Preprint from http://www.philosophyofinformation.net

This is a preprint of a paper accepted for publication in

Synthese (Springer)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

It is a publisher's requirement to display the following notice:
The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a noncommercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

In the case of Springer, it is the publisher's requirement that the following note be added:

"An author may self-archive an author-created version of his/her article on his/her own website and his/her institution's repository, including his/her final version; however he/she may not use the publisher's PDF version which is posted on www.springerlink.com. Furthermore, the author may only post his/her version provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The original publication is available at www.springerlink.com."

On malfunctioning software

Luciano Floridi[1], Nir Fresco[2], Giuseppe Primiero[3]

Abstract. Artefacts do not always do what they are supposed to do, due to a variety of reasons, including manufacturing problems, poor maintenance, and normal wear-and-tear. Since software is an artefact, it should be subject to malfunctioning in the same sense in which other artefacts can malfunction. Yet, whether software is on a par with other artefacts when it comes to malfunctioning crucially depends on the abstraction used in the analysis. We distinguish between "negative" and "positive" notions of malfunction. A negative malfunction, or *dysfunction*, occurs when an artefact token either does not (sometimes) or cannot (ever) do what it is supposed to do. A positive malfunction, or *misfunction*, occurs when an artefact token may do what is supposed to do but, at least occasionally, it also yields some unintended and undesirable effects. We argue that software, understood as type, may misfunction in some limited sense, but cannot dysfunction. Accordingly, one should distinguish software from other technical artefacts, in view of their design that makes dysfunction impossible for the former, while possible for the latter.

1 Introduction

It is a platitude that software is a human construct devised to perform a task. As such, it is artificial and has a function, here in the teleological sense, rather than in the mathematical one. These two properties—human artificiality and teleological functionality—are often considered to be common to artefacts in general, and perhaps characteristic of them.[4] So a description of software as a teleologically-functional artefact suggests that one may understand it by adapting current analyses of artefacts from the philosophy of engineering.[5] Functional categories are the characterising element of artefacts, which are defined by what they are designed to do.[6] If software were essentially different from other technical artefacts, this would show that some function-bearing human products are not artefacts in the sense just specified.

Philosophical research on the nature of software has mainly focused on its rather peculiar ontology and on the nature of the scientific methodology that software may drive. In both cases,

[1] Oxford Internet Institute, University of Oxford.

[2] Sidney M. Edelstein Centre, The Hebrew University of Jerusalem.

[3] Department of Computer Science, Middlesex University.

[4] Admittedly, some artefacts are not obviously function-bearing. For instance, one may be reluctant to attribute a function to some artworks or decorations. But similar exceptions may be disregarded as outside the scope of our analysis.

[5] See e.g. Hansson (2006); Houkes & Vermaas (2010).

[6] Karen Neander writes, "Most (if not all) physiological categories are functional categories [...] (This should seem a familiar idea, because categories of artefacts are similar: a brake is a brake in virtue of what it is supposed to do—was intended or designed to do—not in virtue of having some specific structure or disposition.…)" (1995, p. 117).

the twofold nature of software as an engineered as well as an abstract artefact is salient. This has led several authors to investigate the following topics: the autonomous nature of software as a pattern independent from both data and information (Suber 1988); the concrete as well as abstract nature of software (Colburn 1999); the specificity of the epistemology, ontology and methodology of software engineering as a field of investigation in its own right (Northover et al. 2008) as opposed to the more general paradigm of computer science (Gruner 2011); and the mode of "existence" and "persistence" of software as compared to music (Irmak 2012). The methodology of software driven science has also been a recent focus of philosophical investigation, in particular concerning the role of models (Fetzer 1999) and simulations (Winsberg 1999), the relevance of software for scientific explanations (Symons 2008) as well as prediction (Symons and Boschetti 2013), and the nature of experiments in computing (Schiaffonati and Verdicchio 2014). Within this very rich context, we focus here, specifically, on understanding software as a human-made and teleological artefact in order to investigate the nature of malfunctioning.

Artefacts do not always do what they are supposed to do. Whether because of manufacturing problems, poor maintenance, normal wear-and-tear or other reasons, sometimes tokens, that is, particular physical instances of an artefact type—for example *this* screwdriver, or *that* software program to calculate the factorial of integers—fail to exercise their function completely, or as well as they should. This fact has occasionally served as a kind of litmus test for theories of function: an acceptable theory must be able to account for the fact that a token may have functions in theory (as a token of *that* type) that it does not exercise in practice (as *this* particular token).[7] Consequently, if software is an artefact, it should, at least prima facie, be capable of malfunctioning in the same sense in which other artefacts can malfunction. Indeed, according to Millikan, software should be capable of malfunctioning just by virtue of having a function—and there should be no doubt that software tokens have functions.

Current theories of technical malfunctioning[8] correctly identify *failure of design* and *failure of exercising a function* as causes of malfunctioning. Correspondingly, it has been argued in Fresco & Primiero (2013) that a theory of miscomputation has to account for, respectively, *errors of design* and *errors of functioning* (also referred to as cases of *operational malfunction*), where errors of the former kind should only be attributed to some external agent, i.e., the producer of the computational system, and be traced through various levels of abstraction. The question whether software is on a par with other artefacts when it comes to malfunctioning needs to be better qualified in view of the *level of abstraction* or LoA (Floridi, 2008) considered.

This crucial point about LoAs connects our research with two other philosophical issues relevant to computer science. First, the requirement that software malfunctioning be defined

---

[7] Neander raises this criticism regarding Wright (1973) in (2004) and Ruth Millikan does the same for Robert Cummins' causal-role functions in (1989). Paul Davies turns the criticism around, claiming that historical functions provide no better account of malfunction than causal-role functions in (2000a; 2000b). Indeed, Millikan has argued that this fact applies to functional categories in general, and not just to artefactual and biological types: "it is of the essence of purposes and intentions [and hence, of functions] that they are not always fulfilled" (1989, p. 294).

[8] See, for example, Houkes & Vermaas (2010); Jespersen & Carrara (2011).

taking into account the various LoAs involved in the software production cycle reflects the important role that abstraction plays in the conceptual understanding of computational systems. Computer science focuses on information processing and data manipulation and, therefore, it has abstraction at its core (Colburn & Shute 2007). This feature characterises software development as the evolution of programming paradigms and its relation with data modelling (Colburn 1998). Second, formal verification of software is fundamentally affected by the method of LoAs, for both technical and theoretical reasons (Angius 2013). This is very important for the implications that software malfunctioning has in practice. In particular, the testing of software for verifying requirements satisfiability demands an accurate and precise definition of falsifiable hypotheses about software systems that are being evaluated, and this depends on the LoA adopted (Angius forthcoming). Accordingly, we argue that an appropriate definition of malfunctioning for software needs to start from a correct characterisation of the LoA concerned, which can help the comparison with other technical artefacts.

