APhEx 28, 2023 (pp. 322–338)
Ricevuto il: 31/07/2023
Accettato il: 11/12/2023
DOI 10.13137/2036-9972/35928

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

O P E N P R O B L E M S

Can Computer Programs be Formally Verified?

Nicola Angius

Università di Messina
nicola.angius@unime.it

This paper examines one central problem in the philosophy of computer science, namely the question of whether computer programs can be verified by means of a mathematical proof. Firstly, program verification is defined and the classical debate on its feasibility is recalled under the light of a dual ontology of computational systems. Secondly, the current resurgence of the debate is analysed underlining its logical, technological, and philosophical motivations. Finally, it is shown how adopting a stratified ontology for computational systems one may recombine the different positions in the debate, arguing how program verification involves both deductive and inductive reasoning.

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

TABLE OF CONTENTS

1. Introduction

The idea of *program verification* is as old as Turing, who, in (Turing 1949), provided a proof sketch of a routine with two nested loops (Morris and Jones 1984). In general terms, verifying a computer program means proving that it is *correct* or, in other words, that it behaves as expected. And one of the first opponent of program verification was, few years later, Wittgenstein (1958), who emphasised how machines are considered as if they "could only move in this way, as if they could not do anything else. How is this - do we forget the possibility of their bending, breaking off, melting, and so on?" (Wittgenstein 1958, p. 77)

Program verification was perceived as a problem since the beginning of the modern computing era. Not only, the two issues characterizing the program verification debate from the seventies to the present date were already clear to Turing and Wittgenstein themselves: on one hand programs are abstract entities about which one should in principle be able to attain mathematical knowledge, on the other hand their behaviour also depends on the physical machine executing them.

In contemporary terms, the program verification problem is the problem of what qualifies as a verification of a computer program, that is, how, and to what extent, it is possible to ascertain that an encoded program fulfils the functional requirements it was developed for. And the opposite views of Turing and Wittgenstein on program verification are examined as the problem of what kind computational artefacts can be verified, whether only abstract entities as algorithms or also implemented programs. This paper takes these questions to be two main open problems in the philosophy of computer science.

The two problems have an important impact in computer ethics for those computational systems operating in the so-called safety-critical contexts, wherein human safety is potentially endangered, or in human-computer

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

interactions, wherein human rights can be violated by machines. Functions are usually advanced by developers, users, and all the stakeholders involved in a development project; initially expressed in natural language, requirements are often formulated as *specifications* using some appropriate formal language (such as Z, typed predicate logic, or VDM). Program verification then consists in formally proving that a program satisfies the given specification, that is, that it behaves as requested by its functional requirements.

How this can be done is sketched in section 2 while recalling the classical debate on program verification that, starting from the 1970s, characterized the 1980's. Section 3 underlines the logical and philosophical motivations that are at the basis of the resurgence of the program verification debate in the 2000's and it analyses the terms of the current debate within the philosophy of computer science. The debate kept going up to the present date among sustainers and detractors of formal verification; section 4 examines two approaches that identify potential epistemological solutions to the program verification problem. The concluding remarks in section 5 highlight how the various different answers to the problem are related to different ontologies of computational systems.

2. The classical debate

According to Rapaport (2023), the link between proofs and programs can be initially traced back to Gödel's remark on Turing's account of computation, thanks to whom "a precise and unquestionably adequate definition of the general concept of formal system can now be given" (Gödel 1964, 71). Since rules of inference are computable functions, the proof of a theorem T from a set of axioms using rules of inference can be identified with a program P terminating with T as output and given the axioms as inputs. This analogy was later formulated in more formal terms independently by de Bruijn (1980) and Howard (1980) who, extending Curry's (1934) identification of the material implication with a computable function to all other connectives, were able to interpret all natural deduction inference rules in the λ-calculus. Such a correspondence became known as the *Curry-Howard-de Bruijn* correspondence.

