Philos. Technol.
DOI 10.1007/s13347-013-0112-0

RESEARCH ARTICLE

Miscomputation

Nir Fresco · Giuseppe Primiero

Received: 30 March 2013 / Accepted: 3 June 2013

Abstract The phenomenon of digital computation is explained (often differently) in computer science, computer engineering and more broadly in cognitive science. Although the semantics and implications of malfunctions have received attention in the philosophy of biology and philosophy of technology, errors in computational systems remain of interest only to computer science. Miscomputation has not gotten the philosophical attention it deserves. Our paper fills this gap by offering a taxonomy of miscomputations. This taxonomy is underpinned by a conceptual analysis of the design and implementation of conventional computational systems at various levels of abstraction. It shows that 'malfunction' as it is typically used in the philosophy of artefacts only represents one type of miscomputation.

Keywords Computer science · Specification · Program · Errors · Malfunction · Design · Implementation · Physical computation

1 Introduction

Despite the possibly definitive characterisation of computability offered by the Church-Turing Thesis, the understanding of what is computable has undergone important changes. The debate nowadays oscillates between mathematical computability and its concrete counterpart. Does the definition of an algorithm (viewed as a stepwise solution for some computational problem) square with, say, the Turing Machine (TM) or any other extensionally equivalent computational model? This revolves around the epistemic relation among four aspects of computability and physical computation.

N. Fresco (✉)
School of Humanities, UNSW, Sydney, Australia
e-mail: fresco.nir@gmail.com

G. Primiero
Department of Computer Science,
Middlesex University, London, UK
e-mail: gprimiero@gmail.com

Published online: 28 June 2013

1. The purely abstract notion of a computable function and the problem domain;
2. The algorithm for solving the computational problem;
3. The transition from an algorithm to a program (in some systems); and
4. The physical execution of the program on some computational system.

In what follows, we take computation to be the execution of algorithm(s)[1] and also rely on an extensional characterisation of computation. We maintain that finite state automata, pushdown automata, TMs, conventional digital computers and calculators—all compute. As well, primitive Boolean gates (e.g. AND/OR/NOT gates) and some combinational circuits, such as half adders and full adders, compute but only trivially. If we adopt a broad (and vague) definition of algorithm, such as 'a well-defined sequence of steps that always finishes and produces an answer' (Hopcroft et al. 2001, p. 373), then it is easy to argue that the aforementioned systems compute by executing algorithms. In the present discussion, we shall only be concerned with miscomputations in systems of this type.

Whilst computation has certainly received attention in the literature, surprisingly, miscomputation still lacks a sound characterisation, despite its pervasiveness. One exception in the philosophical literature is the mechanistic account of computation, in the context of which Gualtiero Piccinini emphasises the importance of miscomputation for any adequate account of physical computation (Piccinini 2007, p. 505). A more recent exception in the literature is the instructional information processing account, which gives reasons for the occurrence of computational errors in both trivial and nontrivial computational systems (Fresco & Wolf unpublished). Insofar as physical computational systems are susceptible to errors and verification methods have been developed in computer science to either prevent or isolate such errors, they certainly need to be addressed by any adequate account of computation.

Still, the characterisation of miscomputation offered by Piccinini is too narrow. He characterises miscomputation as a kind of malfunction, that is, as an event in which the computational system fails to fulfil its function. Piccinini lists many cases of miscomputation, including a failure of a hardware component, a faulty interaction between hardware and software, a mistake in computer design and a programming error (Piccinini 2007, pp. 523–524). But, as we argue below, malfunction (what we call *operational malfunction* in the context of our analysis) is only one source of miscomputation. A mistake in computer design, for example should not be classified as an operational malfunction, yet, it still counts as a miscomputation. The objective of this paper is to offer a multi-layered analysis of miscomputation.

Already in 1950, Alan Turing made the following observations about 'mistakes' in computational systems.

> "We may call [… these two types of mistake] 'errors of functioning' and 'errors of conclusion'. Errors of functioning are due to some mechanical or electrical fault which causes the machine to behave otherwise than it was designed to do. In philosophical discussions one likes to ignore the possibility of such errors; one is

[1] This characterisation is not unproblematic. For one thing, the precise notion of algorithm remains unclear and is still debated in theoretical computer science and philosophy of computer science. The first author discusses this problem elsewhere and proposes an alternative account of computation as instructional information processing (Fresco & Wolf unpublished).

> therefore discussing 'abstract machines'. These abstract machines are mathematical fictions rather than physical objects. By definition they are incapable of errors of functioning. In this sense we can truly say that 'machines can never make mistakes'. Errors of conclusion can only arise when some meaning is attached to the output signals from the machine. […] When a false proposition is typed we say that the machine has committed an error of conclusion. There is clearly no reason at all for saying that a machine cannot make this kind of mistake." (Turing 1950, p. 449).

Turing's errors of *functioning* and errors of *conclusion* are examined in our analysis at a finer level of granularity. We argue that a computational system *can only* make an error of *functioning* (i.e. an operational malfunction), yet, to observe that, the analysis of errors has to proceed at different levels of abstraction (LoAs). The different notions of computational errors, such as 'bugs', 'malfunctions', 'failures' or 'mistakes', are typically based on some common sense characterisation and they lack any precision. It is, therefore, important to distinguish the different notions and associated meanings in computer science and computational practice.

The paper is organised as follows. Section 2 briefly discusses the relation between miscomputation and malfunction of artefacts. In Section 3, we give some paradigmatic examples of miscomputation in designed systems and sketch an initial characterisation of miscomputation. In Section 4, we discuss three LoAs pertaining to the design of digital computational systems: the functional specification of the system, the system's design specification and the algorithm design. In Section 5, a distinction is made between algorithm implementation in software and in hardware. Section 6 describes the lowest LoA[2] where some miscomputation is bound to occur: the execution of algorithms in physical systems. In Section 7, we offer a taxonomy of the miscomputations discussed in the paper throughout the various LoAs. Section 8 concludes the paper.

2 Malfunction of Artefacts

Computational systems are different from other technological artefacts due to the tension that exists in the former case between abstract and concrete. Because of the formal and technological dimensions of computational systems, the task of analysing such systems is a multi-layered enterprise. Whilst some attention has been given to the ontology of computational objects (Turner 2013) in philosophy of computer science (Turner and Eden 2011), the methodology of their explanation has not yet been sufficiently investigated. The contribution of our paper hinges on two important philosophical debates. On the one hand, there is a long tradition of philosophical analysis of explanation of scientific practice (cf. Hempel 1965; Kitcher 1989; Strevens 2008; Woodward 2005). On the other hand, there has been an ongoing debate about malfunction in biological systems versus in human-made artefacts (cf. Franssen 2006; Millikan 1989; Neander 1995).