The article is organised as follows. The first two sections provide the platform on which the rest of the article rests. In Section 2, we introduce an essential distinction between dysfunction and misfunction. In Section 3, we outline the production cycles for traditional technical artefacts and for software artefacts, identifying for each of these two kinds of artefacts the possible source of malfunctioning at the correct LoA. Having provided this background, in Sections 4 and 5 we analyse dysfunction and misfunction, respectively, in technical artefacts. In Section 6, we apply to software the characterisation of malfunctioning suggested in the preceding sections and defend three main theses. In brief, according to the first thesis, software tokens cannot disfunction; according to the second, software tokens cannot misfunction; and according to the third, software types only misfunction under narrow conditions. Section 7 concludes the article by summarising the results of our analysis for both technical and philosophical aspects of software malfunctioning.

2 Two Kinds of Malfunction: Dysfunction and Misfunction

Let us distinguish between "negative" and "positive" kinds of malfunction. A negative malfunction, to which we refer here as *dysfunction*, occurs when an artefact token *t* either does not (sometimes) or cannot (ever) do what it is supposed to do. A positive malfunction, to which we refer here as *misfunction*, occurs when an artefact token *t* may do what it is supposed to do (possibly for all tokens *t* of a given type *T*), but it also yields some unintended and undesirable effect(s), at least occasionally. We intend to examine here whether, and if so, how software may malfunction in the sense of dysfunction or misfunction. We argue that software, understood *as type*, may misfunction in some limited sense, but that it cannot dysfunction. Briefly, the reason for this is that malfunction of types is always reducible to errors in the software design and, thus, in stricter terms, incorrectly-designed software cannot execute the function originally intended at the functional specification level. However, it may very well be the case that 'malfunctioning software' in this negative sense works *correctly* for the *un*intended design.

Our present analysis raises important philosophical and technical issues. If one accepts that it is a universal characteristic of functional artefacts that they are capable of dysfunction, in the

sense intended above, it should follow that software *is not* an artefact in the same sense. Of course this is not to say that software *fails* to qualify as an artefact in the broadest sense of the word, i.e. in terms of being 'a product of human workmanship'. We argue that the latter is a trivial and uninformative qualification (it applies to poetry as well, for example) and that a more precise sense of artefact for software should be adopted. Accordingly, one should distinguish software from other technical artefacts, in view of their design that makes dysfunction impossible for the former, while possible for the latter. An important consequence of the following analysis is that, since we accept that software is function-bearing—indeed, that software categories are *functional categories*—Millikan's claim that members of any functional category must be capable of dysfunction is mistaken.

3 Miscomputation and Software Production Cycle

Before discussing the production cycle of software development, some observations regarding miscomputation in software artefacts are in order. The teleological function of artefacts—the fact that they are produced according to a design to perform certain functions—serves to characterise them in virtue of *the task for which they are designed*. An artefact *A* is supposed to do *X* and its design is correct if it (the design) complies with the functional specification that (should) reflect *A*'s intended behaviour to do *X*. Similarly, the design of a computational system, and software design in particular, is the crucial part of software engineering that pertains to the *functional system level* (FSL) and to the *design system level* (DSL).[9] Software type *T* is produced according to a functional specification *S*, for which a design *D* is correct if it allows any token *t* of type *T* to do what is specified by *S*. At least in terms of this description, it seems that software tokens are artefacts and that, as such, the sort of malfunctioning issues encountered by these systems should be comparable to those of other technical artefacts.

The characterisation of malfunction in artefacts—computational or not—relies on their correct structural description. In general, artefacts are described in terms of their *function*. A screwdriver, for example, is used to turn screws and a program (more on this later) is used to process data and instructions in a manner that determines the behaviour of the executing computer. The *purpose* of the artefact is intimately connected to its design or its usage. When the inherent function of a system, e.g. a natural one, is conflated with an epistemologically-attributed purpose to that system, determining its functioning becomes difficult. Consider the following example. A biological heart pumps blood by means of a coordinated series of contractions, sending blood through the body. Its purpose seems clear, but does an artificial heart that is implanted in a human body and performs the same function have the *same purpose* as a natural biological one? In the case of human-made artefacts, in general, and software artefacts, in particular, there is always some agent involved in the background, be that the designer or user of the artefact. Accordingly, the issue of malfunctioning stems from any gaps that may exist between the function (as intended by the designer) and the purpose (as perceived and experienced by the user) of the artefact.

[9] See Fresco & Primiero (2013) for more details. For an analysis of errors related to ethical, policy and legal approaches to software development and maintenance, see Gotterbarn (1998).

The specification and design of artefacts determine the standards according to which their behaviour is deemed correct or incorrect.[10] An artefact of type *T* functions correctly if it does what it was designed to do *and* according to the intended specification *S*. A correct performance of the function *F*, as defined by the specification, can still be impeded by the artefact's actual design. A violation of the functional specification is involved in the *dysfunctional* sense. In this sense, the infringement of functionality is clearly not induced by either a temporary defect or an external impediment. Rather, the artefact is structurally unable to function as defined by its specification, at least occasionally. If this is the case for *all* artefact tokens of a given type, one is compelled to accept that an artefact *t* of type *T* was designed to do *S* which is meant to instantiate a function *F*, while the design *D* of *T* provides a blueprint for some function *F** different from *F*. This is the case of incorrectly-designed artefacts.

Errors of functioning (or 'operational malfunctions' in Fresco & Primiero (2013)) do not violate the standards set by the specification. They can be expressed in the more 'positive' sense of *misfunction* applicable to artefacts. An artefact token *t* malfunctions when it is able to perform the function *F*, as intended by its specification *S* and according to its design *D*, but it also induces *F** as an unexpected function. In the case of software, the first condition means that malfunctioning software sometimes does not produce the desired output—that is, the specification and design according to which the software was produced may still be satisfied—even if only occasionally. The malfunctioning *t*, at least sometimes, produces the intended output; if not, *S* cannot specify the right type for *t*. Additionally, according to the second condition, the execution of the artefact's functionality produces side-effects that may eventually also hinder its correct functioning.[11]

In order to identify the relevant malfunctions in software—as in other technical artefacts—it is crucial to define their production cycle, so that one may establish where malfunctions emerge and qualify the latter correctly. Let us consider, even if only schematically, the production cycle of two simple artefacts: a screwdriver (see Figure 1) and an elementary piece of software that we shall call `factorial` (see Figure 2).

The ability of a technical artefact to work correctly consists in a two-step process. The first step is the choice of the appropriate function (in the case of a screwdriver, the rotational movement) to

---

[10] The literature in both Philosophy of Technology and Philosophy of Computer Science often links the role of specification and design in determining artefacts' correctness to a normative aspect. This link relies on different possible understandings of normativity. For example, on one understanding, it is the description of what an artefact *does* translated into the specification of what an artefact *should do*. On another understanding, it is an artefact's design that indirectly establishes the criteria for ethical and legal evaluation of the consequences of the artefact's use. Yet, on another understanding, it is that stable and successful realisation of technological artefacts that *requires* agents to behave so as to enable their intended functioning. For more on this see, for example, Vincenti (1990), Radder (2009) and Turner (2011).