Given the conformity between programs and proofs, a conformity can also be established between program verification and proof checking. In other words, verifying program correctness amounts to checking whether a proof

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

is valid, that is, whether each proposition in the proof is obtained by means of a rule of inference or is an axiom. [1]

This was the exact project of the first, as well as the staunchest, supporter of program verification: the Turing award winner Tony Hoare. Hoare (1969) understood computer science as a mathematical discipline, identifying programming activities with deductive reasoning:

> *Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning (Hoare 1969, p. 576).*

Developing upon the former work of Floyd (1967), Hoare advanced an annotation strategy allowing one to derive properties of programs from program statements. For every program statement S, a precondition P and a postcontition Q are annotated in brackets in the form $\{P\}S\{Q\}$, the latter being known as the Hoare's triple. Preconditions express the state of the computational system before the execution of instruction S in terms of the values of all variables involved in the program. Postconditions annotate the state of the system after the execution of S. Also a property specification can be formalised in terms of an Hoare's triple $\{P\}C\{Q\}$, stating that, given an initial state of the system P, the execution of the entire code C must bring the system in a state expressed by postcondition Q.[2] First, all program instructions are annotated in terms of Hoare's triples; subsequently, it is checked whether, given as precondition to the first instruction the precondition expressed by the specification, the postcondition of the last program statement corresponds to the postcondition of the specification (see also Primiero 2020, ch. 7).

One first difficulty of Hoare's approach was emphasised by De Millo, Lipton, and Perlis (1979) who stressed one fundamental difference between mathematical proofs and program verification: whereas proofs of program correctness can be obtained, they are nonetheless too much longer, complex, and thereby *unsurveyable.* Mathematical proofs are considered correct when

[1] An in-depth analysis of the *Curry-Howard-de Bruijn* correspondence, and of the conformity between programs and proof, can be found in (Primiero 2020, pp 81-88).

[2] In other words, whereas a program can be expressed in terms of a set of Hoare's triples of the form $\{P\}S\{Q\}$, each triple for each instruction in the program (where S denotes a single program instruction), a specification can be expressed in terms of a *single* Hoare's triple $\{P\}C\{Q\}$, since C denotes the entire program code.

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

accepted by the social community of mathematicians; by contrast, unsurveyable proofs can hardly be shared and accepted by the community of computer scientists. In other words, De Millo, Lipton, and Perlis argued that programs *can* be verified but verification should not count as a proper mathematical proof.[3]

But the actual controversy on program verification was kick-started by a paper published by the philosopher James Fetzer on the *Communication of the ACM* journal arguing that the issue on program verification is all based on a misconception and that programs *cannot* indeed be verified (Fetzer 1988):

> *The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility. (Fetzer 1988, p. 1048).*

The alleged misconception concerns the ontology of programs *qua* object of verification: programs may well be identified with both algorithms or their encodings and with a high-level language, or executable, instruction set. Whereas algorithms are abstract logical structures, that is, mathematical entities, programs in the latter meaning are causal models of those structures, in that they specify a set of variables, corresponding to memory locations, and how their values are changed during execution. Algorithms, by being logical structures, are amenable of deductive reasoning and can be verified using Hoare's logic; however, when one mentions program verification usually refers to programs as a high-level language instruction set. Programs so intended cannot be subject to deductive reasoning since their behaviours also depend on empirical assumptions on the behaviour of the implementing machine. Consider the basic assignment instruction $x \coloneqq n$ assigning value n to variable x; given a triple $\{P\}x \coloneqq n\{Q\}$, one cannot define with certainty the postcondition $\{Q\}$ in so far as one cannot be certain about whether the instruction $x \coloneqq n$ has been executed properly, that is, whether value n has

[3] Other problems associated by De Millo, Lipton, and Perlis (1979) with program verification concern the difficulties of formalising requirements and specifications to make them amenable of formal verification, and of separating verification from software development: unsuccessful verification often require to reformulate the involved specifications, triggering a virtuous circle in the program development method.

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

been stored in the memory location corresponding to x: this also depends on physical facts concerning the implementation.[4]

Fetzer's paper triggered a famous, and harsh, controversy on the pages of the *Communication of the ACM* between opponents and sustainers of Fetzer's position and Fetzer himself. The editor of the journal received several letters from computer scientists claiming that such an ill-informed paper should had never been published. The main critique to Fetzer was that program verification is not indeed aimed to prove that a physical implementation is correct and that physical correctness certainly implies the correctness of the implementing hardware.[5]

As underlined by Jon Barwise (1989), the controversy can be understood as an instance of the general debate in philosophy of mathematics about the relation between a mathematical model and the world. After Fetzer's paper, the debate kept going the following years seeing computer scientists divided among those arguing in favour and those arguing against the possibility of program verification.