[2] It should be noted that, strictly, the 'lowest' LoA we analyse here is not the last one. The analysis can proceed at even lower level, such as at the quantum-physical level.

Whilst the first debate is interesting from a methodological perspective, the second focuses on the ontology of functional systems. Our taxonomy also acts as a bridge between these two debates. Abstract computational systems are susceptible to 'design' errors and physical computational systems are subject to physical noise and are, thus, prone to a different type of errors (namely, operational malfunction). From the methodological perspective, the taxonomy honours the practice of computer science, which is structured according to different LoAs.

The distinction between errors of functioning and errors of design (Turing's errors of conclusion) is also present in the philosophy of artefacts. However, the difference in nature between errors of functioning and errors of design is clearer in the context of computational systems, and it is easier to observe why operational malfunction only applies to a narrow category of errors. Analogously to computational systems that can fail to compute correctly, artefacts, in general, can fail to perform their functions for two reasons. One is indeed malfunction proper (as discussed later in the paper), but the other is a functional failure due to design mistakes. The ICE-theory[3] offers one of the most well developed accounts of technical functions including an analysis of artefact malfunctions. On the one hand, technical malfunctioning is said to come in degrees, 'from minor and major defects to outright failure' (Houkes and Vermaas 2010, p. 103). Accordingly, a basic distinction is drawn between *failing to operate as per design* and *failing to be usable*. On the other hand, another important distinction is drawn between *token*- and *type*- malfunctioning. The latter requires again an appeal to design (e.g. describing all light bulbs of a type as malfunctioning, since they all consume more energy than they *should*; (ibid)).

The evaluation of an artefact malfunction, as in the case of a miscomputation, requires an appeal to the design of the artefact. The ICE-theory, for example, requires an account of what it means for the artefact to have the *capacity* to function, which is distinguished from the *ability to exercise* such function (Houkes and Vermaas 2010, pp. 106–108). The former property is typically defined by appeal to design, whilst the latter is defined purely in terms of errors of functioning (to use Turing's terminology again). In Jespersen and Carrara (2011, p. 120), it is argued that, on the design view, 'an artefact is an F if, and only if, it was designed (hence intended) to function as an F, irrespective of its capacity to function as one'. The design view, which squares roughly with etiological theories of function, embeds a subsective understanding of the semantics of malfunctioning. On this view, '[a] malfunctioning F retains its proper function as an F, but forfeits its capacity to function as an F' (ibid).

'Malfunction' has, of course, deep roots in philosophy of biology as well, for naturally evolving organisms are also error-prone. Any adequate theory of the evolutionary source of a (teleological) function has to account for the possibility of malfunction (Perlman 2010, p. 54). Whether any insights from philosophy of technology about function and malfunction can be extrapolated to philosophy of biology remains an open question, on which we remain neutral here. The debate about the

[3] 'ICE' theory stands for Intentional, Causal-role, Evolutionist function theory. For more details on this theory, see Houkes and Vermaas (2010).

nature of biological function between the Selectionist who relates it to the history and evolutionary selection of the system concerned and the Systematicist who relates it to the actual causal role it plays for the system, remains unsettled (Perlman 2010, p. 53). The ensuing discussion focuses on the analysis of errors in computational systems.

3 Miscomputation in Designed Systems

We begin our analysis of miscomputation proper by considering an example. Suppose that under normal circumstances a computational system *S* performs the computational step C_1 at time T_1 on input I_1 producing output O_1, which feeds in as input to the next computational step C_2 at T_2 producing output O_2 at T_3. Let us further assume that an operational malfunction (at the hardware level) occurs, and yet *S* produces output O_2 at T_3. *S* performs C_1 at T_1 as before, but it then malfunctions and performs C_3 (instead of C_2) at T_2 on O_1 while still producing the same output O_2 at T_3. This is analogous to a two-step deductive argument (P→R→Q), where the first step (P→R) is valid. The second step (R→Q) is invalid and yet the overall argument (P→Q) is valid. Although the final outcome is the same as the intended one (when it functions properly), we would still say that *S* miscomputed (Fresco 2012).

This example shows how computation requires some notion of purpose in order to distinguish a proper computation from a miscomputation. A computational system can be said *to act for a purpose according to its design*. When a system fails to accomplish the purpose for which it was designed, a miscomputation can be identified. In the next section, the notion of purpose is considered explicitly as a defining element of specifications of computational systems.

But first, let us turn to some more examples of miscomputations. Consider the famous Y2K bug, which was the result of the common practice (before the year 2000) of abbreviating a four-digit representation of a year to two digits. A main reason for representing years as two digits was to conserve memory space. It was not clear whether the increment of year=99 to year=00 would be recognised by computational systems as 1900 or 2000. This problem was not limited just to software-based systems.

Another example is the famous 1994 Pentium FDIV bug. This microprocessor bug was caused by an error in a lookup table that was a part of the chip's hardware floating point divide unit. This bug was only triggered upon certain input data and the degree of inaccuracy of the result delivered depended upon the input data and the specific instruction involved (Intel 2004). Importantly, the cause of the bug was an error in a script that downloaded some quotient digit values into a hardware lookup table. That error resulted in a few lookup entries being omitted from that table.

Let us now consider the following two cases. The first one is an algorithm for multiplying two positive natural numbers.

```
1  Read the values of X and Y.
2  Product=−1.//−1 represents illegal input
3  If both X and Y are positive integers: product=X*Y.
4  Return product.
```

Code excerpt 1 An algorithm that multiplies two positive natural numbers.

Let us further assume that this algorithm is implemented in the Java programming language as follows.

```
public static int computeProduct(int factor1, int factor2) {
        int product = -1; // the value -1 represents illegal input
        if(factor1 > 0 && factor2 > 0){
                product = factor1* factor2;
        }
        return product;
}
```

Code excerpt 2 An implementation of the multiplication algorithm above in Java.

Whilst the *algorithm* for multiplying the two factors has no restriction of the result stored in Product, the *program implementation* of product in Java has an upper bound of $2^{31}-1$.[4] If the program is executed and the calculated product exceeds this value, it will not produce the expected output.