[11] Notoriously, in Computer Science the notion of a side-effect is not a negative one. It is used to refer to the ability of a program to modify the state of a system or produce an observable interaction with the environment, besides returning an appropriate value of the function called. Here, and in the ensuing discussion, we use the term *side-effect* in its common sense of an unexpected or unforeseen result of an action.

satisfy the proposed specification (to turn screws). The second step is the mapping of this function into the correct tool design (a tool that has the correct tip which can be inserted into the corresponding head of the screw and that, when rotated, can move the screw accordingly). The possibility of a malfunctioning screwdriver is, therefore, defined by either a function that does not satisfy the specification (e.g. an ill-defined function such as hitting the screw in order to rotate it), or a tool-design that does not satisfy the intended function (e.g. a wrong tip that does not correspond to the head of the screw). Finally, malfunction can originate at the level of physical implementation. At this level, the steps involved are the choice of material that can perform the required function (e.g. plastic and metal for a screwdriver, but not glass) and its construction (e.g. plastic for the handle, metal for the tip). These are what we already referred to as, respectively, forms of failure by design (incorrect functional definition) and failure of exercising a function (incorrect tool design and incorrect physical implementation). Figure 1 provides a summary.

Cycle	*Functional Artefact*
	Screwdriver
Specification	to turn screws
Function Definition	rotational movement
Design	fit to a screw and rotate (anti-)clockwise
Physical Implementation	choice of material for best fit and construction

Figure 1. A schema of the production cycle of a screwdriver.

Next, let us consider the production cycle of `factorial`, our chosen example of an elementary piece of software. A description of the program is as follows.

```
long factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
}
```

In this program, the type of allowed objects is defined as integer. The conditional statement

declares the result of applying `factorial` to be 1 when the value of the argument `n` is 0; otherwise, it returns the argument multiplied by a recursive call to the function on `n-1`. Figure 2 describes the production cycle of the software implementing this program.

Cycle	*Functional Artefact*
	`factorial`
Specification	to obtain the factorial of an integer
Function Definition	integers to integers
Algorithm Design	return 1 if input is 0; otherwise return the result of multiplying the input by the recursive application of this very same function on the input reduced by 1.
Software Programming	C commands: `if-else; return; ==;*; -`
Physical Implementation	choice of material for best fit (sufficient RAM)

Figure 2. A schema of the production cycle of a piece of software called `factorial`.

The description and definition of design is what distinguishes software artefacts from other technical artefacts. For the latter ones, design is understood as a direct implementation of the function identified to satisfy the specification. Program design requires more than that. First, it requires the design of the algorithm or the logical rule(s) satisfying the function specified at FSL. Then it requires the implementation of such a rule (or rules) in a given programming language. The translation required by software programming is peculiar, in that it involves an additional language. When considering malfunctioning in a program, we can partially recover some of the steps considered above for other technical artefacts. For example, a function may not satisfy the specification, e.g. an ill-defined function type, such as `factorial` for type `real` (i.e. real numbers), or an algorithm may not satisfy the function, e.g. use a sum function in the recursive call. An additional step is present in software artefacts, too, namely program design. A choice of the programming language is required and the formulation of code for instantiating the algorithm designed to satisfy the identified function. In our case, the C language, defined over `integer`-objects, requires the commands: `if-else; return; ==; *; -;` in order to implement the algorithm. Finally, in the case of software-artefacts, the level of physical implementation is the source of operational malfunctions.

This brief and simplified overview of the production cycles of technical and software artefacts already suffices to identify the different LoAs where malfunction can occur. It also identifies

where, within the cycle, lies the difference between software artefacts and other technical artefacts, namely at the programming level. Our next task is to consider whether such a distinction can explain different forms of malfunctioning affecting software and other technical artefacts.

4 Dysfunction

For the most part, discussions of artefact malfunction have focused on whether an artefact can do what it is supposed to do. The focus has been on whether the artefact exercises its function by bringing about the desired outcome whenever used properly. This common view is expressed more or less explicitly by Millikan (1989, p. 295):

> *"[A]n obvious fact about function categories is that their members can always be defective…hence unable to perform the very functions by which they get their names".*

Neander (1995, p.11) echoes this claim:

> *"[A] biological part functions properly when it can do what it was selected for and malfunctions when it cannot."*[12]

Presumably the situation is similar for artefacts, although two potential pitfalls need to be avoided.

First, it is not the artefact's actual performance in one or more uses, but its *capacity*. Artefacts may function properly and nonetheless fail to accomplish their tasks, as when a well-functioning[13] anti-aircraft missile misses its target. However, if the anti-aircraft missile *cannot* (rather than *did not*) hit its target, then it is a dysfunctioning missile.[14] As Preston (2000, p.26) remarks, judgments about well- and malfunctioning imply "a justified expectation of capacity or disposition", that is, also a counterfactual assessment, in suitable type-situations, of what the artefact would do, *if* it were used in an appropriate situation, rather than only an assessment of what it *did* do in some particular situation.

Second, it is also generally accepted that dysfunction applies only to proper functions and not to accidental uses. If one uses an artefact in a manner not intended by its designer nor sanctioned by common usage, then its inadequacy to perform as one wishes is not a symptom of

---

12 Neander refines this rough definition later by specifying that "what it was selected for" should be interpreted as the "lowest level of description" applicable.

13 We use the terms "well-functioning" and "properly functioning" interchangeably throughout.

14 We assume that its target is one that the missile is designed and reasonably expected to hit. If the missile is designed for slow-moving aircraft, then the fact that it cannot strike a modern jet fighter is irrelevant to whether it is functioning properly.

dysfunction. A telephone may be used as a doorstop, but if it does not serve this use well, then it would be incorrect to classify it as a dysfunctioning doorstop. It is not a doorstop at all, but it is merely being used as though it were one. In Franssen's terminology (2006), the telephone "is not and does not *make*" a doorstop. This restriction may be legally problematic in more complex scenarios (who is accountable for the misuse of a tool, for example?), but is harmless to our considerations, since we are evaluating software in terms of its *proper* functions.

According to the previous analysis of an artefact production cycle, malfunctioning by inability to perform the intended and designed task can be addressed at different levels. First, it can be addressed at FSL, if the intended specification is unattainable, e.g. because it is ill-defined. Second, it can be addressed at DSL, if the design offered does not match the required specification, e.g. because it satisfies an entirely different sort of function. Third, it can be addressed at the physical implementation level, when it is impossible to satisfy the design and/or the specification, whenever either the physical implementation or the choice of material *prevents* the performance of the required task. The first and second type of problem affect artefact types, while the third affects artefact tokens. As the above quotes by Millikan and Neander suggest, the kind of malfunction we consider here applies to artefact tokens rather than types. Either an individual artefact token *can* perform as it *should* (by virtue of being a token of its type) or it cannot. In the former case, it functions properly, whereas in the latter it dysfunctions. Indeed, one *needs* the artefact type to provide the right context for the proper evaluation or, more precisely, to select the right LoA needed for one's evaluation. Thus, a cheap dart gun may not hit its target often, but as long as it hits its target as often as other guns of the same type do, it is not dysfunctional. We do not expect the cheap dart gun to be as accurate as a more expensive professional gun, but only as accurate as similar tokens. Certainly, some caution needs to be exercised when interpreting the reference class "similar tokens", for one would not want to compare a broken dart gun to other broken dart guns and conclude that it is well-functioning. Yet, these issues do not hinder the development of a proper theory of dysfunction.