3. The resurgence of the interest in program verification

The beginning of the new millennium saw a significant renewed interest in the program verification debate. At the basis of such a revived attention are both technical and philosophical motivations. As what concerns the former, theoretical computer science critically improved formal verification, developing new verification techniques that could successfully be deployed in industry and software development. Besides the *syntactic approaches* of Hoare (1967) and its subsequent developments (Dijkstra 1975), in the 1990's the *semantic approach* to verification became predominant, and the 2000's saw the advent of the *resource-based analysis* (Primiero 2020). One prominent semantic method is *model-checking* (Baier and Katoen 2008), wherein programs are represented in terms of state transition systems, specifications are formalised using temporal logic formulas, and an algorithm checks whether the temporal logic formula holds of the transition system.[6]

[4] Fetzer (1988) distinguished between *absolute verifiability*, occurring when a theorem can be derived only by an axioms set, and *relative verifiability*, taking place when a theorem is derivable from an axioms set together with a set of propositions concerning empirical facts. Only algorithms are absolutely verifiable; programs are rather relatively verifiable.

[5] The reader may refer to (Primiero 2020, ch. 7) for a full reconstruction of the classical debate among Hoare, De Millo, and Fetzer on program verification.

[6] Depending on the temporal logics employed to formalise the property specifications of interest, different model checking algorithms are used. In the case of CTL temporal logic,

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

Resource-based analyses apply *separation logic* (O'Hearn and Pym 1999; Reynolds 2002), a logic of bunched implications[7], to check the correct resource management by programs. Other technical improvements that turned to be significant for the debate on program verification are the development of automatic, computer-based, theorem provers for Hoare's logic (Gordon 1987, Luo and Pollack 1992) as well as the introduction of formal languages for program specifications, including Z (Spivey 1989) and VDM (Jones 1990).

The need to apply formal verification methods in the industry has arisen, in the last twenty years, as a consequence of the deployment, in many safety-critical systems, of control software in place of analogue circuits (Garoche 2019). The aim of control software is that of reading and elaborating inputs coming from sensors and to consequently control actuators. Control software is nowadays present in cars, aircrafts, railway systems, and medical devices; if control software allows a real-time and more efficient elaboration of sensor data, its complexity makes testing activities not able to guarantee a sufficient level of reliability. Formal verification, especially model checking, is being more and more used to assure the fulfilments of the software requirements. Complexity is characterizing not just software but also hardware systems, and formal methods, especially automated theorem proving, is being successfully applied to the verification of micro-processors (see for instance Harrison 2003).

But a greater awareness of the importance of the program verification problem came, starting in the late 2000s, from the debate on the *epistemological status of computer science* (Eden 2007, Tedre 2011). Within the *philosophy of computer science* (Angius *et al.* 2021), such a debate amounts to defining whether computer science is to be understood as a mathematical, a scientific, or an engineering discipline. Some of those arguing that computer programs can be verified using pure deductive reasoning sometimes consider computer science as a branch of mathematics, by explicitly referring to the argument of Hoare (1985). The 1975 Turing Award lecture by Newell and Simon (1975) contains one of the first argument

given a state transition system M and a CTL formula f, a *labelling* algorithm labels each state $s \in M$ with the set $label(s)$ of subformulas of f true in s, starting from the most nested sub-formula. The algorithm terminates when all sub-formulas are processed and if it holds that $f \in label(s)$, then M is said to satisfy f. The reader should refer to the original algorithm presented in (Clarke *et al.* 1999) for technical details.

[7] A logic of bunched implications allows to represent, and formally reason about, finite resource composition in terms of both additive and multiplicative forms of conjunction. Derivations from composed premises to conclusions thereby assume a tree-like structure.

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

in favour of the scientific nature of computer science; today, those framing computer science into an empirical paradigm emphasize the experimental nature of the discipline, which involves inductive reasoning and *software testing* (Ammann and Offutt 2016). Also holders of the engineering paradigm underline the unfeasibility of formal verification, but they stress the limits of empirical testing: software systems are *artefacts* and, as such, they must be treated as any other kind of artefact, like a bridge or an airplane, which cannot be the object of any verification attempt but rather of reliability studies and evaluations.

