Philos. Technol.
DOI 10.1007/s13347-016-0235-1

RESEARCH ARTICLE

Explaining Engineered Computing Systems' Behaviour: the Role of Abstraction and Idealization

Nicola Angius[1] · **Guglielmo Tamburrini**[2]

Received: 10 May 2016 / Accepted: 4 September 2016

Abstract This paper addresses the methodological problem of analysing what it is to explain observed behaviours of engineered computing systems (BECS), focusing on the crucial role that *abstraction* and *idealization* play in explanations of both correct and incorrect BECS. First, it is argued that an understanding of explanatory requests about observed miscomputations crucially involves reference to the rich background afforded by hierarchies of functional specifications. Second, many explanations concerning incorrect BECS are found to abstract away (and profitably so on account of both relevance and intelligibility of the *explanans*) from descriptions of physical components and processes of computing systems that one finds below the logic circuit and gate layer of functional specification hierarchies. Third, model-based explanations of both correct and incorrect BECS that are provided in the framework of formal verification methods often involve idealizations. Moreover, a distinction between *restrictive* and *permissive* idealizations is introduced and their roles in BECS explanations are analysed.

Keywords Philosophy of computer science · Miscomputation · Explanation · Abstraction · Idealization

1 Introduction

Engineered computing systems form a vast class of physical systems, comprising laptops and personal computers, high-performance computing systems and worldwide

✉ Nicola Angius
nangius@uniss.it

Guglielmo Tamburrini
guglielmo.tamburrini@unina.it

[1] Department of History, Human Sciences, and Education, University of Sassari, Sassari, Italy

[2] Department of Electrical Engineering and Information Technologies, University of Naples Federico II, Naples, Italy

Published online: 01 October 2016

networks of other general purpose machines, controllers of vending machines and special purpose computing devices embedded into smartphones, household appliances, robots and myriads of other hybrid systems comprising both computing and non-computing subsystems. Computer scientists analyse the behaviour of computing systems before releasing them into the market; and after release, they monitor their behaviour for a wide variety of purposes. In particular, computer scientists regularly engage in the activity of evaluating whether the behaviour of engineered computing systems (BECS) conforms to, or fails to comply with, the various requirements set out by their users, programmers, producers and other stakeholders. Accordingly, explaining BECS is a pervasive activity in computer science.

The methodological problem of analysing what it is to explain BECS has received relatively scarce attention in the philosophy of computer science.[1] Interestingly, the philosophy of computer science entry (Turner 2014a) of the *Stanford Encyclopedia of Philosophy* mentions the word "explanation" only once in connection with so-called inferences to the best explanation. Fresco and Primiero (2013) remark that "whilst some attention has been given to the ontology of computational objects in philosophy of computer science, the methodology of their explanation has not yet been sufficiently investigated" (p. 255). Notable efforts to fill this gap include an analysis of computer science explanations as causal-mechanistic explanations (Piccinini 2007; Piccinini and Craver 2011; Piccinini 2015) and an analysis of incorrect executions of computing systems and their explanations (Fresco and Primiero 2013; Floridi et al. 2014). This article aims to contribute to analyses of explanation in computer science with an examination of the crucial role that *abstraction* and *idealization* play in explanations of both correct and incorrect executions of engineered computing systems. Moreover, it is worth pointing out that the ensuing analysis is exclusively concerned with *engineered* computing systems. It is beyond the scope of this article to address the question of how behaviours of computing systems that are not human artefacts are best explained.

The central role of abstraction in computer science explanations is highlighted by means of a straightforward example from the intricate landscape of BECS. This example concerns miscomputations by a general purpose digital computer running off-line a stored program written in some high-level programming language. Explanations concerning incorrect behaviours of this computing system are usually advanced against the background of *what-how hierarchies* of functional specifications (Sect. 2). At the top layer of these hierarchies, one finds user specifications about what the computing system is expected to do. These functional specifications are broken down into sets of progressively more elementary specifications as one moves downward along the what-how hierarchy.

The bottom layer in what-how hierarchies provides a description of physical components of the computing system (e.g. transistors) and a description of the processes they engage into that is couched in the language of physical theory. It turns out that correct, parsimonious, and intelligible explanations of miscomputations are in many circumstances profitably advanced without mentioning descriptions of physical entities and processes that one finds at this bottom layer of the what-how hierarchy (Sect. 3). The crucial role of abstraction in explanation can be expressed in the language of mechanistic approaches to explanation (Glennan 1996; Machamer et al. 2000; Bechtel and Abrahamsen 2005) by stating that in many circumstances *mechanism schemata* enable one to achieve correct, parsimonious, and

[1] This neglect is at odds with a rising interest in methodological analyses of explanation in other areas of technological inquiry (see for instance Pitt 2011 and van Eck 2015).

intelligible explanations of miscomputations without having to fill in their functionally characterised causal roles by making direct reference to specific physical entities and processes (Boone and Piccinini 2016).

Clearly, there are many other BECS explanation problems where an adequate *explanans* cannot be identified by abstracting away from descriptions of physical components and processes of computing systems that one finds at the bottom layer of what-how hierarchies. A significant case in point are explanations of incorrect BECS arising from hardware malfunctions with respect to the functional specifications of logic circuits and gates, where selective reference must be made to physical role fillers and their processes. However, parsimonious causal models abstracting from structural details concerning the continuous trajectories in the state spaces of the physical role fillers are to be preferred there too (Sect. 4).

Abstractions come with idealizations in some classes of BECS explanations. Idealization is the practice of representing a target system *counterfactually* with the explicit aim of simplifying the examination of some phenomenon pertaining to such system. Idealizations are usually introduced by either listing some set of ceteris paribus *clauses* that are assumed to hold (Cartwright 1989) or by directly including false assumptions into a model (Nowak 1979; McMullin 1985). Idealization practices are as such distinct from abstractions, in that the latter involve the removal of data that are not required to examine the target phenomenon without necessarily introducing distortions in its representation (Cartwright 1989; Jones 2005). The combination of abstractions and idealizations in BECS explanations is investigated here in the context of reactive computing systems interacting with their environment (Sect. 5). In particular, model-based explanations of both correct and incorrect BECS that are provided on the basis of formal model checking (Baier and Katoen 2008) are often found to involve significant idealizations. There, the observed behaviour of some reactive system S satisfying a certain property Q is explained on the basis of an abstract model of S involving spurious computation paths, that is, paths which do not represent potential executions of system S. The presence of spurious computation paths in the model does not jeopardise the correctness of the explanation unless they correspond to counterexamples showing a violation of the checked property Q, often called *false negatives*. This paper investigates the combined effect of abstraction and idealization, which prevents some downward expansions of what-how hierarchies: while abstraction hides elements of lower-level descriptions, idealization makes the inclusion of those elements impossible.

2 Specifications and Their What-How Hierarchies

Why was an incorrect output observed in this particular run of program P on personal computer C?