An artefact type incapable of doing what it is supposed to do means that unexceptional tokens of that type will be unable to perform their function. The analysis here should not address specification design as the relevant LoA where dysfunctioning takes place: one may call a ball-pen, which does not write at all, *badly designed*. One should not say that the tokens of a badly designed type, for example, a ball-pen that writes very poorly on most kinds of paper, dysfunctions. They behave as well as one can expect for that type, namely, badly. Finally, one would also be reluctant to say that such tokens are functioning properly, since they cannot achieve their functional goal. The terms 'proper' and 'well' convey an approval of the token's capacity that would be unwarranted in this case. We say 'reluctant' because some exceptional tokens of a badly designed type may actually work effectively enough to warrant a more positive assessment and qualify as well-functioning, if only fortuitously.

There are two ways in which a particular token *t* can perform comparatively poorly with regard to its function.

1. An artefact token *t* may be less likely to bring about the desired outcome. For instance, an automobile starter motor with a few missing teeth will start the car more often than not, but

occasionally, when the missing teeth are aligned with the flywheel, it will fail to do so. Because it fails to start the car more often than "normal" starter motors of the same type do, it is dysfunctional. When a token is less likely to bring about the desired outcome than one justifiably expects for its type, we say that the token is *unreliable*.

2. In some artefact tokens, functions are realised to a greater or lesser degree. A starter motor either starts the car or it does not, but windshield wipers can produce either a well-cleaned surface or a streaked and dirty surface (or anything in between). If an artefact token *t* is incapable of achieving its goal as well as one justifiably expects, then it is an *ineffective* token. An old and cracked wiper that cannot clean the windshield as well as we reasonably expect is dysfunctioning *because* it is ineffective.

With this terminology clarified, we can now give a preliminary definition of dysfunction.

> **Definition 1 (Dysfunction)** *An artefact token t* dysfunctions *if and only if it is less reliable or effective in performing its function F than one justifiably expects for tokens of its type T.*

Because tokens may be members of several different types (each with possibly multiple functions), this definition should be taken as relative to a selected LoA, identifying the function and artefact type under consideration. To give a full account of the definition, we would need to make explicit the notion of justifiable expectations. But let us postpone a fuller analysis and accept this as our working sketch of dysfunction so that we may proceed to define *mis*function.

5 Misfunction

Roughly, an artefact token dysfunctions when it is incapable of performing as well as it can reasonably be expected. That is, we have certain justified expectations about how tokens of type *T* perform, but this particular token *t* of type *T* fails to meet those expectations. In these cases, we reasonably claim that *t* is malfunctioning. There is another way in which artefacts may malfunction.

Rather than failing to do what they should do, an artefact may malfunction by doing something that it was not intended to do, while still having the capacity to do what it should. Consider a film camera that overheats when turned on, but otherwise functions normally.[15] The overheating does not affect the camera's capacity to take pictures, but it does pose a risk to the user, namely, that she may be burned when using the camera. In this case, the camera is not dysfunctioning, since the assumption is that it can take perfectly good pictures when used appropriately. Nonetheless, it is malfunctioning—or more precisely, *misfunctioning*—since a proper usage poses an unintended risk to the user. An artefact misfunctions when its proper usage produces an undesirable, and at

[15] This example is drawn from the recall notice for certain models of Olympus film cameras (http://www.cpsc.gov/cpscpub/prerel/prhtml06/06250.html). These cameras were prone to overheat due to defects in the flash circuit. The recall notice reports no other symptoms.

least in principle unrelated, side effect. We clarify this initial definition in this section.

The distinction between dys- and misfunction is subtle, since the misfunction of an artefact token may be due to a dysfunction of some component. A water heater that emits carbon monoxide into its surrounding environment is misfunctioning, since it is producing an undesirable effect while performing its function. But modern water heaters come with components designed to prevent such emissions (such as seals or air filter doors) and so, typically, this undesirable effect is a symptom that at least one of these components are *dysfunctioning*, a seal that is supposed to prevent carbon monoxide emissions is failing to fulfil its function.

It is common that an artefact misfunctions due to a dysfunction of some component. To take another example, an automobile that functions adequately, yet produces an annoying squeal, is misfunctioning. Such squeals are commonly symptoms that a component is not performing as it should—say, that the serpentine belt is slipping, due to age or misadjustment of the belt tensioner. The car may nonetheless be a reliable means of transportation, and so the artefact itself is not dysfunctioning, but rather is producing an undesirable effect while performing its (intended) function. Because misfunction is often a symptom of a related dysfunction, the distinction between the two conditions may easily be overlooked. Nevertheless, producing an undesirable side effect is clearly different from being unable to perform a function. Usually a careful handling of the LoA at which the artefact is being analysed facilitates the identification of what is misfunctioning and what is dysfunctioning.

One may object that this distinction is in fact mistaken, merely due to an incomplete description of the relevant function. After all, it is reasonable to suppose that cameras have the function of taking pictures without causing harm to the user. If the proper function of a camera includes this restriction, then the overheating cameras are dysfunctioning, since they are likely to harm the user. According to this objection, then, misfunction is subsumed by dysfunction. A complete description of the function of a camera, say, is that it takes pictures without causing harm to its user, interfering with other appliances, producing distracting noises, smoke, foul odors, and so on.

Our reply to this objection is that it simply demands too much from the concept of function. To put it more technically, it pretends to work in an LoA-free context, thus generating a slippery slope problem. Because there is no limit to the sorts of undesirable side-effects that an artefact may cause, a full list of them is always going to be inevitably incomplete or too generic, if no LoA is fixed. Thus, if we attempt to broaden the notion of function to exclude negative effects, our functional ascriptions will always end with a “and so on” or a “and the like”. In order to deny that there are two distinct kinds of malfunction, the notion of “function” would be encumbered to the point that it is practically unusable.

Instead, the desire that the camera works without overheating produces a *constraint* on how the artefact should perform its function. An artefact’s function is the positive effect that its use should bring about. In this case, the camera is supposed to record images without burning the photographer, but this condition simply restricts the manner in which it should realise its function. Thus, it is a constraint on, rather than a component of, the artefact’s function. Failing to

satisfy such constraints can be evidence of misfunction rather than dysfunction.

The proposed distinction between dys- and misfunction is important, not least because it has legal implications in terms of standard practices of recalling faulty products. Dysfunction claims apply to tokens, rather than types, since our judgment about artefact capacities is a comparative one: does the token concerned perform as well as *normal* tokens of the same type? Misfunction does not depend on such comparisons. What is at stake is whether an artefact produces a negative effect that is *inessential* for its function. This condition may apply to types as well as to tokens. When it applies to tokens, misfunctions are symptoms of hidden dysfunctions; when misfunctions apply to types, they are not further reducible to dysfunctions but rather attributable to poor design. If every Olympus Infinity Twin failed to take pictures, it would not be dysfunctioning *per se*, but rather that type would have failed to be properly designed. Alternatively, one may say that that particular type would be dysfunctioning when compared to the more general type 'camera'. On the contrary, if every Olympus Infinity Twin overheated each time it was turned on, then the type itself would be misfunctioning, because it was poorly designed and would have to be recalled rathern than merely repaired.