The development of theorem provers, that is, of software able to carry out proofs, at first sight seemed to have addressed De Millo, Lipton, and Perlis's (1979) objection against program verification: proofs of program correctness, despite being long and complex, can nonetheless be accomplished automatically using a theorem prover. This argument was quickly shown not to support the mathematical nature of program verification (Turner 2018, ch. 25). First, the usage of theorem provers simply shifts the verification problem from the program under scrutiny to the theorem prover which, by being a program, needs to be verified: in case of an incorrect theorem prover, the supplied proofs may be faulty. Secondly, the theorem prover is a program which is executed on a physical machine: the kind of proof it provides is not *a-priori* and, consequently, it should not count as a mathematical proof. This latter objection recalls the well-known Tymoczko's (1979) argument against the mathematical nature of the computer-based proof of the four-color theorem.

Arkoudas and Bringsjord (2007) address the first, main, concern by underlining that one should differentiate the search of a proof (the context of discovery) from the justification of the discovered proof (the context of justification). The theorem prover is used for discovering a proof and it thus needs not to be verified: the supplied proof can be justified through a proof checker, namely a software that checks whether a proof is correct. Proof checkers are relatively small programs and can easily be verified.

As what concerns the second difficulty, Turner (2018, ch. 25) recalls Burge's (1988) distinction between the *justification* of a knowledge and its *possibility*. A knowledge is *a-priori* when its justification does not depend on sensory experience, whereas its possibility may or may not depend on sensory experience. For instance, the sentence 'red is a colour' is *a-priori* since its justification is of analytical kind, even though the possibility of formulating such a statement relies on the sensory experience of red. In the very same way, computer-based proofs are *a-priori* insofar as only their possibility is related to an implementing machine, whereas their justification is not.

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

In the opposite direction, there are current reformulations of the argument that programs cannot be verified. Symons and Horner (2014, 2020) have it that there is no general solution to the program verification problem. By general solution they mean that there is no efficient method assuring that every behaviour of a program satisfies the specification(s) of interest. First, theorem proving using any reformulation or extension of Hoare's method is affected by undecidability limitations.[8] As what concerns model-checking, they provide a reformulation of Fetzer's argument stating the model-checking algorithm checks whether a *model* of the program, *qua* abstract entity, satisfies a given specification, and not the program itself. Accordingly, it remains to be checked if the model is an adequate representation of the program. To do this, the only way is to check whether to each program execution corresponds a path in the model. However, this cannot be done exhaustively since execution traces are potentially infinite for non trivial software.

Symons and Horner (2014, 2020) go even further with respect to the classical debate by arguing that also the inductive method of empirical testing cannot assure for program correctness. Testing consists in launching a program and evaluating whether the observed executions satisfy the given specification(s). This cannot be done in practice due to the *path complexity* of modern programs that count, on average, one binary conditional instruction every ten lines of code. For instance, testing a small software containing only 1000 code lines would require to test $\frac{2^{1000}}{10} = \sim 10^{30}$ paths. The authors argue that Conventional Statistical Inference Theory (CSIT, see Hogg *et al.* 2005) cannot be applied to estimate the error distribution of software code, since conditional statements do not allow CSIT to analyse a sufficiently representative sample.

4. The ontology of computational systems and their verification

By giving a careful reading of the outlined debate one may notice that the methodological and epistemological problem of program verification is related to an ontological one, namely the nature of computational systems. This is particularly evident in Fetzer's analysis with his distinction between programs as algorithms and programs as causal structures of algorithms. Primiero (2020) moves from this assumption to recompose the whole debate

[8] Any programming language involving *while* or *for* loops as a construct must include elementary arithmetic. See on this Van Leeuwen 1990.

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

on program correctness in light of a stratified ontology of computational systems; this, in turns, allows the author to supply a different answer to the vexed question. By developing the notion of *Level of Abstraction* (LoA) of (Floridi 2008), Primiero provides an ontology of computational systems as defined by the following hierarchy of LoAs: intention – algorithm – high-level programming language instruction set – machine code operation set – execution. Ideally, a computational system is developed by first advancing a set of functional requirements, here the intention, which are subsequently fulfilled by an algorithm, the latter is then implemented in a high-level language program, which is compiled into a machine language program and finally executed.

