
- Execution: at this level, operations are executed by electrical charges and the information flow on the hardware.

This description includes both syntax and semantics at different levels. The difference from Turner's analysis, is that it goes beyond the standard understanding of the normativity of rules (see [11, p.79]):

> *"To say that meaning is essentially normative is to say that certain norms are valid, or in force. In terms of programming languages, the" normativity of meaning" has it that any semantic account must provide a criterion of correctness."*

Distinguishing an abstract (intention/specification/algorithm), an implementation (high/low level language) and a physical (execution) level of abstraction for computational artifacts means also to establish properties for each. These properties require to account for:

1. the role of language and encoding of the algorithm;
2. the identification of the physical architecture and its effects on the behavior of programs;
3. the validation and verification of formal and computational models.

These properties allow us to identify several, interdependent notions of correctness: functional, procedural, executional, and of the artifact with its object of study or intended output. Validity and Correctness for the formal, physical and experimental notions of computing should be studied comprehensively, a task which the philosophy of computing at large should make its own.

Turner's contribution leaves open these and other venues of investigation into the ontology, epistemology, metaphysics and ethics of computational systems. The hope is that the community of philosophers of computing will further explore them to establish its role as the most relevant philosophical task of this century.

References

[1] L. Floridi. The logic of design as a conceptual logic of information. Minds and Machines, 27(3):495–519, 2017.

[2] L. Floridi, N. Fresco, and G. Primiero. On malfunctioning software. *Synthese*, 192(4):1199–1220, 2015.

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

[4] Simon Kramer and Julian C. Bradfield. A general definition of malware. *Journal in Computer Virology*, 6(2):105–114, 2010.

[5] G. Primiero. A taxonomy of errors for information systems. *Minds and Machines*, 24(3):249–273, 2014.

[6] G. Primiero. Information in the philosophy of computer science. In L. Floridi, editor, *The Routledge Handbook of Philosophy of Information*, pages 90–106. Routledge, 2016.

[7] G. Primiero. A logic of efficient and optimal designs. *Journal of Logic and Computation*, 2019.

[8] G. Primiero, F.J. Solheim, and J.M. Spring. On malfunction, mechanisms and malware classification. *Philosophy & Technology*, Nov 2018.

[9] W.J. Rapaport. Implementation is semantic interpretation. *The Monist*, 82(1):109–130, 1999.

[10] R. Turner. Specification. *Minds and Machines*, 21(2):135–152, 2011.

[11] R. Turner. *Computational Artifacts - Towards a Philosophy of Computer Science*. Springer, 2018.