In which sense do the above examples qualify as miscomputations? We consider Piccinini's definition of a miscomputation: '[a] mechanism *m* miscomputes just in case *m* is computing a function *f* on input *i*, $f(i)=o_1$, *m* outputs o_2, and $o_2 \neq o_1$' (Piccinini 2007, p. 505). We note that this definition cannot be correct for all concrete computations. For a computational system may produce the correct output fortuitously as a result of some hardware malfunction. The first example discussed above shows that a computational system *S* produces the output O_2 despite a hardware malfunction. *S* produces the same output that would have been produced had it not malfunctioned. However, O_2 is produced by following an *incorrect* computational step (compared with the specification of *S* that we discuss in the following section). If we take $f(I_1)=O_2$, then *S* miscomputes and still gives the correct output O_2. This, we believe, shows why it is important to not only consider errors at the appropriate LoA, but also to consider other LoAs for determining whether a particular scenario is a miscomputation or not. The taxonomy that is offered below fulfils this task.

4 Purpose, Specification and Algorithm Design

'The complexity of many contemporary computational systems irrespective of their ontological nature, demands that they be treated as *physical* systems' (Turner 2011, p. 148, italics added). It is not a logical impossibility to verify the correctness of a (complex) computational system but a practical one: the more complex the system is, the less feasible it is to determine its correctness (or ensure the lack of *any* miscomputation in principle). Whatever the method

[4] See http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html#MAX_VALUE for details.

of verification is (in abstract or physical systems), in order to determine whether a certain operation of the system is incorrect, some correctness criterion (or criteria) is required. Determining whether an *artificial* system miscomputes is certainly easier than determining it for a *natural* computational system (whatever 'natural computational system' means precisely). At least in the former case, we can refer to its *purpose* and *specification*.

Whether or not some system performs the correct computation can be established objectively by examining its functional structure or design (below, we make a finer distinction between the two). It is well known that the problem of whether there exists a design, which satisfies a given set of functional requirements, is, in general, undecidable. This problem is akin to that of finding a proof for a given sentence in first order logic. On the other hand, the problem of verifying the correctness of a particular design (*ex post facto*) against a given requirements specification is decidable. The undecidability of the first problem exceeds the scope of this paper. To a first approximation, we stipulate that a system miscomputes when it does not comply with the principles made explicit by its specification(s).

Specifications of computational systems have a normative function that is central for determining the correct or wrong behaviour of these systems. '[I]t is the act of taking a definition to have *normative force* over the construction of an artefact that *turns* a mere *definition* into a *specification*' (Turner 2011, p. 140, italics added). Arguably, '[w]hether a [computational system] malfunctions is not a property of the [*system*] *itself* but is determined by its specification' (Turner 2011, p. 141, italics added). But, as we argue below, it is only errors of *conclusion* that violate the normative value of specifications. Errors of *functioning* (i.e. operational malfunctions), on the other hand, do not violate the norms set by the specification. They are the result of *noise*. This reinforces the importance of Turing's distinction between these two types of computational errors.

The characterisation of miscomputation relies on another relevant distinction that should be made in the context of *purpose*, namely between *internal* and *external* teleology. Objects that have some immanent property that makes them goal-directed can be said to be internally teleological. If an object has some goal assigned to it by some goal-conceiving agent (e.g. its designer), then it is externally teleological (Mahner and Bunge 1997, p. 368). Accordingly, a human-made artefact has an externally assigned purpose, or in other words, a purpose-by-design. Yet, it seems excessive to say that a TM only has a purpose as long as there is a purpose-attributing agent. Consider the following claim. '[A] computer on a dumping-ground has no purpose […] whatsoever, although [there may exist] somebody who retrieves it from there and […] is able to rethink the intentions of its designer, e.g. by examining its structure' (Mahner and Bunge 1997, p. 369). Once its purpose has been built in into its structure the goal of the TM remains unchanged even in the absence of any purpose-attributing agent. A specific-purpose TM is always directed to attaining a specific goal, namely, the computation of some function (for example, $f(n)=n+1$). For a similar reason, a specific-purpose computer, which was designed to compute the successor function, but somehow produces the output (n-1) for the input ($n\in\mathbb{N}^+$), is said to *miscompute* rather than just serve some other purpose.

For the present discussion, we stipulate that the purpose of a computational system is determined by its design as it is reflected in its internal structure. Since the focus of

this paper is 'conventional' (rather than natural) computational systems, we adopt the notion of purpose by design, and, hence, characterise a computational system as externally teleological. When the analysis of computational systems is extended to apply to the computational theory of mind, 'purpose' is problematic, but this need not concern us at present.

We now turn to the requirements specification of a computational system that is the highest LoA[5] where the computational problem and problem domain are defined. We refer to this level as the functional specification level (FSL). In the IT industry, a software specification (referred to as software requirements specification (SRS)) provides a complete description of the expected behaviour of the computational system to be developed. An SRS specifies what the system is supposed to do functionally. However, it may also specify some non-functional requirements, such as the system's hardware, some possible interactions with other systems, design constraints that are imposed on the implementation of the system (e.g. policies for database integrity, resource limits, etc.) or performance constraints (e.g. the speed, response time and availability of various software functions) (IEEE Computer Society 1998).

Whilst the satisfaction of functional requirements by the corresponding implementation is a crucial feature of both hardware and software verification, the process of specification has typically been done by producing a natural language description of a set of functional requirements. This introduces ambiguity that can lead to unverifiability, due to the lack of a standard machine-executable representation. The use of specification languages is common in computer science for properly describing the purpose and conduct of a system at a very general LoA. These languages are typically non-executable sets of description patterns for the properties and actions the system is supposed to have and be able to perform. Data and functions are generally translated into properties (e.g. using abstract state machines, or algebraically in Common Algebraic Specification Language, or in a model-theoretic notation using the Vienna Development Method or set-theoretically in Z, which is roughly based on the Zermelo–Fraenkel axioms for set theory).[6]

Although a specification does not (typically) specify the algorithm design of the system in question, it does limit the possible number of algorithm designs. In some cases, the functional requirements of a specification may also include processing requirements (indicating the computational steps of the system). But in the most general sense, these specifications define the system's input and output requirements. Thus, even if the processing requirements are not defined explicitly, the possible set of algorithms is constrained by the extension of the <*permitted input*, *expected output*> pairs defined.[7] A specification defines the problem(s) to be solved, the set of permitted inputs and expected outputs (pre- and post-conditions of the system), as well as supporting assumptions regarding the system to be designed for solving the problem(s) in question. The purpose of the system is to solve this problem under the given assumptions.

[5] A similar caveat applies here as in footnote 2.

[6] For an overview of the logics of many of these languages, see, for example, Bjørner and Henson (2008).