This question expresses an explanation request about an observed behaviour of personal computer C executing (or running) program P. This explanation request presupposes the existence of *norms* for executions (or runs) of program P on C insofar as the observed behaviour is qualified as incorrect. Therefore, in order to understand what it is to answer adequately the above question, a preliminary task is to identify the behavioural norms that are presupposed in the explanation request.

Prospective users are a chief source of computing systems' behavioural norms. Typically, users ask programmers to fulfil some of their goals and intentions. Thus,

for example, bank executives express the interests of both their company and its clients when they require that an e-banking program must always verify the PIN of bank account holders before accepting and processing transaction orders. In explaining why some executions of an e-banking program P conform with (or deviate from) user requirements one is *ipso facto* explaining why certain human goals and intentions are (or are not) fulfilled in those runs of P.

Requirements on computer programs that are advanced by users, programmers, producers, and various other stakeholders are usually called *specifications* (Turner 2011). Specifications emerge at various stages of the composite process of designing, developing, programming, testing, and revising computing systems. Programmers expand and unfold user specifications through iterated cycles of program development, testing, and revision. In the course of this process, one may even undertake to modify user specifications that turn out to be difficult or even impossible to comply with (Primiero and Raimondi 2015). Deviations from user requirements detected in some run of program P count as *failures* of P *relative to* that particular requirement (Fresco and Primiero 2013). More generally, the entire set of stakeholder requirements on runs of P determines whether in any given execution there are failures of P.[2]

Usually, user specifications concern *what* is to be accomplished (or avoided) without saying much about *how* this is to be done. Accordingly, programmers must choose one among the alternative courses of action that are available to fulfil user specifications. Thus, by selecting Java, Pascal, C++, or some other high-level programming language L for writing a program, programmers introduce additional constraints on how to fulfil user intentions, including the identification of the primitive instructions that the virtual machine associated to L can carry out. The primitive instructions of L contribute to determine *how* programmers intend to fulfil the *what* expressing user intentions and goals. In turn, the programmer's "how" with respect to some user "what" becomes an additional "what" specification for the tasks of translating into machine language, and eventually running on some computing system, the high-level instructions of L figuring in P.

In computer architecture textbooks, one finds several *layers* or *levels* of functional organization descriptions, jointly forming a hierarchy of layers induced by a binary what-how relationship. Tanenbaum (2006, pp. 2–8) describes computing systems with as many as six hierarchically organised layers. At the bottom level, one finds descriptions of logical gates and circuits. At a higher level, one finds a description of the microarchitecture, which includes functional specifications of registers forming a local memory and of the distinguished circuit called arithmetic logic unit (ALU). Still ascending in the stratified architecture, but well before getting at the top layer of programs written in high-level programming languages, one finds the so-called instruction set architecture (or ISA) layer, describing instructions that are, on the one hand, more elementary than those that are written in any high-level programming language and, on the other hand, less elementary than assembly language instructions.

[2] Floridi et al. (2014) distinguish between a *dysfunction*, an artefact's behaviour not complying with user specifications, and a *misfunction*, an artefact's behaviour which does comply with its specifications but is nevertheless prone to bring about undesired side effects. The authors emphasise how "the misfunction of an artifact token may be due to a dysfunction of some component" (p. 1209). Accordingly, explaining a misfunction of computing systems amounts to explaining a dysfunction of some given component of the system on the basis of the relevant specifications which ought to be fulfilled by such component.

Turner (2011; 2014b) points out that the descriptions one finds at each layer in what-how hierarchies are to be identified with *functional prescriptions* for computing systems. For example, any Pascal program is a text expressing must-prescriptions for the behaviour of a computing system running that program. And these must-prescriptions bring with them a cascade of additional must-prescriptions for the same computing system at lower layers in the hierarchy. One should be careful to note that the must-prescriptions one finds at any one of these layers are solely concerned with *functionally characterised properties* of computing systems (Turner 2014b). As we shall see, it turns out that abstraction from descriptions of physical components and processes of computing systems that one finds below the logic circuit and gate layer plays a crucial role in many BECS explanations. Indeed, abstractions of this sort enable one to eliminate irrelevant causal information and to achieve greater intelligibility without losing in explanatory force. Let us examine in some detail this role of abstraction in connection with the problem of explaining some given incorrect BECS.

3 Abstraction in Explanations of Incorrect BECS

Consider some program P which is supposed to compute on a personal computer the factorial function $n!$, whose recursive definition is given by the following equations:

$$0! = 1$$
$$(n+1)! = (n+1) * n!$$

A programmer may decide to expand this specification into a program by means of the following pseudo-Pascal code P:

```
1  BEGIN
2    read (n);
3    i := 0;
4    f := 1;
5    WHILE i < n DO
6      BEGIN
7        i := i + 1;
9        f := f * i;
10     END
11   write f
12 END
```

P expresses functional must-prescriptions that are inherited at lower levels in the what-how hierarchy which is induced by P and includes the assembly language code layer. In the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture (Patterson and Hennessy 2013), this layer is characterised by 32 registers. Of these

$\$s0$, $\$s1, \ldots, \$s7$ are registers for program variables, $\$t0, \$t1, \ldots, \$t9$ are additional temporary registers that are needed to translate a high-level language program into an assembly language program, and *$zero* is a register which is always set to zero. Suppose the compiler, while translating the program for the factorial function, assigns to register $\$s0$ and i to register $\$s1$. Then, line 5 of P is translated into the following assembly language instructions:

$$5.1 \quad slt \;\; \$t0, \$s1, \$s0$$
$$5.2 \quad beq \;\; \$t0, \$zero, L1$$

MIPS computes the value of the inequality condition by means of instruction *slt* ("set on less than") which is composed of three fields: the first field denotes a temporary register which is set to 1 in case the value contained in the register in the second field is less than the value assigned to the register in the third field, and to 0 otherwise. Assembly instruction 5.1 above sets $ t0 to 1 in case $i < n$, and to 0 otherwise. Assembly instruction 5.2 subsequently checks the value of $ t0 by the MIPS instruction beq ("branch if equal") which leads to instruction labelled (by the compiler) L1 in case the value contained in $t0 is equal to the value contained in $ zero, i.e. if it equals 0.