Take another example: a recently manufactured gas range lacked an adequate heat shield beneath the oven.[16] As a result, these ranges tended to scorch the floor beneath. This is clearly not a case of dysfunction, since the oven cooked food as it is supposed to. Furthermore, it is not a case of misfunction caused by a dysfunctioning component: it was not that the heat shield failed to perform as it should, but rather that no heat shield was installed. Due to this design oversight, normal usage of the oven damaged the floor. Hence, the artefact type misfunctions due to a poor design: a well-functioning oven should not damage the floor underneath.

Our discussion has thus far focused on the distinction between dys- and misfunction, but there is another distinction that should be examined. What is the difference between a well-functioning artefact that nonetheless produces negative effects and a misfunctioning artefact? An automobile produces pollutants when used as intended. There is no doubt that these pollutants are undesirable; we would much prefer cars that do not pollute. Nevertheless, we do not claim that because cars pollute, they are misfunctioning. How do we distinguish the negative effects of a polluting auto from the negative effects of a floor-scorching oven?

Both effects are undesirable and neither effect is an intended consequence of the artefact's design. All cars produce pollutants and, clearly, most ovens do not damage the floor. So, we may conjecture that because the oven's negative effect is an aberration, the oven is malfunctioning while the car is not. Indeed, this seems to point in the right direction, but we argue that the distinction between the two cases is more than merely *statistical*.

While the production of pollutants was not intended by the car's designers, it was certainly foreseeable. Modern automotive engineers are well aware that cars produce pollutants and they choose to manufacture them nevertheless. In fact, modern automobiles are designed to reduce pollution compared to earlier models.[17] It is hard to imagine that one would, in normal

---

16 See http://www.cpsc.gov/cpscpub/prerel/prhtml06/06181.html.

17 It may be that pollution controls are primarily motivated by regulation rather than consumer interest, but this is

circumstances, manufacture or purchase an oven that knowingly damages the kitchen floor. The potential damage produces a decisive practical reason not to use the oven (a warning against use is also included in the recall notice), while the pollutants produced by a vehicle apparently do not produce a similarly strong reason against using the car.

This comparison suggests that normal, undesirable side effects can be distinguished from misfunction in terms of the practical reasons they produce (in the sense specified below), an approach first presented in Franssen (2006). Franssen offers the following definition of malfunction (p. 5):

> **Definition 2 (Franssen's Malfunction):** *'x is a malfunctioning K' expresses the normative fact that x has certain features f and that because of these features, if a person p wishes to achieve the result of K-ing, then p has a reason not to use x for K-ing.*

This definition is notable for its flexibility. It includes both dys- and misfunction. Unfortunately, it is still insufficient to distinguish the two cases presented above. The fact that our car pollutes is a *prima facie* practical reason not to use the car. How strong must the reason be in order to count? One may suggest reading Franssen's definition in terms of *decisive* reasons and assume that the threat of property damage is more decisive than the contribution of pollution. If the assumption is correct, one may seem to have drawn a distinction between the two cases. However, this strengthening excludes too much. An automobile that produces a horrible squealing noise may still be a useful means of transportation. The fact that it produces an annoying noise may easily not be a decisive practical reason to avoid driving it. Nonetheless, we would not hesitate to classify this automobile as misfunctioning: it should not produce such awful noise. The strengthened version of Franssen's criterion still yields a doubtful judgment about the car, as, by that criterion, the car is *not* malfunctioning.

Instead, we suggest that the difference between mis- and well-functioning is *modal*, and concerns the inevitability of the deleterious effects. A car that produces an awful noise misfunctions, because we know that cars do not have to make that noise in performing their function. Similarly, we know that ovens do not have to damage the kitchen floor in cooking and that cameras do not have to produce so much heat in taking a photo. We know this in part because we are familiar with well-functioning cars, ovens and cameras. So, the aforementioned statistical argument was persuasive, but its conclusion was misplaced. The evaluations about misfunction do not refer to how the majority of tokens behave. Instead, an artefact misfunctions if the negative effect it produces is *avoidable*. In our example, we know that other ovens do not damage the floor, so we know that this effect *is* actually avoidable. On the other hand, locomotion by gas combustion induces pollution, so the fact that our cars pollute is not evidence that they misfunction.[18]

---

beside our main point.

[18] Of course, one could still say that standard cars do misfunction by polluting, when comparing them to other locomotion means that satisfy the same functions without polluting. The point is whether this evaluation is clear (it

The concept of "avoidability" must allow for alternatives to be both available without extraordinary effort, and comparable in their utility. For vehicles, a comparable alternative would include, for instance, an extant infrastructure for fuel delivery—as long as constructing such infrastructure would require significant effort, we are likely to accept automobile pollution as unavoidable, rather than a form of misfunction. However, the situation regarding polluting locomotion is changing and vehicles that emit little or no pollution have been becoming widely available at comparable costs. Accordingly, it is possible that our attitudes towards pollution emissions will change in the foreseeable future. Pollution emissions will not merely be a regrettable feature, but rather a form of misfunction—an avoidable and deleterious effect. What we argue will have changed, not the evaluation, but the choice of the correct LoA at which the evaluation is conducted.

All these considerations lead to the following definitions of *token* and *type* misfunctions.

Definition 3 (Token Misfunction):
An artefact token t misfunctions *if and only if it satisfies the following three conditions:*

1. *using t produces some specific side-effects e of type E;*
2. *because of e, one has reason not to use t; and*
3. *other ("normal") tokens of the same type do not produce the same side-effects of type E.*

Definition 4 (Type Misfunction):
An artefact type T misfunctions *if and only if it satisfies the following three conditions:*

1. *using tokens of type T typically produces some specific side-effects of type E;*
2. *because of those side-effects of type E, one has reason not to use tokens of type T; and*
3. *there are in principle other comparable artefact types with the same function which do not produce the same side-effects of type E.*

The comparability of artefact types with the same function immediately fixes the reference to the artefact specification and design, which lies at a higher LoA than just the collection of *all* artefacts of the same type. In particular, the reference is made to a well-designed artefact type. To see this, consider two artefact types *X* and *Y* that realise the same function. Type *X* produces an undesirable effect *a*. Type *Y* produces an undesirable effect *b*. *X* and *Y* are comparable, but neither of them is useful to establish misfunctioning. Because there is another type (namely *Y*) with the same function as *X* but without effect *a*, our definition suggests that *X* misfunctions. But the exact same reasoning suggests that *Y* misfunctions, too. This is possible only in terms of a comparison with the design according to which *X* is not intended to produce *a* and *Y* is not

---

is), not whether it is justified (it may not be).

intended to produce *b*. Counterfactually, if there were a type *Z* satisfying such design without undesirable effects, it would be the one against which both *X* and *Y* would be judged as misfunctioning. Note that the token-type relation is recursive: types can, in their turn, be token of other, "higher" types, in the same Russellian sense in which sets can be members of other sets. In our case, the comparison of *X* and *Y* is only possible because of some common type *S* making the comparison a tertiary, rather than a binary, relation. This point will be discussed in detail in the next section.

The analysis above and the corresponding four definitions suffice to establish the following two claims:

1. a misfunction is a deleterious, avoidable effect of normal usage rather than an incapacity; and
2. misfunction applies to both types and tokens.

We are now ready to apply the two claims to the analysis of malfunctioning software.