[7] Formally, the requirement for the correctness of the permitted input is expressed by a pairwise disjoint union partition of the input domain in (1) standard domain, (2) admitted exceptional domain, (3) failure domain and (4) unexpected domain. The correctness of the expected output domain is ensured by expressing (3) and (4) as empty domains as a precondition of the system.

The more fine-grained specification (often referred to as a system design description (SDD), in the IT industry) describes (at a high level) what informational states the system accesses and possibly by which instructional states it does so. We refer to this level as the design specification level (DSL). A design specification reflects a conceptualisation of the system under design that embodies its essential characteristics and demonstrates a means to fulfil the system's requirements (IEEE Computer Society 2009). The objective of a design view, which guides the SDD, is to address design constraints pertaining to the relevant system's requirements. For an SDD to be deemed correct, the complete set of design constraints needs to be addressed, and for it to be valid there should be no conflicts among the design elements. This type of specification is more than a stipulative definition of the system—it is a functional definition of the system to a varying degree of accuracy. The more general the specification is (e.g. listing only a few broad requirements), the less constrained the algorithm is.

At the FSL, only the intention(s) of the producer(s) of the (requirements) specification can be considered. For convenience, we refer to this producer as the *Architect*, and to the system designer as the *System Designer* hereinafter. This means that the inevitable gap between the FSL and DSL leaves much room for error creep already at the DSL. In this sense, the general criterion of correctness seems to be functionality-centric: 'make the system do what you want it to do' (Smith 1996, p. 416). At the FSL, the conceived functionality is simply specified by way of incomplete conditions that the system should satisfy to produce a specific operation.

When considering an algorithm design, the functional design specifications can only provide an informal measure of correctness by way of comparing the former with the latter. We refer to the level of algorithm design as algorithm design level (ADL) and to the agent(s) responsible for the algorithm design as the *algorithm designer* (who may, but need not, be the same as the system designer). To establish a formal criterion of correctness at the ADL, it is crucial to determine the model of implementation under consideration. On the one hand, one can use a subset of possible physical simulation patterns for verifying the given design (say, as specified by an SDD). A different approach, which is less used in computer practice, is known in theoretical computer science as *formal methods*. The specifications used in formal methods are well-formed statements in mathematical logic and the formal verifications are deductions in that logic (cf. Meyer 1985 and Black et al. 1996 for a basic introduction).

By way of example, consider the execution of a program p under a given set of conditions (call it N, for 'network') for a certain specification (S) of program type (p).[8] This can be formally expressed as the validity of the logical expression $N \vdash p:S$. Program termination formally corresponds to the correctness of the typing relation, in the form of type-checking (i.e. given a program p, a specification S and a network N,

[8] Here, we implicitly refer to the proof-theoretical result known as Curry–Howard isomorphism establishing a direct relation between computer programs and derivations. A valid contextual proof (or lambda term) is isomorphic to a program correctly executable in a network. This result, also known as proofs-as-program identity, is based on the interpretation of formulae and specifications as types, of which respectively proofs and programs are instantiating elements. This formal identity allows treating a running program as formal objects, whose properties—including correctness and termination—can be established by way of deductive methods. The isomorphism originates in observations by Curry (1934), Curry et al. (1958) and Howard (1980). For a systematic treatment of the issue, see Sørensen and Urzyczyn (2006).

is N⊢p:S a derivable expression?) and type-inhabitation (i.e. given a program p, is there a specification S in a network N such that N⊢p:S a derivable expression?). The benefit of formal methods is that, in principle, they provide a means to establish correctness for all possible inputs.

Our analysis shows that purpose is a conceptual, rather than an operational, criterion of correctness. The abstract characterisation of the system is made more concrete when a *particular* algorithm is designed according to the assumptions and various constraints imposed by the specification(s). A procedural elaboration of the system's specification at lower LoAs (i.e. algorithm design and algorithm implementation) is crucial, because it directly provides a criterion of correctness in terms of the right procedure(s) to be executed to produce the expected output (which as we have seen is missing at the FSL). At the ADL, one has to refer to either practical correctness of the system by using simulation patterns or a purely formal understanding of logical correctness (i.e. by using formal methods). The result is that any list of *all* possible miscomputation scenarios would be inherently incomplete. For verification methods in the former case consist in generalisations from sample cases used in simulations and in the latter case consist in abstraction from concrete aspects of the physical implementation of the system. It seems inevitable then that the analysis of miscomputation must be extended to the level of algorithm implementation (in either hardware or software).

5 Implementation of Algorithms as Programs or in Hardware

In the section above, we have considered purpose, functional specification, design specification and algorithm design—all leaving room for (different types of) error creep. We turn next to consider the level of algorithm implementation (in either software or hardware), to which we refer as AIL. The responsible agent is referred to as the *Engineer*.

Some systems compute by virtue of executing programs: a stored-program computer is an epitome of such a system. Other physical systems do not execute programs (in the classical sense used in computer science) but compute nonetheless, such as Boolean gates. Boolean gates qualify, by our lights, as trivial computational systems, yet they do not require the simple algorithm that they execute to first be converted into a program in some programming language. The algorithm of a conventional AND-gate performs a logical conjunction operation on two input data. This algorithm is executed directly by the physical gate that implements it. More complex combinational circuits, such as half and full adders, execute algorithms that are implemented in hardware by using a network of primitive Boolean gates. In the taxonomy of miscomputation that follows, we note that the implementation of algorithms as programs adds another layer of potential errors that is not present as such in the implementation of algorithms in hardware.

5.1 Software Implementation of Algorithms

The most common type of *miscomputation* in algorithm implementation in software is *compilation* errors. A compiler translates the source program (i.e. the algorithm

written in a specific programming language) into a low-level machine language of the CPU (or into assembly language in some cases) that can then be executed by the computational system. When a compilation error occurs, the compiler fails to compile at least one part of the source program. The result is a failure of the compilation process to produce the target machine code.

The first type of compile-time errors is *syntactic*. It is the result of a violation of the syntax or grammatical rules of a programming language. Some compiled languages, such as C, Java and Pascal, for example, require that programs explicitly state the data type of any variable at either the time it is declared or first used. In some interpreted languages data type checking is performed at run-time (as opposed to at compile-time), thereby a variable can refer to a value of any type, thereby making data type declarations unnecessary. Other compile-time errors occur when some part of the source program does not conform to the syntactic rules of the programming language. A simple, but common, example is a missing semicolon at the end of some statement when the programming language specifies that each statement must be terminated by a semicolon. Another example is calling an undeclared variable or function. This error is typically the result of misspelling the variable or function name.