The two assembly language instructions above provide in their turn functional "what" specifications for machine-code instructions, defined by binary digits that are storable into memory. In MIPS, the number of bits per instructions and data is 32. Assembly instruction 5.1 is translated into the following six-field machine-code instruction

5.2.1	000000	10001	10000	01000	00000	101010
	op	*rs*	*rt*	*rd*	*shamt*	*funct*

The operating code (*op*) field, jointly with the last field *funct*, expresses that this is an arithmetic instruction, specifically a *set on less than* instruction. Fields *op* and *funct* instruct the main combinatory element of the processor, the ALU, to evaluate an inequality. Fields *rs* and *rt* refer to the source register and the target register containing the operands with respect to which the ALU has to evaluate inequality (10001 is the binary code for $\$s1$ and 10000 for $\$s0$). The five bits of field *rd* point to the destination register $\$t0$, that is, the temporary register that must be set to 1 if the inequality holds and to 0 otherwise. Fields *rs, rt*, and *rd* refer to the *Register File* in the processor (a state element collecting 32 registers which can be either read or written).[3]

Before getting to lower levels of the what-how hierarchy, let us suppose that the computing system C running P manifested the following incorrect BECS with respect to user specifications: C outputted some value k, with $k \neq m!$, when m was given as input. As was emphasised in Sect. 2, the what-how hierarchy outlined so far supplies one with a set of functional specifications (Turner 2014b) against which one may look for an explanation of the observed incorrect BECS. Downward movements along this hierarchy correspond to the breaking down of higher-level specifications into sets of

[3] The shift amount field, named *shamt* and usually involved in data transfer and conditional-branch instructions, refers to the shift left side or right side of a given bit in an instruction. Here, it is set to 0, since it is of no significance for the *slt* instruction.

progressively more elementary specifications. Thus, in particular, the recursive definition of $n!$, and the capacities that are required to compute $n!$ in accordance with that definition, are analysed into specifications provided by means of the Pascal program instructions and the capacities that the corresponding virtual machine must be endowed with in order to carry them out. These specifications are in their turn broken down in terms of detailed assembly language instructions until one reaches, further down along the what-how hierarchy, the logic circuit and gate layer.

Candidate explanations for the observed miscomputation can be identified by selecting, in a what-how hierarchy, the higher-level functional specification that fails to provide a correct how decomposition for a what prescription that one finds at the next upper level in the hierarchy. To exemplify, one might summon an incorrect recursive definition of the factorial function at the top level of the hierarchy—where, say, the first recursion equation reads $0! = 2$ instead of $0! = 1$—as an erroneously conceived specification of the factorial function. [4] This is a perfectly adequate explanation of the observed BECS *provided that the incorrect specification expressed as a recursive definition is inherited throughout the lower hierarchical levels*. If this ceteris paribus assumption holds, no additional explanatory force is achieved by mentioning any lower-level functional or structural properties. Descriptions of functional roles or their causal role fillers one may find at lower levels are selectively concerned with causal factors that do not "make a difference" (Strevens 2008) in explaining the observed miscomputation under the above ceteris paribus assumption. Those causal factors do make a difference in explaining why the observed computation occurred, but they fail to make any difference in explaining why such computation is incorrect. The reason is that the factorial recursive definition is itself a specification for lower levels: lower levels simply add information about how the incorrect specification is instantiated without adding difference-making what-details about its incorrectness. This, we argue, is a crucial feature of BECS explanations which is rooted in the richly layered structure of what-how hierarchies of functional specifications for engineered computing systems.

Supposing that the recursive equations correctly define the factorial function, an explanation of the observed incorrect BECS might be alternatively found by moving downward along the what-how hierarchy and looking for Pascal code errors. Thus, a program debugger might point to an error in, say, code line 5—which reads *WHILE* $i \leqslant n$ *DO* instead of *WHILE* $i < n$ *DO*[5]—to explain an observed incorrect BECS. Clearly, explanations appealing to some program code error only—without mentioning lower-level functional roles and their causal role fillers—presuppose that the identified program code error is inherited throughout lower hierarchical levels.

If there are no errors in the Pascal code, one may look for an explanation of the observed incorrect BECS by proceeding further down along the what-how hierarchy. There, the functional specification expressed by means of pseudo-Pascal instruction 5 can be further analysed in terms of, say, MIPS assembly language instructions 5.1 and 5.2 and their corresponding machine code instructions.

[4] In the taxonomy provided by Fresco and Primiero (2013), miscomputations engendered by wrongly conceived specifications are called *mistakes*.

[5] Syntax encoding errors count as *slips* in Fresco and Primiero's (2013) terminology.

Let us now briefly consider some implications of the above remarks about BECS explanations, as these can be plausibly construed in the framework of causal-mechanistic models of explanation (Piccinini 2007; Piccinini and Craver 2011; Piccinini 2015). For our present purposes, a mechanism is identified with a set of "entities and activities organized such that they are productive of regular changes from start or set-up to finish or termination condition" (Machamer et al. 2000, p. 3), and a mechanistic explanation of some empirical phenomenon with some description of the mechanism that brings about that phenomenon. A full-fledged description of an actual mechanism is usually distinguished from a *mechanism schema*, that is, "a truncated abstract description of a mechanism that can be filled with descriptions of known component parts and activities" (Machamer et al. 2000, p. 15); and the latter is in turn distinguished from a *mechanism sketch*, "an abstraction for which bottom out entities and activities cannot (yet) be supplied or which contains gaps in its stages" (p. 18).

Clearly, the candidate BECS explanations outlined above are not based on a full-fledged description of an actual mechanism. More specifically, only functional roles for causal fillers in the computational mechanism are used to explain some incorrect BECS. This fact does not unveil a weakness of the candidate BECS explanations insofar as no explanatory benefit is accrued by supplementing the description of an actual functional mismatch between, say, user requirement R and program instruction I with a description that one finds at lower layers of the what-how hierarchy of physical processes by means of which the incorrect specification I is actually carried out. If the incorrect specification is inherited downward all the way through the what-how hierarchy, these additional descriptions merely introduce non-difference-making information into a more parsimonious *explanans*. Indeed, it is not always the case that if there is a mismatch between R and I then some constraint is violated at lower levels too. Causal role fillers in lower-level mechanistic details do not play a significant evidential role in this explanatory context—as long as no information is available that some constraint has been violated at lower levels of the what-how hierarchy. Thus, according to any reasonable construal of the notion of explanatory force, no additional explanatory force is ceteris paribus accrued by supplementing the functional mismatch description with additional information that one finds downward along some suitable what-how hierarchy. Moreover, from a cognitive perspective, the inclusion of non-difference-making causal details may jeopardise the intelligibility of *explanantia*. Therefore, abstraction from causally irrelevant functional or structural details is an explanatory virtue in computer science.

A significant class of explanations in computer science are solely based on abstract mechanism schemata, and the process of filling in these "truncated abstract descriptions" of mechanisms "with descriptions of known component parts and activities" (Machamer et al. 2000, p. 15) is not pursued for good reasons of explanatory adequacy and intelligibility. Machamer et al. (2000) underline how "bottoming out is relative" (p. 13), that is, there is no a priori bottom mechanistic level in natural phenomena. Bottom levels are identified with mechanisms involved in the *explanandum* phenomenon (where lower mechanisms are not). In the case of the candidate *explanantia* of the observed miscomputation analysed in this section, one *already knows* that there are bottoming-out mechanisms insofar as those *explanantia* are provided by specifications imposing what-constraints on the lower specification layer in what-how hierarchies.