6 Software

A characteristic feature of artefacts — and perhaps function-bearers in general — is that they can dysfunction (Wright 1973, Millikan 1989, Preston 2000, p. 24). This seems to be the case for artefact types: if type *T* has the function *F*, then it is possible that a token *t* of type *T* is less *reliable* or *effective* in performing *F*. Misfunction, on the other hand, remains largely unanalysed in the literature. To the best of our knowledge, nobody has suggested to date that the possibility of misfunction is an important feature of artefacts. However, this is a crucial fact in understanding software.

For the present purposes, we adopt the following broad terminological conventions.[19] A *program* is usually a single, complete and self-contained ordered sequence of instructions. The logical unity of this sequence need not correspond to a single physical unity. For a program may consist of different routines distributed over different physical devices (e.g. a remote server and a local machine) and may require some additional components (e.g. from a software library) in order to be executable in principle. For simplicity, we identify a particular program with the source code that implements a given algorithm in a specific programming language. The source code implementation imposes restrictions on the operation types and data types defined at the algorithm level (e.g., dealing with variable of data type 'int', which can store integers in the range -32767 to 32767). A program, then, is identified with a source code in a textual form. The Skype program, for example, is equivalent to its source code implementation. Because the medium on which the source code is stored does not seem to have any particular import for evaluating the correctness of the program itself, even a piece of paper with the instructions printed on it counts as a program as long as it is well formed. An algorithm *x*, which has the teleological function *F*, that is implemented using two different programming languages, L_1 and L_2, yields two different source codes, sc_1 and sc_2, respectively. We say that sc_1 and sc_2 are two program tokens of the

[19] As a disclaimer, it should be noted that there are many subtleties in the distinctions made here that exceed the scope of the article.

same program type, which is defined by F.[20] A program, thus defined, is not executable yet. This brings us to software.

Software is any ordered sequence of *machine executable* instructions. Its execution typically changes some machine state, which can be loosely viewed as a set consisting of one or more memory banks and/or the controllers of the executingmachine. A source code in a textual form needs to be converted first into the corresponding machine language instructions in a process of compilation. The program is compiled into one or more object files for a particular machine architecture (e.g., a Mac computer running Mac OS-X ver 10.9 or a PC running Windows 8.1). But, even at that point, the compiled program, which can be identified as *software type*, is still not executable in that format on the particular operating system.[21] The compiled program has to be converted into an executable through the process of linking, in which a single executable file is created from (possibly) multiple object files. A particular copy of an executable (say, Skype.exe for Windows 8.1 or Skype {Unix Executable File} on Mac OS-X 10.9) is a *software token*.[22]

According to our definition, an artefact token t dysfunctions if it is incapable of performing as well as we expect for tokens of its type T. For physical artefacts, the type/token distinction is relatively unproblematic: artefacts can be distinguished by design, structure, function, etc., yielding the *types*, while *tokens* are physical instantiations of the various types. But for software the issue is subtler due to the tricky ontological nature of software. Even a program requires some physical medium (e.g. some paper) on which the text of the program is inscribed. A software token is manifested via a physical file that is stored on some physical medium (e.g. a hard drive, a USB flash drive or a CD-ROM). Yet, if a CD-ROM, for example, is scratched such that a particular machine instruction gets corrupted and causes an undesirable side effect when the software token - that is, a particular copy of an executable - is executed, is it the software itself that malfunctions?

When one considers a software type, its tokens are all implementations of the *same* underlying program. A software token corresponds to what is standardly called in industry a software *copy*. Such copies are, for example, different installable software packages of the same program intended to run on different machines, where each one is typically assigned a unique serial number and, in case of proprietary software, a licence code. When considering *comparable* software types that perform a similar task[23]—for example, two text-editors (such as MS Word for

---

[20] x is a mathematical entity and as such it can be identified with a mathematical function. We do not offer a taxonomy of programs and software based on this property.

[21] Here, there is another subtle distinction that should be made between compiled and interpreted programming languages. We do not include it, because it is not significant for the remaining discussion.

[22] The type/token distinction can be approached differently if another LoA is considered. For example, if one takes into account only the machine code and not the source code (which is essential for our analysis), one could consider the executable as a type and its copies as tokens. The LoA is crucial to our formulation of Thesis 3. For an accessible discussion of some curious features of programs and software see, for example, Berry (2011, ch. 2 and 4).

[23] Since we are no longer dealing with mathematical functions, but teleological functions, we might say that MS Word for Windows and OpenOffice Writer have a *similar*, but *not* the *same*, basic function, namely, word processing. But they implement different sets of features, only some of which overlap.

Windows and OpenOffice Writer) or two audio/video chat programs (such as Skype and Google Hangouts)—we shall talk of *distributions*. While Skype and Google Hangouts are implemented using different algorithms and different source codes and are, therefore, of different software types, they may be considered comparable artefact types for the present purposes.[24]

Our three main theses can now be stated more precisely. We start with the thesis concerning software dysfunction.

> **Thesis 1 (Software Dysfunction)**: *A software token t cannot dysfunction, since it cannot be less reliable or effective in performing its function F compared with other tokens of the same type T independently of the supporting hardware used to run it.*

To help clarify the peculiar dual ontological nature of software and its implications for malfunction classification, consider, more broadly, operational malfunction in physical computational systems. An operational malfunction in a computational system occurs whenever some piece of hardware fails in some way. For example, a cooling fan that stops working unexpectedly can eventually allow the CPU to overheat and melt down. Any such mechanical failure that adversely affects the expected behaviour of the computational system induces a *physical* miscomputation (Fresco & Primiero 2013).

On the one hand, software is intimately linked to the underlying abstract algorithm that is implemented in a particular programming language. On the other hand, software is physically realised as a computer file and occupies physical memory space. It is debatable whether a case where a physical register, which contains some software instruction, goes awry—thereby causing the software not to produce the "expected" behaviour—qualifies as a *software malfunction*. If such a physical malfunction is indeed *always* attributable to the hardware used to run the software token, rather than the software itself, then there is a very limited sense in which software *can* malfunction. Different software tokens of the same software type will only perform the expected function reliably or effectively, if physical hardware malfunctions do not occur.

In this sense, software tokens do not, and cannot, dysfunction. The evaluation of the correctness of a software token is done on the basis of the underlying algorithm implemented. But once the algorithm is implemented using some programming language, the function of the software is *fixed* by its type. Any potential error in the implementation process can only be attributed to the implementation of the algorithm (cf. Fresco & Primiero 2013).

Typically, "software malfunctions" are simply design errors for which only the designer can be deemed responsible.[25] Suppose that a program with variables *(x, y)* of type *int* was *intended* to calculate ($x * y$), but in the way it *was* inadvertently *written* it actually calculates ($x + y$) instead.

---

[24] One could add here a further classification to refer to the different instances of the same distribution by talking of *software versions*. For example, the ordered numbered - or sometimes alphabetically named - instances of the same software or program issued during a given period of time. The problem is that software versions are not necessarily instances of the same underlying algorithm design. For a program version 1.1 might contain modification to the design of version 1.0, such that the two versions can no longer be considered instances of the same algorithm.

[25] See also Dewhurst (2013, p. 3).