The second type of compile-time errors is *semantic*. This type of errors is sometimes identified with logic errors (cf. Dale and Weems 2005, p. 237, 2008, p. 251), but not always (cf. Dooley 2011, pp. 297–298; Feldman and Koffman 1999, p. 109; Purdum 2012, p. 33; Shelly et al. 2006, pp. 76–77). Unlike syntax errors, semantic errors may or may not be detected by the compiler depending on the particular programming language and the compiler used. Semantic errors occur when the program complies with the grammatical rules of the programming language, but the context of some statement or expression is wrong. These errors violate the rules of meaning of the programming language. If the compiler does not identify them, the program will produce the wrong behaviour at runtime. Common examples of semantic errors are the following.

- Using a variable though it has not been initialised (e.g. for(int *i*; *i*<10; *i*++) doSomething();). A Java compiler, for instance, should detect such an error (Campione et al. 2001, p. 395).
- Passing the wrong number (or type) of parameters to a function;
- Adding a semicolon after the condition in an *if* or *while* statement in C/C++ or Java (Dooley 2011, p. 182);

Unlike syntax errors, some semantic errors may often be the result of a faulty algorithm design, and not just the result of wrongfully translating the algorithm into some programming language. A paradigmatic example is a division by zero (Dale and Weems 2005, p. 237). Yet, this is where the distinction between *semantic* and *logic* errors is blurry.

The next type of *miscomputation* at the present LoA is *logic* errors. In the literature they are characterised as '[errors that occur] when a program does not behave as intended due to poor design or incorrect implementation of the design' (Shelly et al. 2006, p. 77). This characterisation shows that this type of errors is hard to discern from semantic errors. The distinction is indeed subtle. Whilst semantic errors 'reflect a bending of the syntax rules of the language […,] logic errors […] are the result of

design errors the programmer makes when manipulating […] data' (Purdum 2012, p. 297). It is not clear that this is always the case. A program code, such as

```
if (x<0 && x>0){
        doSomething();
} else {
        doSomethingElse();
}
```

Code excerpt 3 A Java program code with an if-else conditional statement.

does not manipulate data, yet it qualifies as a logic error, for regardless of the value of *x*, the statement 'doSomething();' will never be executed when the program is executed. We turn next to a brief discussion of errors in algorithm implementation in hardware.

5.2 Hardware Implementation of Algorithms

As said above, some algorithms are implemented directly in hardware. Errors identified at the present LoA may be collectively referred to as hardware design faults. Unlike software that is not subject to physical noise (e.g. radiation, friction, power spikes and temperature influence), hardware components are extremely vulnerable to it. As indicated by Moore's law, the number of transistors on integrated circuits increases as time progresses, thereby leading to the growing design complexity of hardware circuits. When this complexity is augmented with random noise signals, the possible types of miscomputation in hardware grow rapidly. There are many different factors that may induce hardware errors at runtime, which must be taken into account at the design phase. A complete analysis of all the possible types of hardware design errors exceeds the scope of this paper. We only make some observations that characterise hardware design, but do not typically apply in the case of algorithm implementation in software.

The increasing demand for greater performance, complex functionality and exponential reduction in size of computer hardware (in accordance with Moore's law) has resulted in the functional test generation being widely acknowledged as the bottleneck of the hardware design phase. The main focus of this testing approach is to generate test vectors that can verify the complex functionality of and interaction between multiple design units. This is commonly done by generating millions of random test vector sets. But this random test generation cannot guarantee the coverage of all possible functionalities, particularly, in complex designs (Hari et al. 2008, p. 408). The traditional worst-case design methodology becomes infeasible partly due to energy overhead and the required a priori knowledge of all possible error sources at design time. As a result, hardware design has to cater for the correction or mitigation of possible errors (May et al. 2008, p. 456). It is, therefore, hard to classify all noise-induced errors as faulty requirement specification of the hardware system. But certainly some assumptions

should be made at the level of system requirements, such as the expected temperature, range of operation and voltage ranges.

Reliability and fault tolerance are crucial in hardware design. A common technique for tolerating hardware faults is hardware redundancy by replicating all hardware components that need to be introduced into the system to overcome possible operational malfunctions. These replicated components are superfluous if no faults occur, and their removal does not diminish the system's computing power in the absence of faults (Williams et al. 2003, p. 126). Another technique is the introduction of software redundancy that would otherwise not be needed in fault-free computation. Even computers that recover from faults mostly by hardware means use programs to control fault recovery. The software recovery design depends on the type of possible operational malfunction that is expected (Williams et al. 2003, p. 127).[9]

By way of closing this section, we note that, unlike the verification of algorithm implementation in software, the ever increasing complexity of hardware circuits makes it impractical, if not impossible, to verify every circuit on a breadboard. Hardware description languages (e.g. Verilog, VHDL and SystemC) make possible the verification of digital circuits' functionality before fabricating them on a chip. These languages play a role in providing both a descriptive and a normative reading of the system in question, offering assertions of properties and cycle behaviours to be satisfied by way of sequential or conditional expressions, enriched by temporal operators. Properties in this context are concise, declarative and unambiguous specifications of the desired system behaviour that are used to guide the verification process (IEEE Computer Society 2005). These languages allow the elimination of most design bugs and use a very abstract level of description of algorithm implementation in hardware without choosing a specific fabrication technology (Palnitkar 2003). Still, it is practically impossible to anticipate all possible error conditions at runtime.[10]

6 Execution of Algorithms

In this section we discuss the algorithm execution level (hereafter, *AEL*). By the execution of algorithms we mean the actual computational process in real-world systems. Typically, any error that was introduced and not corrected at the higher LoAs is bound to manifest itself at runtime under the appropriate conditions. Program runtime errors are only detected when the program is executed on some physical computational system. These are errors in either logic or arithmetic operations, but they could also occur as the result of hardware failure. Typical examples are attempts

[9] Error handling has a proper counterpart in the logical design of programs that might fail to provide their specified service (particularly in distributed architectures). This is typically done by providing rigorous definitions of crucial concepts in design: the specification, the semantics of the program involved, its correctness, the possible exceptions and the cases of fault, failure and error. At each LoA of the program's semantics, an appropriate handling procedure is defined. For a comprehensive analysis of exception handling design, see Lee and Anderson (1990) and Buhr et al. (2002).

[10] For more on simulation-based and formal verification techniques of hardware design the reader is referred to Lam (2005).

to divide a number by 0, running out of memory (unlike a TM with an infinite tape), invalid input (unexpected input data for which there is no defined operation in the program) and software race conditions where separate computer processes depend on some shared unsynchronised state.