The three paradigms of computer science defined in (Eden 2007), namely the mathematical, scientific, and engineering paradigms, are called *foundations* in Primero (2020) and related to different LoAs; in this way they are not intended as mutually exclusive. A mathematical foundation is given at the algorithm LoA, the only level wherein, according to Fetzer (1988), mathematical correctness can be guaranteed for program verification.

An engineering foundation of computing considers all the LoA defining a computational system. According to this view the whole of the system needs to be verified; each LoA, and not just the program, has to be shown to be correct with respect to the required functional requirements. This cannot be pursued by means of formal verification, but through software testing techniques.

The engineering foundation does not prevent one from accomplishing scientific knowledge of computational systems. This indeed is the case of the experimental foundation of computing which Primero (2020) considers for the algorithmic method: given a computational problem, it is the practice of advancing an algorithm as an hypothesis for the solution of that problem; the hypothesis is then evaluated by implementing the algorithm into a program, by executing the program and checking whether the executed outputs provide the correct solution to the initial problem. This clearly does not amount to providing formal verification of the computational system; the author argues that a *hypothetical-deductive method* is rather involved, wherein formal verification is combined with inductive reasoning.

This very last point has been developed by (Angius 2013, 2020). First, it is argued that program verification alone cannot guarantee for program correctness. The reason comes from an application of Fetzer's (1988) argument to the case of model-checking. State transition systems and automata used in model checking to represent non-trivial programs are characterized by the so-called *state explosion problem*, preventing the model-

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

checking algorithm to be executed with reasonable time and space resources. To overcome such a problem, models are abstracted, using only a subset of the actual program variable set, and idealised, introducing false, simplifying, assumptions.[9] Data abstraction techniques are known to produce *over-approximations* and *under-approximations*, that is, models that respectively contain paths not corresponding to actual executions or that lack some paths corresponding to actual executions. Suppose now that an abstract model was checked against a given specification and that the procedure terminated with a negative answer: the algorithm outputs a *counterexample*, that is, a path in the model showing a violation of the given specification. In case of an over-approximation such path may be a *false-negative*, a counterexample not corresponding to some actual program execution. In order to ascertain whether this is the case, the program has to be tested against that counterexample: the program is prompted to try to make it execute the counterexample. Therefore, formal verification requires testing.

At the same time, and in line with (Symons and Horner 2014), it is emphasised how testing alone does not guarantee program correctness either. Specifications expressed as universally quantified statements cannot be justified by any finite number of observations of executions fulfilling them: an incorrect run may be carried out at any subsequent execution. A limitation similar to the enumerative induction problem in empirical sciences requires that models of the program be used to identify paths violating the specification under examination; the program is then tested against those paths. Therefore, testing requires formal verification.

The conclusion is that both deductive and inductive reasoning are necessary to determine program correctness. This practice, known as model-checking automated software testing (Callahan *et al.* 1996), can be identified with the *model-based reasoning approach* in science (Magnani *et al.* 1999), wherein an empirical system is represented by a scientific model, deductive reasoning is applied to the model to derive hypotheses about the target system, and finally those hypotheses are justified inductively by experimenting on the system.

5. Conclusions and future directions for the program verification debate

The problem of whether computer programs can be formally verified is rooted in the dual ontology of computational systems: they are both abstract and

[9] Common simplifying assumptions include the correctness of the implementing hardware and the appropriate interaction of the system with users and environment.

APhEx
Rivista Italiana di Filosofia Analitica
ISSN 2036-9972

concrete and, as any other artefact, they are defined by *functions* and *structure* (Turner 2018). Computable functions are realised as mathematical entities, viz. algorithms and automata; computing structures, such as high-level or language programs and implementing machines are physical structures executing those functions. This paper reconstructed the debate on program verification in light of such a dichotomy. Those, such as Hoare, holding that program correctness is a matter of mathematical analysis focus on the functional properties of programs. While those, Fetzer in the first place, arguing that programs cannot be verified emphasise the structural properties of programs which cannot be but the object of empirical investigation.

Whereas that of program verification is still an open problem, this paper suggested how one way out is replacing the dual ontology of programs by the stratified LoA ontology provided by Primiero (2020), wherein each LoA is considered functional for the lower ones and structural for the higher levels. The outcome is that mathematical and empirical analysis are combined to assess program correctness. Deductive reasoning in program verification is indeed limited, as underlined by its detractors, but, at the same time, one cannot do without it! And empirical testing alone is not sufficient to evaluate the reliability of programs. Deductive and inductive reasoning are rather combined in the actual practice of evaluating the correctness of programs.