Strictly, a software token of this program should be evaluated against the function ($x + y$), rather than ($x * y$). This software token does not malfunction, because it computes the former function. Other tokens of the software type will not be either more reliable or effective in performing the expected function. All software tokens of this type will compute the *same* function, namely *(x + y)*. This token does not, and cannot, dysfunction.

It could be argued that the performance of a software token is also related to other *non-physical* processes, such as compilers and linkers. In this sense, the possible malfunctioning of a software token is not just induced either by the design or by the hardware. Rather, it can result from a bug in other elements involved in the running of software, e.g. a compiler-specific bug. Yet, such elements are entirely analogous to other software tokens and Thesis 1 is equally applicable. Assume the existence of source code *s* which is correctly designed to perform a certain function *f* and a compiler *C* such that *C(s)* does not result in a correct compilation of *f* in the target language *L*. As a result the execution of *s* in *L* fails, by stipulation. While the original formulation of *f* in *s* was correct and the hardware is not at fault, the end result is still a malfunction that should be attributed to *C*.

Thesis 2 (Software Token Misfunction): *Software tokens of a given type T in isolation do not misfunction, since they all inherit a single software design D and are not comparable with other "normal" tokens of the same type T.*

To understand the plausibility of Thesis 2, consider the bug from 2008 in the multi-player online video game *EVE Online*. A software patch, developed and deployed for *EVE Online*, used a file in the software library named *boot.ini*, which is the same file name used by the operating system for the standard boot instruction file. As a result, when installed and run, the patch caused several thousands computers not to start up. While the malfunction manifested itself at a token level, its root cause should be traced back to the level of algorithm design. The software engineer defined (some of) the routines or (parts of) the code underlying the program incorrectly or did not follow correctly the practice related to software design (as in the case of conflicting file names), such that *every* token of that type would malfunction. This clearly shows that there are *no* other "normal" tokens of the same type (*EVE Online*) that do not produce the undesirable effects in question. Software tokens of *EVE Online*, and of similar software types, cannot be described as misfunctioning.

***Thesis 3 (*Software Type Misfunction)*:** *A software type T_x misfunctions only comparatively, when its tokens produce a side effect that is not produced by software tokens of (a possible) type T_y, where T_x and T_y are tokens of a higher order type (supertype) T_0.*

Let us see how some cases of software malfunction can be correctly described as type misfunction in accordance with Thesis 3. Suppose that a particular version of MS-Word for Windows (call it type *W*) restarted (call such undesirable side effect *e*) every time a user executed

an operation consisting in typing some text in a left to right language, changing the input locale to a right to left language and indenting the text to the right (call it use case *c*). So, by stipulation, all software tokens of type MS-Word for Windows would restart when executed under the specified circumstance. Suppose further that the same behaviour did not produce a similar side effect *e* when using another *existing* software of a comparable type, e.g. OpenOffice Writer running on the same Windows machine (call it type *O*). What are the circumstances under which a type misfunction occurs in the case of *W*? Using software tokens of type *W* typically produces *e* on *c*, but only because *every* such software token would exhibit this behaviour: it is the result of some inherent design error. Because of *e* occurring, one has reason not to use software tokens of type *W* if one uses both left to right and right to left languages. There is, by stipulation, at least one other comparable software type with the same function, namely *O*, that does not produce *e* on *c*. In this loose sense, software may be subject to *type* misfunction.

The comparison of *W* and *O* is possible, as all functional requirements expressed in *W*'s specification and implemented in the actual code are compatible, by stipulation, with those expressed in *O*. For example, if *W* has a function to change the input locale to a right to left language, the same function exists in *O*. In this sense, compatibility admits requirements extension (e.g. that *W* includes some function, which does not exist in *O*, and yet they remain comparable at the relevant LoA), while it requires consistency preservation (i.e. all the *essential* features of *W* exist in *O*). On the other hand, compatibility excludes a comparison between different software types, such as Skype and MS-Word. The abstract representation comprising both specifications for *W* and *O* is more informative than the specification of each piece of software implementing them and it can be understood as the model (or theory specification) of which both *W* and *O* are tokens.[26] We refer to this abstract representation of software types specifications as their *supertype*.

Consider next the following objection. Imagine a world in which *W* were the only existing word processing software[27] and suppose further that *W*, in that world, consistently restarted on *c*. To claim that *W* was malfunctioning — so the objection continues — Thesis 3 requires the existence of some comparable software type (such as *O*), which now does not exist by stipulation. A possible reply is that the identification of a misfunction in *W* can be grounded on the relation between *W* as a software type and its *correct* specification, which corresponds to *W* but *without* side effects *e* on *c*.[28] Unfortunately, such a reply renders the explanation vacuous: just compare the misfunctioning software to one, which possibly does not exist, and does not misfunction. The appropriate reply is rather that, by taking the functional requirements specification at a higher

---

[26] For more details, see, for example, Hodges (1993, 1995), Kirchner & Mosses (2001), Turner (2005).

[27] We thank an anonymous referee for this important objection.

[28] In the example above, *c* may be initially overlooked when the functional requirements for *W* are documented. It will, thus, escape the normal testing process that is common in software engineering practice. If, at some point, this is discovered, *W* will be deemed to malfunction (as indicated by the eventual fixing of *W*, the adding of *c* to the requirements specification and the adding of an appropriate test case). This clearly shows that software cannot be "blamed" for malfunctioning, as it always results from some design error.

LoA as a supertype of *W*, one makes it possible (by the definition of a type) to consider the existence in principle of another token of the *same supertype*, call it *W'*. Software types *W* and *W'* behave as tokens, and the argument about software *token* misfunction (following from Thesis 2) is applicable again. A misfunction occurs if and only if an undesirable side effect occurring in *W* is not manifested in *W'*. If the supertype is specified in terms of other unaccounted requirements, such as the programming language to be used and the intended machine architecture, then the comparison between its tokens might become impossible thereby no type misfunction can be said to occur.

The above analysis shows that, in a strict sense, it seems impossible for software to malfunction — either by *dys*functioning or *mis*functioning. Although there is a loose sense in which software *type* can *mis*function, this is reliant on an analysis that compares programs at a very high LoA. Even then, it is unclear that such "malfunctions" count as a genuine case of *mis*function. For these malfunctions are not induced by software issues *per se*, rather they can always be reduced to design- and specification-related errors.

7 Conclusions

We started from the trivial observation that software is a human construct produced for a particular task and that, as such, it is an artefact by definition. We have stressed, though, that such commonality of definition with technical artefacts is also qualified in terms of the specific production cycle that makes them differ. On the one hand, the ability of technical artefacts to work well is the result of the choice of the appropriate function for the intended specification, and the mapping of the correct design to such function. On the other hand, software is identified by an additional step in its production cycle, by which well-functioning also requires that the choice of programming language to implement the designed algorithm be adequate and that the actual implementation of the algorithm be correct.

We have suggested that software should be considered at the LoA of execution, if one wants to analyse the phenomenon of software malfunctioning. Here, too, one needs to consider whether the algorithm execution indeed matches its design, whether the latter is consistent with the specification, and whether it is internally coherent. Only in the first case can one correctly analyse malfunctioning software in the proper sense. We have shown how this conclusion matches our definitions of token- and type misfunction.