These errors (if unhandled) cause the execution of a program to terminate abnormally, because the operation attempted is impossible to carry out. Since runtime errors can be caused either by the program, an input to the program or a hardware failure, they can be anticipated but hard to avoid. Where such errors can be anticipated, corrective steps should be taken by the Engineer to avoid either an unexpected behaviour or an abnormal termination of the program. But these errors cannot be completely avoided. Consider a program that expects to read a file that was stored on the filesystem, only that the file is either corrupted (e.g. due to some hardware failure) or nonexistent (e.g. another process may have deleted it).

A hardware failure is simply a physical failure of some component in the computer (that may be introduced by a faulty design). It can be a computational component, such as an AND-gate, but it can just as well be a non-computational component, such as a cooling system. Hardware components typically contain error detection mechanisms that can detect when an error condition exists. Hardware errors can be classified as either permanent or transient as a function of their duration (Williams et al. 2003, p. 126). Permanent errors are caused by solid failures of components. They are easier to detect but typically require the use of more drastic correction techniques than their counterparts. Transient errors are intermittent failures that prevent the normal operation of a unit for only a short period of time, which is typically not long enough to allow detection and testing as in the case of permanent errors. Hardware errors can also be classified as either correctable or uncorrectable errors (MSDN 2012). A correctable error is an error condition that can be corrected by the hardware or software by the time that the operating system is notified about its presence. An uncorrectable error is an error condition that cannot be similarly corrected.

7 A Taxonomy of Miscomputations

The analysis presented above encompasses Turing's errors of functioning and errors of conclusion. Errors of *conclusion* can be viewed as the parent category for various errors that may be induced at various LoAs starting with FSL all the way 'down' to AIL (in either software or hardware), inclusive. They are typically associated with agents, such as the Architect, the System Designer, the Algorithm Designer or the Engineer who is/are responsible for '[attaching] some meaning […] to the output signals from the machine' (Turing 1950, p. 449). On the other hand, errors of *functioning* are confined to operational malfunctions at AEL. The taxonomy provided below maps each LoA with one of four different kinds of error: mistakes, failures, slips (see Primiero 2013) and operational malfunctions. The taxonomy is summarised in Table 1.

Table 1 This table summarises the various LoAs discussed and the different possible errors

Agent	LoA	Conceptual	Material	Performable	Error type
Architect	FSL	Contradicting requirements			Mistake
System designer	DSL	Invalid design (mismatch with the FSL)			Mistake
System designer	DSL		Incomplete design (relative to the FSL)		Failure
Algorithm designer	ADL	Invalid routine (a violation of either well-formedness or consistency rules)			Mistake
Algorithm designer	ADL		Incorrect routine		Failure
Engineer	AIL		Syntax error		Slip
Engineer	AIL	Semantic error			Slip
Engineer	AIL	Logic error			Mistake
Engineer	AIL (hardware)	A wrong selection of hardware (e.g. using an AND-gate for a logical disjunction operation)			Mistake
Engineer	AIL (hardware)		Wrong implementation (e.g. the 1994 Pentium FDIV bug)		Failure
N/A	AEL			Hardware and/or software error	Operational malfunction

In our analysis, we consider three categories of miscomputations.

1. Miscomputations due to the breaching of *validity conditions*[11] are classified as *conceptual*.
2. Miscomputations due to the breaching of *correctness conditions*[12] are classified as *material*.

[11] Validity conditions here refer to both validity and satisfiability criteria on the specification. The logical validity of a specification is typically given by formulating the behavioural description of a program in (propositional) conjunctive normal form. One can check whether for each disjunctive atom in the same conjunct the negation is also available, and if so, then logical validity is proven. Otherwise, logical validity cannot be proven. Satisfiability is the weaker logical counterpart that only requires one positive evaluation for an atom. The class of propositional formulas in conjunctive normal form has a straightforward check for syntactic validity, but a hard one for satisfiability.

[12] Correctness conditions here refer to the relation of logical, total and partial correctness between the program and its specification. Logical correctness refers strictly to the relation that the syntax of the expression (program) bears to the construction rules allowed by the alphabet and language in question. An incorrectly formulated program will not yield an output that conforms to its specification. More generally, an algorithm is said to be *totally correct*, if it outputs the value required by its specification and halts. It is said to be *partially correct*, if no claim is made about its termination.

3 Miscomputations due to the breaching of *physical conditions* are classified as *performable*.

The first and second categories of miscomputation apply to FSL 'down' to AIL. At the FSL, a specification that contains two contradicting requirements fails to satisfy validity conditions. For example, a global requirement might be that the computational system continues operating even when database connectivity is lost (say, by using the file system instead for temporary storage). But another contradicting requirement pertaining to some critical subsystem would be such that it implies a strong coupling with the database (thereby, failing to operate in the absence of database connectivity). At the ADL, AIL (and sometimes DSL), a validity condition concerns how the routine selected and the task required match. For example, at the ADL and AIL, unbounded recursion or unbounded while loops in *real-time* systems[13] are the wrong fit because of the mission criticality and response-time sensitivity of these systems.

Correctness conditions refer to the DSL, ADL and AIL. They refer to the *structure* of the selected routine in consideration of the output to be generated. At the DSL, consider, for example, an interaction among three components of the system, *S1*, *S2* and *S3*. *S1* (correctly) invokes some subroutine resulting in *S2* being called to process this request. Yet, if *S3*, which is the target consumer of the output generated by *S2*, does not receive this output, the overall interaction is said to not satisfy correctness conditions.

At the ADL, we may consider a simple TM, *T*, that fails to satisfy some correctness conditions. Given any tuple as input, *T* replaces it with a single '1' on its tape as output (i.e. *T* computes $f(x_1 \ldots x_n)=1$). Its configuration would be defined by the following set of six quadruples of the form (current state, symbol scanned, operation and next state).

1. STATE1-READ1-WRITE0-STATE1
2. STATE1-READ0-MOVERIGHT-STATE2
3. STATE2-READ1-WRITE0-STATE1
4. STATE2-READ0-MOVERIGHT-STATE3
5. STATE3-READ1-WRITE0-STATE2
6. STATE3-READ0-WRITE1-STATE4

STATE4 is the Halting State. Suppose that a second TM, *T*', whose configuration is given by a similar set of quadruples only that the fifth one does not move the TM back to STATE2, but rather moves it directly forward to STATE4. In other words, once *T*' reaches STATE3 it scans a symbol, writes either a '0' or a '1' and moves to the Halting state. *T*' would still be valid, but incorrect for computing $f(x_1 \ldots x_n)=1$. After having deleted the first input argument, *T*' would either delete the first symbol of the possible second input argument and halt (e.g. for '111' only the first '1' would be deleted) or find no such input and write 1. *T*' would not start the initial cycle again to delete all entries in the second input argument.