The conclusion that abstraction from descriptions that one finds at lower layers of the what-how hierarchy does not invariably come with a diminished explanatory force of BECS explanations converges with similar conclusions about explanations in psychology (Barrett 2014) and in biological modelling (Levy and Bechtel 2013), about the explanatory role of functional constraints in mechanistic explanations (Piccinini and Craver 2011) and about the role of abstraction in adequate mechanistic explanations (Boone and Piccinini 2016). In particular, Boone and Piccinini (2016) distinguish between epistemic roles and ontic roles of abstraction in mechanistic explanations. Mechanism schemata are adequate explanations when epistemically motivated by the choice of a specific level of organization of the involved mechanism. And epistemically motivated abstractions are often used to explain misfunctions of mechanisms by identifying some faulty properties of the mechanism and omitting causal details concerning proper functioning. Specifications in the what-how hierarchy that fail to correctly instantiate requirements that one finds higher up in the hierarchy are schemata of computing mechanisms providing adequate explanations of occurred miscomputations. Indeed, one is identifying the appropriate mechanism organization level wherein some required property of the underlying mechanism is not satisfied, while avoiding reference to lower functional and causal details concerning satisfied properties of the mechanism.

4 Role Filling and the Useful Idealization of Digital Behaviour

Higher-level descriptions cannot be always insulated in BECS explanations from lower-level descriptions that one finds in what-how hierarchies. Clearly, if hardware malfunctions are the source of an incorrect program run, then an adequate explanation of the observed BECS must summon descriptions of the malfunctioning physical components. In that case, one must proceed downward along the what-how hierarchy, reaching the descriptions that one finds at the logic circuit and gate layer and further down to the descriptions of physical role fillers for logic circuit and gate functions. Before considering an instance of hardware malfunction and its candidate explanations, let us first unfold some of the functional specifications provided above for the factorial function, all the way down to the logic circuit and gate layer.

Code instruction 5.2.1 examined in Sect. 3 can be functionally analysed in the framework of a MIPS architecture (Fig. 1) in terms of *state* and *combinatory* elements corresponding to the six fields appearing in that instruction (Patterson and Hennessy 2013, p. 248). For example, state (or memory) elements represented in Fig. 1 include the Register File, the data/instruction Memory, the Program Counter (PC) containing the address of the instruction to be executed next, and additional registers saving data from input registers that have to be processed in the immediately following clock cycle.[6] These additional registers include the Instruction Register, the Memory Data Register, registers A and B inputted from the Register File, and the ALU Out register receiving as input values computed by the ALU (see Fig. 1). The ALU in Fig. 1 is a combinatory element.

[6] In MIPS implementations, at each clock cycle of the processor, new data are written in all registers (whereas data can be read anytime).

Instruction 5.3.1 takes four clock cycles to be executed by the processor in Fig. 1. At the first clock cycle, the instruction is picked up from memory using the memory address contained in PC; the instruction is saved in the Instruction Register so that it will not be lost before the next clock cycle; and PC is incremented so as to point to the address of the next instruction. At the second clock cycle, fields *rs* and *rt* in the machine code instruction (*$s*1 and *$s*0 in the Register File) are read; the values contained there are saved in registers A and B to be used in the next clock cycle. At the third clock cycle, fields *op* and *funct* are read to let the ALU compute the corresponding mathematical operation; in the present case, the ALU verifies the inequality between the values inputted from registers A and B, outputting 1 in case the first value is less than the second value and 0 otherwise. The value is saved in the additional register ALU Out. During the last clock cycle, the destination register in *rd*, in this case *$t*0, is used to save the value in ALU Out.

This is plainly a description of a computing mechanism (Piccinini and Craver 2011): state elements and combinatory elements involved in the description, and described in Fig. 1, are defined in terms of *functional units* (Patterson and Hennessy 2013, p. 245), that is, black boxes which satisfy certain input-output relations. The Registers box at the centre of Fig. 1 is given there as a black box characterised by two input lines specifying the number of the two operand registers to be read, two output lines for the two read values which have to be stored in the additional registers A and B, and two input lines for the data to be stored in the destination register—the first one specifying the register number and the second one sending the data to be saved. The MIPS register file is functionally analysed in terms of the organization of the 32 registers composing it. And each register is in turn functionally analysed in terms of flip-flop functional organization. Consider now, as a simple kind of flip-flop, the S-R latch of Fig. 2.[7]

This basic memory element is functionally characterised in terms of two input lines, S (set) and R (reset), and two output lines, Q and its complement $\overline{Q}$. It is obtained by suitably composing two NOR logic gates. In *S-R* latches, when S is affirmed, Q is affirmed and $\overline{Q}$ is not affirmed. When S stops being affirmed, Q keeps being affirmed. When R is affirmed, $\overline{Q}$ is affirmed, Q is not affirmed, and $\overline{Q}$ keeps on holding when R stops being affirmed.

Figure 2 describes functional roles, specified in terms of input-output (I/O) relations, which any physical device must satisfy in order to count as a physical role filler for an S-R latch. In current computing systems, one generally uses sets of properly integrated transistors as physical role fillers for S-R latches and the NOR logic gates that are involved. The identification of a single transistor in this set which fails to make the state transitions that are associated with some I/O relationship in the functional specification of an S-R latch may suffice to explain a variety of incorrect BECS that are observed.[8] In this case too, one is identifying the higher layer in the what-how hierarchy which fails to satisfy some functional requirements that have been inherited downward throughout the hierarchy. One should be careful to note that this layer—which is the higher one from the viewpoint of functional requirement violation—happens to coincide with the bottom layer of the what-how hierarchy. This is a relevant circumstance for

[7] S-R latches do not take into consideration clock signals, so they are not, technically speaking, flip-flops. However, for the sake of simplicity, only S-R latches are considered here.

[8] This is an *operational malfunction* in Fresco and Primiero's (2013) taxonomy.

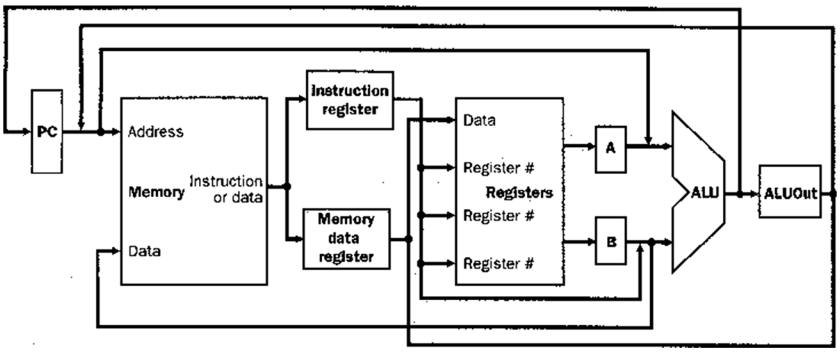

Fig. 1 A MIPS architecture

explanations that one builds by reference to the descriptions that one finds in this layer. Indeed, unlike the explanations discussed in the previous section, the explanation advanced for violations of the functional must-prescription for the S-R latch does make direct reference to physical role fillers that are expected to fulfil that must-prescription.

The explanation advanced for violations of the S-R latch functional must-prescription turns out to be a parsimonious explanation which can hardly achieve greater explanatory force by including additional causal details about the behaviour of (incorrectly or properly working) transistors implementing the S-R latches. Let us explain.