In philosophy of technology, dysfunction has been identified as the core property of functional categories. Our analysis shows that for software, this qualification needs to be rectified. We have argued that executed software tokens cannot dysfunction. A software token cannot dysfunction in the sense of being less reliable or effective than expected, because it will always satisfy its design. It is the latter that can be judged less reliable (i.e., if it does not completely satisfy the intended specification) or less efficient (e.g. with respect to parameters of time, space, or complexity).

Our analysis also suggests that the separation of the notions of dysfunction and misfunction as belonging to different production cycles, or to stages within the same cycle, should be taken into account when malfunction prevention procedures are applied. The software production cycle is,

in particular, the result of different strictly connected levels going from functional specification to algorithm design to implementation and execution. At DSL, where logical and engineering aspects are concerned, efficiency and reliability are considered. This means that one should consider the design of an algorithm against its specification (e.g. does my algorithm satisfy my specification? Does it do so in the most reliable and efficient way?), while the latter can only be evaluated against the selected relevant standards (e.g. does the specification fulfils my intention? Does it do so in the most reliable and efficient way?). At the implementation level, where a particular programming language is concerned, the right choice of elements (e.g. predicates, functions, but also hardware components) will offer the same level of reliability induced at higher levels, but it cannot deviate from the given design. Here, type misfunction can only be considered as an inter-level comparison concerning whether a distribution has better performance values than others.

Acknowledgments

This article was developed initially as a collaboration between Jesse Hughes (see especially Hughes 2009) and Luciano Floridi. We are extremely grateful to Jesse for having allowed us to re-use his very valuable work. We would also like to acknowledge the constructive feedback of the anonymous referees, whose comments enabled us to improve the article significantly.

References

Angius, N, 2013. Abstraction and Idealization in the formal verification of software systems. *Minds & Machines,* 23(2):211-226.

Angius, N. forthcoming. The problem of justification of empirical hypotheses in software testing. *Philosophy & Technology,* DOI: 10.1007/s13347-014-0159-6.

Berry, M. D. 2011. The philosophy of software: code and mediation in the digital age. New York: Palgrave Macmillan.

Colburn, T, 1998. Information modelling aspects of software development. *Minds & Machines,* 8(3):375-393.

Colburn, T. 1999. Software, Abstraction and Ontology. *The Monist,* 82(1):3-19.

Colburn, T., Shute, G. 2007. Abstraction in Computer Science. *Minds & Machines*, 17(2):169-184.

Davies, P. S. 2000a. Malfunctions. *Biology and Philosophy*, 15(1):19-38.

Dewhurst, J, 2013. Mechanistic Miscomputation: A reply to Fresco & Primiero, *Philosophy & Technology,* DOI: 10.1007/s13347-013-0141-8

Davies, P. S. 2000b. The nature of natural norms: Why selected functions are systemic capacity functions. *Noûs,* 34(1):85-107.

Fetzer, J. 1999. The Role of Models in Computer Science. *The Monist*, 82:20-36.

Floridi, L., 2008. The Method of Levels of Abstraction. *Minds & Machines*, 18 (3): 303-329.

Franssen, Maarten. 2006. The Normativity of Artefacts. *Studies in History and Philosophy of*

Science, 37:42-57.
Fresco, N. & Primiero, G. 2013. Miscomputation, *Philosophy & Technology*, 26(3): 253-272.
Gotterbarn, D, 1998. The Uniqueness of Software Errors and Their Impact on Global Policy. *Science and Engineering Ethics*, 4(3):351-356.
Gruner, S. 2011. Problems for a philosophy of Software Engineering. *Minds & Machines,* 21(2):275-299.
Hansson, S. O. 2006. Defining Technical Function. *Studies in History and Philosophy of Science*, 37(1):19-22.
Hodges, W. 1995. The meaning of specifications I: Initial models, *Theoretical Computer Science* 152: 67–89.
Hodges, W. 1993. The meaning of specifications II: Set-theoretic specification, *Semantics of Programming Languages and Model Theory*, ed. Droste and Gurevich,Gordon and Breach, Yverdon 1993, pp. 43–68.
Houkes, W., & Vermaas, P. E. 2010. Technical Functions: on the Use and Design of Artefacts. Dordrecht: Springer.
Hughes, J. 2009. An artifact is to use: an introduction to instrumental functions, *Synthese* 168, no. 1 (2009): 179-199.
Kirchner, H. & Mosses, P. 2001. Algebraic specifications, higher-order types and set-theoretic models, *Journal of Logic and Computation* 11:453–481.
Irmak, N. 2012. Software is an Abstract Artifact. *Grazer Philosophische Studien*, 86(1): 55-72.
Jespersen, B. & Carrara, M. 2011. Two Conceptions of Technical Malfunction, *Theoria,* 77(2): 117-138.
Millikan, R. G. 1989. In Defense of Proper Functions. *Philosophy of Science*, 56(2): 288-302.
Neander, K. 1995. Misrepresenting & Malfunctioning. *Philosophical Studies,* 79(2): 109-141.
Neander, K. 2004. Teleological Theories of Mental Content. *The Stanford Encyclopedia of Philosophy* (2012 Ed.), E. N. Zalta (ed.), http://plato.stanford.edu/archives/spr2012/entries/content-teleological.
Northover, M., Kourie D.G., Boake, A., Gruner, S., Northover, A. (2008). Towards a Philosophy of Software Development: 40 Years After the Birth of Software Engineering. *Journal for General Philosophy of Science,* 39(1): 85-113.
Preston, B. 2000. The Functions of Things: a Philosophical Perspective on Material Culture. In P. G. Brown (ed.), Matter, Materiality and Modern Culture, pp, 22-49. London: Routledge.
Radder, H. 2009. Why Technologies are Inherently Normative. In A., Meijers, A. (ed.) Philosophy of Technology and Engineering Sciences. (Handbook of the philosophy of science, volume 9), 887-921. Amsterdam: North-Holland.
Schiaffonati, V., Verdicchio, M. 2014. Computing and Experiments: A Methodological View on the Debate on the Scientific Nature of Computing. *Philosophy & Technology,* DOI:10.1007/s13347-013-0126-7.
Suber, P. 1988. What is Software. *Journal of Speculative Philosophy* 2(2):89-119.
Symons, J. 2008. Computational Models of Emergent Properties. *Minds & Machines*, 18(4):475-491.

Symons, J., Boschetti, F. 2013. How Computational Models Predict the Behavior of Complex Systems. *Foundations of Science*, 18(4): 809-821.
Turner, R. 2005. The foundations of specification, *Journal of Logic and Computation* 15:623–662.
Turner, R. 2011. Specification. *Minds & Machines*, 21(2), 135–152.
Vincenti, W. G. 1990. What Engineers Know and How They Know It : Analytical Studies from Aeronautical History, Johns Hopkins Studies in the History of Technology [New. Ser., No. 11]. Baltimore: Johns Hopkins University Press.
Winsberg, E. 1999. Sanctioning Models: the Epistemology of Simulation. *Science in Context*, 12(2): 275-292.
Wright, L. 1973. Functions. *Philosophical Review*, 82(2): 139-168.

View publication stats