[13] These are systems that are used to control and monitor physical processes and are rigidly constrained in terms of their response time and/or the validity of their data.

Similarly, at the AIL, the program has to be well formed, but correctness conditions are not limited only to the grammatical rules of the particular programming language. Consider the following example in Haskell pseudo-code:

```
let list l := [2,3]
let list m := [4,5]
l ++ m := [2,3,4,5]
in 1 : [l++m] = [1,2,3,4,5]
```

Code excerpt 6 A pseudo code in Haskell using the '++' and ':' operators.

The ++ operator appends one list to another and the : operator adds elements to the head of a list. Hence, while the statement

```
in l : [[4,5]] = [2,3,4,5]
```

is a legal regular expression, the statement

```
in [2,3] : [4,5] = ?
```

is illegal. For the latter attempts to add a list to integer elements that are not a list, thereby resulting in a non-homogeneous object (see, e.g. Davie 1992, pp. 24–25). This in turn induces a compile time error, due to a type mismatch between [Int] and a list. We note that the Y2K bug discussed above falls in this category too. For, at the AIL, the wrong representation of the year data was used.

We call *mistakes* those conceptual errors that may occur at the FSL, DSL, ADL and AIL. They correspond to design-level errors, induced by either the System Designer's interpretation of the Architect's intention or the Engineer's interpretation and translation of the system's design and/or purpose. If the purpose of the system were only to operate under some circumstances, but not others, then it would be wrong to classify that system as miscomputing under different circumstances when it does not produce the expected result. Mistakes could occur due to a miscomprehension of the requirement(s) formulation by the system designer, a semantic error due to a bad translation of algorithm design into some programming language or a logic error due to a poor algorithm design. They could also occur at the hardware implementation level, where they are introduced at the level of hardware system design or by a wrong matching between software and hardware.

We call *failures* those material errors occurring at the DSL, ADL and AIL. They can occur at the DSL when a conceptually correct system requirements specification is wrongly formulated as a system design specification. Consider a software system intended to track and manage goods requests, goods production, their delivery and payments. Assume that the system designer receives a valid and correct system requirements specification. Let us further assume that the system design specification produced is valid insofar as no contradictory actions are specified. However, the design specification is not correct if, for example, no routines are included for the tracking and management of payments. Failures can occur, at the ADL, when, say, a correct and valid SDD is translated into a faulty algorithm (e.g. the TM *T*' discussed

above for computing the function f($x_1 \ldots x_n$)=1). An example of a failure at the AIL is the 1994 Pentium FDIV bug discussed above. The error in the script, which downloaded some quotient digit values into a hardware lookup table, occurred whilst implementing the algorithm directly on the chip.

The third category of performable miscomputations applies to the AEL only. We call a computational system error under physical conditions an *operational malfunction*. Malfunctioning at the AEL corresponds simply to the well-known cases of errors at runtime by either hardware failures or a combination of software and hardware failures. This family of miscomputations can occur despite satisfying validity and correctness conditions. Malfunctions may be introduced at higher LoAs but manifest themselves at runtime. Consider the case of a CPU meltdown at runtime. This meltdown may be the result of some non-computational component failing to perform its designated operation (e.g. a breakdown of the cooling system due to wear and tear or a faulty design). It can also be the result of the system's exposure to physical processes or forces, such as radiation, heat or friction. It can also be induced by faulty hardware and/or software design, such as a poor choice of physical material for electronic components that dissipate excessive heat in normal operation or faulty software design incorrectly leading to a constant voltage feed in some chip.

The last type of errors is a slip. Slips are not produced by a wrong categorisation, a wrong system design, or a faulty translation of algorithm design at AIL. They apply to either syntax or semantic errors that are induced at the level of algorithm implementation in software. Notice that any of these errors would be characterised as a slip, only if it occurs notwithstanding the intention of the engineer to avoid blunders in translating a valid and correct algorithm design into the corresponding code (i.e. moving from ADL to AIL). The engineer fails to perform those tasks correctly, although s/he knows the syntactic and semantic rules of the particular programming language for implementing the algorithm design.

Is our taxonomy general enough to include abstract computational systems? TMs, for example, are seen as perfect abstractions of conventional digital computers. It is possible to argue that TMs, by definition, do not miscompute. But this claim is true, only if by *miscomputation* we mean operational malfunction (or errors of functioning). TMs are analysable at the ADL and as such they are susceptible to any errors that may be induced at that LoA or the ones above (i.e. FSL and DSL). A specific TM, M, whose input tape is inscribed with a string s, which is not well formed, might seem similar to the invalid input runtime error mentioned above. If M receives s as input and halts, but not in accepting state, does it indeed miscompute for s? Hardly. M computes correctly. It is simply undefined for computing on input s and so M halts but does not accept s.

8 Conclusions

This paper provides a systematic taxonomy of miscomputations in conventional computational systems. We have tracked errors that may be introduced at various levels starting from the functional description of the system that defines its purpose through algorithm design and implementation down to the lower level of physical

computation. In this taxonomy we identify four types of miscomputation: mistakes, slips, failures and operational malfunctions. Arguably, the first three types of miscomputation are introduced at the levels above the physical computation. The last one only occurs at the physical level (AEL).

The philosophical debate on malfunctioning thus far has been deprived of a systematic way to account for different ways of explaining what it actually means for a system to behave incorrectly. We believe that the present taxonomy is a backdrop for further research, at least in the case of conventional computational systems. How much this taxonomy extends to other human-made artefacts or biological systems remains to be established.

We note that it is only what we call 'operational malfunction' at AEL that may be relevant in the case of computational theories of mind. For this type of miscomputation need not presuppose the notion of external purpose. Another curious result is that by reserving the notion of miscomputation to apply only to operational malfunctions, the cause of the miscomputation is often the physical substrate that is contingent to the computational process itself.

Acknowledgments We thank Marty Wolf for a useful discussion on miscomputation in physical systems. We also thank the two referees for their constructive comments and suggestions that have improved the paper. This research was conducted while Giuseppe Primiero was a Post-Doctoral Fellow of the Research Foundation Flanders (FWO) at the Centre for Logic and Philosophy of Science, Ghent University, Belgium. He gratefully acknowledges the financial support.

References

Bjørner, D., & Henson, M. C. (2008). *Logics of specification languages*. Berlin: Springer.

Black, P. E., Hall, K. M., Jones, M. D., Larson, T. N., & Windley, P. J. (1996). A brief introduction to formal methods (pp. 377–380). IEEE. doi:10.1109/CICC.1996.510579.