The class of potential role fillers is circumscribed in the above functional specification of an S-R latch simply by requiring that any physical role filler must possess distinguished states to be conventionally associated to one value from the set {0, 1} and that suitable Boolean functional conditions must be satisfied for state transitions in the same set. Accordingly, these functional specifications of an S-R latch involve the use of a finite collection of discrete variables only. However, the orbits and state space trajectories of transistors and other devices (e. g. vacuum tubes) that are used as role fillers for functional specifications of S-R latches and logic gates are described in classical physics by means of continuous—rather than discrete—macroscopic variables.[9] As a consequence, a plurality of non-isomorphic logical models satisfies the above functional specifications for S-R latches and logic gates. The domain of some of these models is a state space formed by a non-countable set of states; the domain of some other models satisfying the same specifications is a state space formed instead by a finite set of states. In models of the former type, any switching transition between states representing the S-R latch values 0 and 1 is described by means of a continuous trajectory in the state space. In models of the latter kind, the same switch is described as an instantaneous change involving no intermediate state whatsoever. Accordingly, in order to explain hardware malfunctions by reference to the logic circuit and gate layer, one can often dispense, and profitably so, with models involving uncountable domains and continuous trajectories in the state space, relying instead on models allowing for instantaneous transitions between states representing the values 0 and 1, respectively. Indeed, by mentioning idealised digital behaviours and by simplifying accordingly the

[9] See (Trautteur and Tamburrini 2007, and especially pp. 107–108) for a discussion of the discrete-continuous polarity in the context of computing systems and their functional role fillers.

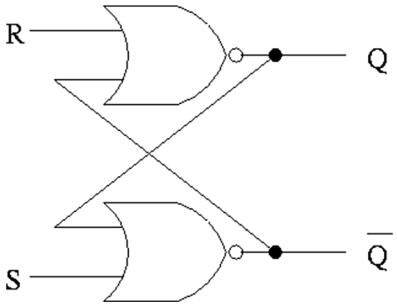

Fig. 2 An S-R latch

causal story that one may provide about the behaviour of transistors in terms of classical physics, one focuses on the violation of the relevant functional specifications at the logic circuit and gate layer by some S-R latch physical role filler. If the explanatory goal is to identify in the faulty hardware component the causes of the functional prescription violation, no additional explanatory force is accrued by providing a detailed description, in the framework of (classical) physical theories, of how that faulty hardware component brings about the incorrect behaviour in question. Those causal details turn out to be irrelevant in that they do not make a difference in the violation of the S-R latch specification. The core factors (Weisberg 2007) causing the violation of the involved specifications are to be identified here with failures to make transitions between states that are abstractly described as states 0 and 1 in accordance with the S-R latch specification. Any other states in the continuous trajectory are causal factors contributing to bring about the transistor's behaviour, but they do not make any difference with respect to the classification of the observed behaviour as incorrect.

One should be careful to note that, by providing explanations of incorrect BECS by reference to idealised digital behaviours of hardware components satisfying the laws of classical physics, one is making use of idealizations in explanation (Weisberg 2007). The use of idealizations for explaining both incorrect *and* correct BECS is a widespread practice in computer science, which comprises as a particular case the idealizations involved in the conceptual shift from a continuous to a discrete dynamics in connection with descriptions of the behaviour of physical components of computing systems. Prominent examples of a different sort, which we now turn to examine, are found in explanations of BECS relying on specification and verification methods developed in theoretical computer science.

5 Combining Abstraction and Idealization in BECS Explanations

Many computing systems of interest in computer science are appropriately qualified as *reactive* systems, that is, as systems interacting with their environment. A relatively simple case in point are controllers of beverage vending machines, which interact with an unlimited sequence of users, recognise their requests, and distribute beverages accordingly as long as they are properly maintained and supplied with beverages as needed. Additional examples of simple reactive systems are

controllers of household microwave ovens and controllers of traffic light junctions interacting with pedestrian requests.

Formal verification methods in theoretical computer science enable one to provide explanations for both correct and incorrect BECS manifested by reactive systems on the basis of suitably specified models of those systems. An examination of these models enables one to advance and support the claim that many explanations of reactive systems' behaviours involve an extensive use of both abstraction and idealizations. On the one hand, abstraction *hides* the details of some lower-level descriptions (Colburn and Shute 2007). On the other hand, idealization makes the inclusion of some lower-level descriptions impossible. Both abstraction and idealization are introduced with the aim of offsetting the impact of complexity limitations on the applicability of formal verification to predict and explain the behaviour of reactive systems. Here, we consider interactions between abstraction and idealization in the context of *model checking* formal verification methods (Baier and Katoen 2008; Clarke et al. 1999) and their implications for BECS explanations.

Model checking enables one to verify whether the unending runs of some reactive system R satisfy, *according to a suitable model* M of R, properties that are decidable (and efficiently so) in M. Decidable properties of reactive systems that are of interest for system users, programmers, and various other stakeholders, include *safety, invariant, and liveliness* properties. Safety statements assert that states possessing some undesired feature are unreachable in any run of R, *invariant* statements assert that all reachable states satisfy some desired feature, and *liveness* statements assert that states satisfying some desired feature are eventually always reachable from any state of R. Model checking allows one to explain why a reactive computing system reached (or did not reach) a desired/undesired state by showing a computational path in the model starting from some initial state and going through (or not going through) the desired/undesired state. In case of explanations of miscomputations, those paths are often used to trace back error states in the running program code (see, for instance, Callahan et al. 1996).

Let us concretely consider the case of a microwave oven controller and suppose one would like to check the liveness property that whenever the oven is on, it will eventually start heating. For this verification purpose, one may develop a model abstracting from many features of the actual computing system which controls the oven. A model consisting of eight states only will do—with each state represented by a circle, arrows between states representing transitions between them, and initial states represented as circles pointed to by an arrow incoming from an unspecified source. Each one of the represented states is an abstract macrostate obtained by collapsing, into one state, many actual states of the microwave oven controller. Abstract macrostates are obtained by a *data abstraction function* mapping variables appearing in more fine-grained representations of the controller into some macrovariables. The functional properties of macrostates that one needs in order to check the desired liveness property can be represented as atomic propositions *start*, ¬ *start*, *close* , ¬*close*, *heat*, ¬*heat*, *error,* ¬*error*.[10] The resulting model can be represented as the state transition diagram of Fig. 3.