The program verification debate is likely to see a renewed phase in the upcoming years, due to the unexpected flourishing of deep neural networks that, in the last five years, have been successfully applied to many learning tasks, including vision, autonomous driving, decision making, and language generation (Plebe and Grasso 2019). Verification is about establishing, either formally or empirically, whether a computational system complies with a set of functional requirements. However deep learning (DL) systems are not developed so as to comply with specified functions but so as to *learn* the desired functions; additional, non requested, functions are learnt as well, and they may turn out to be advantageous but also harmful for the developed systems. In DL, functions depend more on the dataset used to train the network rather than on specifications, which are not even formally advanced (Angius and Plebe 2023). Examples of unexpected, performed functions include autonomous cars that crash over a vehicle that was identified as a non-moving object (such as a building), or decision support systems that perpetrate biases over ethnic minorities (Muller 2023).

The challenges that DL poses for formal verification are related to the fact that program correctness cannot be defined anymore as a formal relation between a model of the system and a specification set. On the one hand, there is not a properly specified specification set; on the other hand, due to the well-

known opacity and non-interpretability of DL networks, it is hard to build models thereof (Lipton 2018). Potential approaches to tackle the verification of DL systems include the construction of probabilistic models of networks and to apply probabilistic model checking (Termine *et al.* 2018); the lack of formal specifications is sometimes addressed by providing formal statements formalizing functional input-output relations that have been observed while testing the system (see for instance Liu *et al.* 2021).

The next few years will reveal whether formal verification will be successfully applied to DL networks; in the meanwhile, philosophical inquiry is requested to advance an ontology of DL systems and consequently to define what should count as a proof of a neural network correctness.

References

Ammann, P., Offutt, J., 2016, *Introduction to software testing*, Cambridge, Cambridge University Press.

Angius, N., 2013, «Model-based abductive reasoning in automated software testing», *Logic Journal of IGPL*, 21, 6, pp. 931-942.

Angius, N., 2020, «On the mutual dependence between formal methods and empirical testing in program verification», *Philosophy & Technology*, 33, 2, pp. 349-355.

Angius, N., Plebe, A., 2023, «From Coding To Curing. Functions, Implementations, and Correctness in Deep Learning», *Philosophy & Technology*, 36, 3, p. 47.

Angius, N., Primiero G., Turner, R., 2021, «The Philosophy of Computer Science», *The Stanford Encyclopedia of Philosophy* (Spring 2021 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/spr2021/entries/computer-science/>.

Arkoudas, K., Bringsjord, S., 2007, «Computers, Justification, and Mathematical Knowledge», *Minds and Machines*, 17, 2, pp. 185-202.

Baier, C., Katoen, J. P., 2008, *Principles of model checking*, Cambridge (CA), MIT press.

Barwise, J., 1989, «Mathematical proofs of computer system correctness», *Notices of the American Mathematical Society*, 36, pp. 844-851.

Callahan, J., Schneider, F., Easterbrook, F., 1996, «Automated software testing using modelchecking», in Gregoire, J. C., Holzmann G. J., Peled, D.(eds), *Proceeding Spin Workshop*, pp. 118-127. Rutgers.

Clarke, E., Grumberg, O., Peled, D., 1999, *Model Checking*, Cambridge (CA), The Mit Press.

Curry, H. B., 1934, «Functionality in combinatory logic», *Proceedings of the National Academy of Sciences*, 20, 11, pp. 584-590.

de Bruijn, N. G., 1980, «A survey of the project Automath», in *To HB Curry: Essays on combinatory logic, lambda calculus and formalism* (pp. 579-606), Academic Press Inc.

De Millo, R. A., Lipton, R. J., Perlis, A. J., 1979, «Social processes and proofs of theorems and programs», *Communications of the ACM*, 22, 5, pp. 271-280.

Dijkstra, E. W., 1975, «Correctness concerns and, among other things, why they are resented», *ACM SIGPLAN Notices*, 10, 6, p. 546.

Eden, A. H., 2007, «Three paradigms of computer science», *Minds and machines*, 17, pp. 135-167.

Fetzer, J. H., 1988, «Program verification: The very idea», *Communications of the ACM*, 31, 9, pp. 1048-1063.