Buhr, P. A., Harji, A., & Russell Mok, W. Y. (2002). Exception handling. *Advances in Computers, 56*, 245–303. Elsevier.

Campione, M., Walrath, K., & Huml, A. (2001). *The Java tutorial: short course on the basics*. Boston: Addison-Wesley.

Curry, H. B. (1934). Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences of the United States of America, 20*(11), 584–590.

Curry, Haskell B., Feys, R., & Craig, W. (1958). *Combinatory logic*. *Vol. 1*. Amsterdam: North-Holland.

Dale, N. B., & Weems, C. (2005). *Programming and problem solving with C++*. Boston: Jones and Bartlett Publishers.

Davie, A. J. T. (1992). *An introduction to functional programming systems using Haskell*. Cambridge: Cambridge University Press.

Dooley, J. (2011). *Software development and professional practice*. New York: Apress. doi:10.1007/978-1-4302-3802-7.

Feldman, M. B., & Koffman, E. B. (1999). *Ada 95: problem solving and program design*. Reading: Addison-Wesley.

Franssen, M. (2006). The normativity of artefacts. *Studies in History and Philosophy of Science Part A, 37*(1), 42–57. doi:10.1016/j.shpsa.2005.12.006.

Fresco, N. (2012). Concrete digital computation: competing accounts and its role in cognitive science. University of New South Wales, Sydney, Australia. Retrieved from http://handle.unsw.edu.au/1959.4/52578.

Hari, S. K. S., Konda, V. V. R., Kamakoti, V., Vedula, V. M., & Maneperambil, K. S. (2008). Automatic constraint based test generation for behavioral HDL Models. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 16*(4), 408–421. doi:10.1109/TVLSI.2008.917424.

Hempel, C. G. (1965). *Aspects of scientific explanation: and other essays in the philosophy of science*. New York: Free Press.
Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). *Introduction to automata theory, languages, and computation*. Boston: Addison-Wesley.
Houkes, W., & Vermaas, P. E. (2010). *Technical functions: on the use and design of artefacts*. Dordrecht: Springer.
Howard, W. A. (1980). The formulae-as-types notion of construction. In J. P. Seldin & J. R. Hindley (Eds.), *(pp* (pp. 479–490). London: Academic.
IEEE Computer Society. (2005). IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2005*, 0_1–143. doi:10.1109/IEEESTD.2005.97780.
IEEE Computer Society. (2009). IEEE Standard for Information Technology—Systems Design—Software Design Descriptions—Redline.
IEEE Computer Society. (1998). *IEEE recommended practice for software requirements specifications*. New York: Institute of Electrical and Electronics Engineers.
Intel. (2004). FDIV replacement program: statistical analysis of floating point flaw. Available from http://www.intel.com/support/processors/pentium/sb/CS-013005.htm. Accessed 28 January 2012.
Jespersen, B., & Carrara, M. (2011). Two conceptions of technical malfunction. *Theoria, 77*(2), 117–138. doi:10.1111/j.1755-2567.2010.01092.x.
Kitcher, P. (1989). Explanatory unification and the causal structure of the world. In P. Kitcher & W. C. Salmon (Eds.), *Scientific explanation* (pp. 410–505). Minneapolis: University of Minnesota Press.
Lam, W. K. C. (2005). *Hardware design verification: simulation and formal method-based approaches*. Upper Saddle River: Prentice-Hall.
Lee, P. A., & Anderson, T. (1990). *Fault tolerance, principles and practice*. Wien: Springer.
Mahner, M., & Bunge, M. (1997). *Foundations of biophilosophy*. Berlin: Springer.
May, M., Alles, M., & Wehn, N. (2008). A case study in reliability-aware design: a resilient LDPC code decoder (pp. 456–461). IEEE. doi:10.1109/DATE.2008.4484723.
Meyer, B. (1985). On Formalism in Specifications. *IEEE Software, 2*(1), 6–26. doi:10.1109/MS.1985.229776.
Millikan, R. G. (1989). In defense of proper functions. *Philosophy of Science, 56*(2), 288–302.
MSDN. (2012). Hardware Errors and Error Sources. Microsoft. Available from http://msdn.microsoft.com/en-us/library/windows/hardware/ff559382%28v=vs.85%29.aspx.
Neander, K. (1995). Misrepresenting & malfunctioning. *Philosophical Studies, 79*(2), 109–141. doi:10.1007/BF00989706.
Palnitkar, S. (2003). *Verilog HDL: a guide to digital design and synthesis*. Upper Saddle River: SunSoft Press.
Perlman, M. (2010). Traits have evolved to function the way they do because of a past advantage. In F. J. Ayala & R. Arp (Eds.), *Contemporary debates in philosophy of biology* (pp. 53–72). Malden: Wiley-Blackwell.
Piccinini, G. (2007). Computing mechanisms. *Philosophy of Science, 74*(4), 501–526. doi:10.1086/522851.
Primiero, G. (2013). A Taxonomy of Errors for Information Systems. *Minds and Machines*. doi:10.1007/s11023-013-9307-5.
Purdum, J. J. (2012). *Beginning object oriented programming with C#*. Hoboken: Wiley.
Shelly, G. B., Cashman, T. J., Starks, J., & Mick, M. L. (2006). *Java programming: comprehensive concepts and techniques*. Boston: Thomson/Course Technology.
Smith, B. C. (1996). Limits of correctness in computers. In R. Kling (Ed.), *Computerization and controversy: value conflicts and social choices*. San Diego: Academic.
Sørensen, M. H., & Urzyczyn, P. (2006). *Lectures on the Curry–Howard isomorphism*. Amsterdam]: Elsevier.
Strevens, M. (2008). *Depth: an account of scientific explanation*. Cambridge: Harvard University Press.
Turing, A. M. (1950). Computing machinery and intelligence. *Mind, 59*(236), 433–460.
Turner, R. (2011). Specification. *Minds and Machines, 21*(2), 135–152. doi:10.1007/s11023-011-9239-x.
Turner, R. (2013). Programming languages as technical artifacts. *Philosophy & Technology*. doi:10.1007/s13347-012-0098-z.
Turner, R., & Eden, A. (2011). The Philosophy of Computer Science. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2011.). Available from http://plato.stanford.edu/archives/win2011/entries/computer-science/.
Williams, T. J., Schaffer, E. J., & Rohr, A. (2003). *Redundant and voting systems. In Instrument engineers' handbook*. Boca Raton: CRC Press.
Woodward, J. (2005). *Making things happen: a theory of causal explanation*. New York: Oxford University Press.