The state transition diagram of Fig. 3 can be viewed as a Kripke structure (KS) $M = (S, S_0, R, L)$ defined by states in S with a subset of initial states S_0, the transition

[10] The example is taken from Clarke *et al.* (1999, pp. 38–39).

relation $R = S \times S$, and a labelling function $L: S \rightarrow 2^{AP}$ labelling each state with subset of a fixed set AP of atomic propositions that are true in that state (Clarke et al. 1999, pp. 13–26). Abstract KSs of the kind depicted in Fig. 3 enable one to model the multiple processes that are concurrently executed by a machine while carrying out a given task. Causal interactions among those processes can be represented in an abstract model by means of compositions of different state transition systems—where each one of these systems models a single process (Baier and Katoen 2008, p. 35).[11]

In order to check whether the KS model satisfies the property one is interested in, the latter is usually formalised by means of a temporal logic formula. The liveness property that whenever the oven is on, it will eventually start heating can be formalised in computation tree logic (*CTL)* by means of the formula $\boldsymbol{AG}$ (*Start* → $\boldsymbol{AF}$ *Heat*). This formula states that in all paths ($\boldsymbol{A}$) starting from any initial state, and in every state (globally: $\boldsymbol{G}$) of those paths, if *Start* holds then in all paths ($\boldsymbol{A}$) starting from there *Heat* will finally ($\boldsymbol{F}$) hold. The model checking technique makes use of a depth-first search algorithm to explore the state space of a KS (or model) M and to check whether $M \vDash f$, that is, whether the temporal ordering constraints on program behaviours that are expressed by a formula f are satisfied by model M. In the microwave example, the model checking algorithm enables one to check whether $M \vDash \boldsymbol{AG}$ (*Start* → $\boldsymbol{AF}$ *Heat*) or not. In case of a positive answer, the algorithm outputs a set of "witnesses", that is, of paths in the KS that fulfil the checked properties. And in case of a negative answer, "counterexamples"[12] are advanced by the algorithm, which consist of paths in the model violating the temporal logic formula.

In model checking, an abstract description of the functional organization of the system is provided in terms of macrostates and transitions between them (e.g. the microwave oven KS). Many elements that would occur in a complete description of the system are irrelevant for prediction and explanation purposes (its microstates). In particular, interactions between these components in a fine-grained system's description (microstates and their transition conditions) do not make a difference in explaining a wide variety of the system's temporal ordering properties holding among the specified macrostates. The adopted explanation strategy is based on an abstract model (that is, the KS viewed as an abstract model) which shares many distinctive aspects with scientific models that are used in the empirical sciences for predictive and explanatory purposes (Angius and Tamburrini 2011). If a system is functionally organised as specified in the KS, then it must possess the reachability, safety, and liveness properties that hold of the KS, and whose instances one can observe in the system's behaviour. Thus, one explains both complex behavioural regularities (reachability, safety, liveliness, etc.) and their instances by reference to the state transition trajectories that are permitted or forbidden by the abstract description of the computing *mechanism* (Piccinini 2015).

[11] Kripke Structures have been more explicitly used to represent causal structures (see for instance Alur et al. 1998, p. 45). More in general, checking KSs against specified temporal formulas can be useful to infer causal relations from temporal data (Kleinberg 2012).

[12] Paths in the model violating the temporal logic formula are identifiable with counterexamples under the assumption that model M provides a correct representation of the reactive system for the specific verification purpose at hand. This crucial assumption can be empirically controlled by starting the reactive system under suitable initial conditions and verifying whether the runs that are actually observed are correctly modelled by those paths in the model which violate the temporal logic formula according to the model checking algorithm.

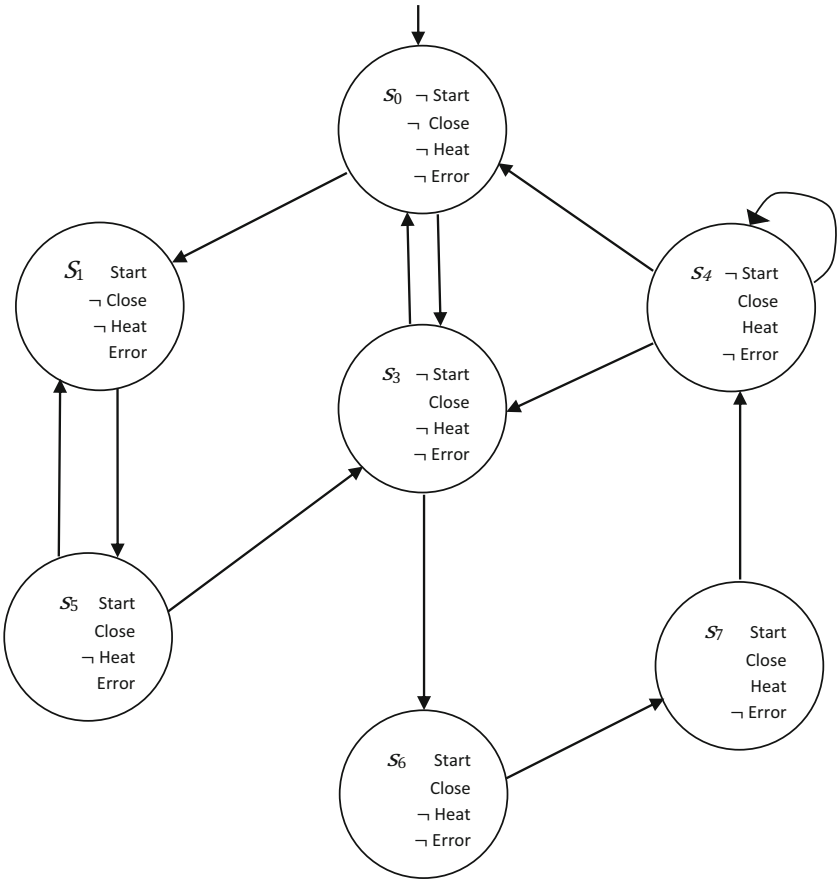

Fig. 3 An abstract model for verifying a liveness property of a microwave oven

In addition to abstraction, idealizations (Nowak 1979; Cartwright 1989; Weisberg 2007) are extensively introduced to build models in model checking. Abstraction and idealization, which are introduced to simplify empirical models for representational or explanatory purposes, usually interact with each other in the construction of simplified models. In the following, interactions between abstraction and idealization are examined in the context of model checking by reference to the microwave oven controller example. In particular, a distinction between *restrictive* and *permissive* idealizations is introduced and their respective impact on BECS explanations is unravelled.

Restrictive idealizations come into play when one assumes that physically possible trajectories, either reflecting "unreasonable" interactions with the environment or else arising on account of hardware failures, will never occur. For example, path $\pi' = s_0, s_3, s_0, s_3, \ldots, s_0, s_3$, which is allowed in the abstract model (KS) of Fig. 3, corresponds to the odd behaviour engendered by users repeatedly opening and closing the oven door (whereby *start* is true but *heat* never holds). And a malfunctioning oven hardware may give rise to a path π , s_0, s_1, s_5, s_3, s_0, wherein the oven never heats up and the states in the sequence are all error states. If these physically possible trajectories are taken into account, then the CTL formula $\boldsymbol{AG}(\mathit{Start} \rightarrow \boldsymbol{AF}\,\mathit{Heat})$ is not satisfied in the abstract

model M and the two paths shown above will be advanced as counterexamples together with the negative answer to this satisfaction problem. In order to check solely whether the controller's *software* fulfils given functional requirements, one should discard executions in which the violation of the property under examination does not depend on failures of the controller's software. This is usually achieved by imposing suitable *fairness constraints*. A fairness constraint is defined in terms of the set of states that are required to appear infinitely often in any travelled path of the KS and is expressed in terms of CTL formulas on a par with property specifications.[13] Models satisfying fairness constraints are idealised models in that fairness constraints can be identified with ceteris paribus *clauses* imposed on the model behaviour, whereby one assumes the correct functioning of hardware components (Angius 2013). Fair KSs may still be used to develop full-fledged mechanistic descriptions of computing systems carrying out computations satisfying the functional constraints expressed by the KS, insofar as one is only *making the false assumption that* some physically possible trajectories which are not relevant to explain software functional properties will never occur. This can be achieved in the microwave oven example (a) by dropping the fairness constraints concerning hardware failures or unreasonable interactions with the environment and (b) by adding lower-level descriptions of the microwave oven physical components and their physical processes.

Permissive idealizations come into play when one includes in the model trajectories that fail to represent actual execution paths of the system that one is modelling. Some of these unrealistic trajectories, however, are quite relevant to the BECS *explanandum*, unlike those excluded by means of restrictive idealizations. In particular, consider a KS allowing for spurious paths that do not match actual program executions. Permissive idealizations of this kind can be introduced by the abstracting function from the program variables to the set of abstract functional variables, which usually increases the *granularity* of the KS, so that each state in S is a macrostate corresponding to many actual program states. A very simple case is illustrated in Fig. 4 wherein abstract path π°=1°,2°,3°,4° is a spurious path in that it does not represent any software execution represented by transitions between states in the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$. Each abstract state is obtained by the collapse of three different program states, and among those fine-grained states, one may rather find executions from state 1 to state 9 *and* from state 7 to state 12. The collapse of both state 7 and state 9, together with state 8, into the abstract state 3° is responsible for the spurious abstract path.

After checking that $M \vDash \boldsymbol{AG}(Start \rightarrow \boldsymbol{AF}\, Heat)$, spurious paths of this sort may be outputted, together with a positive answer, as elements of the set of witnesses. Some argue that idealised models in science should be de-idealised, in order to restore the empirical adequacy of the model, after using the distorted model for some representational or explanatory purposes (McMullin 1985; Weisberg 2013). However, a spurious path does not jeopardise *locally* the empirical adequacy of the KS, that is, its empirical

[13] A path is fair in case it satisfies each CTL fairness formula infinitely often, and a fair KS $N = (S, S_0, R, L, F)$ is also defined by a set $F \subseteq 2^S$ of fairness constraints. A fairness constraint avoiding that unfair path be travelled by the model checking algorithm exploring the KS of Fig. 3 may be given by the formula $Start \wedge Close \wedge \neg Error$ which, when satisfied by some fair path, requires that the system will eventually enter a non-error state. Considering a fair KS M° for the microwave oven, $M^{\circ} \vDash_F \boldsymbol{AG}(Start \rightarrow \boldsymbol{AF}\, Heat)$, if there exists a fair path starting from an initial state, that is, witness $\pi''' = s_0, s_1, s_5, s_3, s_6, s_7$.

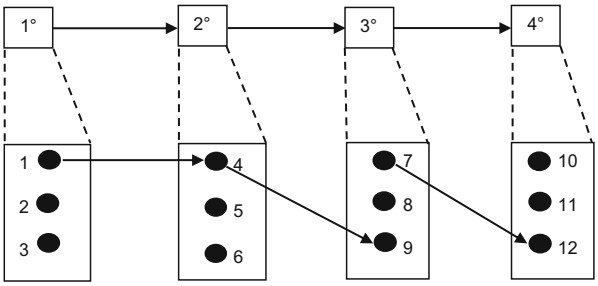

Fig. 4 A spurious abstract path

adequacy with respect to the temporal formula to be checked for satisfiability in the KS. The spurious path needs to be removed if it corresponds to a false counterexample (or false negative), insofar as counterexamples are used to remove errors from the checked faulty program. In such cases, the granularity of the KS is decreased until one isolates the faulty modelled transitions, this process being known as *abstraction refinement* (Wang et al. 2006). If the spurious path does not correspond to a false counterexample, there is no need to remove it from the KS, for the satisfiability of the temporal formula under examination is independent of the presence of the spurious path. In particular, if the formula $\boldsymbol{AG}(\mathit{Start} \rightarrow \boldsymbol{AF}\,\mathit{Heat})$ is positively model-checked, the observed behaviours of the microwave oven fulfilling the liveliness property under consideration are correctly predicted and explained by reference to a model (the KS) allowing for system runs that the actual microwave oven cannot perform.

Spurious paths are introduced as a by-product of principled approaches to reduce the amount of computational resources one has to allocate in order to carry out the model checking procedure. If one is interested in the temporal ordering of two system's properties (such as those corresponding to labels *Start* and *Heat*), and transitions are given between the program's states where one property holds and states where the other property holds, then one can idealise the system's representation as if there were one single transition between two macrostates. Spurious paths need not be removed if the idealised representation allows one to successfully check the other temporal formulas of interest. If false counterexamples are traced by the model checking algorithm, spurious paths are removed to check whether the property is still violated by the revised model or not.

In conclusion, KSs and other state transition systems that are used in model checking to explain correct BECS are models that are built by making extensive use of abstractions and idealizations. Both abstractions and idealizations are often needed to decrease the state space of the resulting model and to make an exhaustive search in the set of system trajectories computationally feasible. Abstractions from program variables to macrostates in the model hide physical descriptions of entities and activities of the full-fledged instantiating computational mechanisms. It was argued in the previous sections that by adding to an abstract model information about physical role fillers in computational mechanisms, one does not necessarily increase the explanatory force of explanations that are based on the abstract model. In this section, we have illustrated the practice of introducing idealizations which distort computational models by assuming that some "disturbing" processes that are involved in the actual computational mechanism do not take place (restrictive idealizations) or by introducing processes that are

not to be found in the actual computational mechanism (permissive idealizations). Accordingly, BECS explanations relying on models obtained by permissive idealizations are locally tailored to the (liveliness) property *P* under examination. And the underlying model cannot be developed in order to serve a variety of additional explanatory purposes, insofar as the model cannot accommodate some downward expansions of what-how hierarchies—so as to include functional specifications of microstates and their transitions, in addition to full-fledged descriptions of their role fillers that are couched in the language of physical theory.

6 Conclusions

Explaining BECS is a pervasive and significant activity in computer science practice. An examination of this activity was carried out above with the aim of addressing the methodological issue of what it is to explain observed incorrect BECS and to explain regular behaviours that some classes of engineered computing systems are capable of manifesting. Both notions of correct and incorrect BECS have been clarified on the basis of sets of specifications prescribing desired behavioural properties of interest. Users, programmers, engineers, and other stakeholders furnish specifications that are organised into hierarchies of what-how descriptions—wherein each description affords a functional specification for lower-level descriptions, and the descriptions at the bottom layer of the hierarchy include structural details about the physical devices implementing logic circuits and gates.

Explanations of both correct and incorrect BECS can be given using as *explanantia* specifications that one finds at various levels of the what-how hierarchy, without necessarily including descriptions of the physical role fillers that one finds at the bottom level and that are couched in the language of physical theory. Accordingly, explanations abstracting away from descriptions provided at the bottom level of what-how hierarchies provide adequate answers for a wide variety of explanatory requests arising in computer science practice. In the language of mechanistic approaches to explanation, this means that in various circumstances, explanations that are based on abstract mechanism schemata, rather than full-fledged mechanism descriptions, are to be preferred on the multiple grounds of explanation correctness, relevance, and intelligibility.

The present analysis of combined abstraction and idealization in BECS explanations may prove useful to evaluate, from a methodological perspective, explanatory strategies that one adopts in areas of scientific inquiry that are collected under the name of "software intensive sciences" (Symons and Horner 2014), insofar as one makes extensive use of computational methods there. Notably, model checking and other formal verification methods are being profitably used to perform in silico experiments in systems biology and to explore exhaustively the trajectories in the state space of simulated biological cell systems (Fisher and Henzinger 2007; Angius 2015). There, computational models provide abstract and idealised descriptions of cell systems, which are used to predict and explain a variety of cell behaviours. However, an extensive use of permissive idealization in executable cell biology may give rise to multiple and possibly incompatible models (Weisberg 2007) of cell systems, whose respective representational, predictive, and explanatory roles must be properly understood in the broader scientific context of systems biology.

Acknowledgments We are grateful to the anonymous reviewers who helped us to focus on the core theses of the paper and suggested valuable improvements.

References

Alur, R., McMillan, K., & Peled, D. (1998). Deciding global partial-order properties. In *International colloquium on automata, languages, and programming* (pp. 41–52). Springer Berlin Heidelberg.

Angius, N. (2013). Abstraction and idealization in the formal verification of software systems. *Minds and Machines, 23*(2), 211–226.

Angius, N. (2015). Computer simulations without simulative programs in executable cell biology. Hypothesis discovery and justification. *Paradigmi, 32*(3), 67–82.

Angius, N., & Tamburrini, G. (2011). Scientific theories of computational systems in model checking. *Minds and Machines, 21*(2), 323–336.

Baier, C., & Katoen, J. P. (2008). *Principles of model checking* (Vol. 26202649). Cambridge: MIT press.

Barrett, D. (2014). Functional analysis and mechanistic explanation. *Synthese, 191*(12), 2695–2714.

Bechtel, W., & Abrahamsen, A. (2005). Explanation: a mechanist alternative. *Studies in History and Philosophy of Science Part C: Studies in History and Philosophy of Biological and Biomedical Sciences, 36*(2), 421–441.

Boone, W., & Piccinini, G. (2016). Mechanistic abstraction. Forthcoming in *Philosophy of Science*, doi:10.1086/687855.

Callahan, J., Schneider, F., & Easterbrook, F. (1996). Automated software testing using model checking. In J. C. Gregoire, G. J. Holzmann and D. Peled (Eds), *Proceeding spin workshop*, pp. 118–127. Rutgers.

Cartwright, N. (1989). *Nature's capacities and their measurement*. Oxford, New York: Oxford University Press.

Clarke, E. M., Grumberg, O., & Peled, D. (1999). *Model checking*. Cambridge: MIT press.

Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds and Machines, 17*(2), 169–184.

Fisher, J., & Henzinger, T. A. (2007). Executable cell biology. *Nature Biotechnology, 25*(11), 1239–1249.

Floridi, L., Fresco, N., & Primiero, G. (2014). On malfunctioning software. *Synthese, 192*(4), 1199–1220.

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

Glennan, S. S. (1996). Mechanisms and the nature of causation. *Erkenntnis, 44*(1), 49–71.

Jones, M. R. (2005). Idealization and abstraction: a framework. In M. R. Jones & N. Cartwright (Eds.), *Idealization XII: correcting the model. Idealization and abstraction in the sciences* (pp. 173–217). Amsterdam: Rodopi.

Kleinberg, S. (2012). *Causality, probability, and time*. Cambridge University Press.

Levy, A., & Bechtel, W. (2013). Abstraction and the organization of mechanisms. *Philosophy of Science, 80*(2), 241–261.

Machamer, P., Darden, L., & Craver, C. F. (2000). Thinking about mechanisms. *Philosophy of Science, 67*(1), 1–25.

McMullin, E. (1985). Galilean idealization. *Studies in History and Philosophy of Science Part A, 16*(3), 247–273.

Nowak, L. (1979). *The structure of idealization. Towards a systematic interpretation of Marxian idea of science*. Dordrecht: Kluwer.

Patterson, D. A., & Hennessy, J. L. (2013). *Computer organization and design: the hardware/software interface*. Waltham, MA: Morgan Kaufmann.

Piccinini, G. (2007). Computing mechanisms. *Philosophy of Science, 74*(4), 501–526.

Piccinini, G. (2015). *Physical computation: a mechanistic account*. Oxford: Oxoford University Press.

Piccinini, G., & Craver, C. (2011). Integrating psychology and neuroscience: functional analyses as mechanism sketches. *Synthese, 183*(3), 283–311.

Pitt, J. C. (2011). *Doing philosophy of technology: essays in a pragmatist spirit* (Vol. 3). Dordrecht: Springer.

Primiero, G., & Raimondi, F. (2015). Software theory change for resilient near-complete specifications. *Procedia Computer Science, 52*, 988–995.

Strevens, M. (2008). *Depth: an account of scientific explanation*. Harvard University Press.

Symons, J., & Horner, J. (2014). Software intensive science. *Philosophy and Technology, 27*(3), 461–477.

Tanenbaum, A. S. (2006). *Structured computer organization*. Upper Saddle River, NEW JERSEY: Pearson.

Trautteur, G., & Tamburrini, G. (2007). A note on discreteness and virtuality in analog computing. *Theoretical Computer Science, 371*(1), 106–114.

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

Turner, R. (2014a) The philosophy of computer science. Resource Document. *The Stanford Encyclopedia of Philosophy* (Winter 2014 Edition), Edward N. Zalta (ed.). http://plato.stanford.edu/entries/computer-science/.

Turner, R. (2014a). Programming languages as technical artifacts. *Philosophy and Technology, 27*(3), 377–397.

van Eck, D. (2015). Mechanistic explanation in engineering science. *European Journal for Philosophy of Science, 5*(3), 349–375.

Wang, C., Hachtel, G. D., & Somenzi, F. (2006). *Abstraction refinement for large scale model checking*. Berlin: Springer.

Weisberg, M. (2007). Three kinds of idealization. *The Journal of Philosophy, 104*(12), 639–659.

Weisberg, M. (2013). *Simulation and similarity: using models to understand the world*. New York: Oxford University Press.