Floridi L., 2008, «The method of levels of abstraction», *Minds and machines* 18, 3, pp. 303-329

Floyd R., 1967, «Assigning meanings to programs», *Proc. Symp Appl Math Math Asp Comput Sci*, 19, pp. 19-32.

Garoche, P. L., 2019, *Formal verification of control system software*, Princeton, Princeton University Press.

Gödel, K., 1964, «Postcriptum», in Davis M. (ed.), *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, pp. 71–73, New York: Raven Press.

Gordon, M., 1987, «A Proof Generating System for Higher-Order Logic», *Univ. of Cambridge Computing Laboratory Tech* (No. 103), Report.

Harrison, J., 2003, «Formal verification at Intel», in *18th Annual IEEE Symposium of Logic in Computer Science, 2003, proceedings.* (pp. 45-54), IEEE.

Hoare, C. A. R., 1969, «An axiomatic basis for computer programming», *Communications of the ACM*, 12, 10, pp. 576-580.

Hoare, C. A. R., 1985, «The mathematics of programming», in *International Conference on Foundations of Software Technology and Theoretical Computer Science* (pp. 1-18). Berlin, Heidelberg: Springer Berlin Heidelberg.

Hogg, R., McKean, J., Craig, A., 2005, *Introduction to Mathematical Statistics*. 6th edition. Pearson.

Howard, W. A., 1980, «The formulae-as-types notion of construction», in *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44, pp. 479-490.

Jones, C. B., 1990, «Systematic software development using VDM», *Prentice Hall International Series in Computer Science*.

Lipton, Z. C., 2018, «The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery», *Queue*, 16, 3, pp. 31-57.

Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., Kochenderfer, M. J., 2021, «Algorithms for verifying deep neural networks», *Foundations and Trends in Optimization*, 4, 3-4, pp. 244-404.

Löwenheim, L., 1915, «Über Möglichkeiten im Relativkalkül», *Mathematische Annalen* 76, 4, pp. 447-470.

Luo, Z., Pollack, R., 1992, *LEGO proof development system: User's manual*, LFCS, Department of Computer Science, University of Edinburgh.

Magnani, L., Nersessian, N., Thagard, P. (eds.), 1999, *Model-based reasoning in scientific discovery*. Springer Science & Business Media.

Morris, F. L., Jones, C. B., 1984, «An early program proof by Alan Turing», *IEEE Annals of the History of Computing*, *6*(02), 139-143.

Mostowski, A., Robinson, R. M., Tarski A., 1953, «Undecidability and essential undecidability in arithmetic», in Tarski, A., Mostowski, A., Robinson, R. M., *Undecidable Theories*, Dover reprint.

Müller, V. C., 2023, «Ethics of Artificial Intelligence and Robotics», in *The Stanford Encyclopedia of Philosophy* (Fall 2023 Edition), Edward N. Zalta & Uri Nodelman (eds.), URL = <https://plato.stanford.edu/archives/fall2023/entries/ethics-ai/>.

Newell, A., Simon, H. A., 1976, «Computer Science as Empirical Inquiry: Symbols and Search», *Communications of the ACM*, 19, 3, pp. 113-126.

O'Hearn, P. W., Pym, D. J., 1999, «The logic of bunched implications», *Bull Symb Log* 5, 2, pp. 215-244.

Plebe, A., Grasso, G., 2019, «The unbearable shallow understanding of deep learning», *Minds and Machines*, 29, pp. 515-553.

Primiero, G., 2020, *On the foundations of computing*, Oxford, Oxford University Press.

Rapaport, W. J., 2023, *Philosophy of Computer Science: An Introduction to the Issues and the Literature*, Hoboken, John Wiley & Sons.

Reynolds, J. C., 2002, «Separation logic: a logic for shared mutable data structures», in: *Proceedings 17th annual IEEE symposium on logic in computer science*, IEEE, pp 55-74.

Skolem, T., 1920, «Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen», *Videnskapsselskapet Skrifter, I. Matematisknaturvidenskabelig Klasse*, 6, pp. 1-36.

Spivey, J. M., 1989), «An introduction to Z and formal specifications», *Software Engineering Journal*, 4, 1, pp. 40-50.

Symons, J., Horner, J., 2014, «Software intensive science. «*Philosophy & Technology*», 27, pp. 461-477.

Symons, J., Horner, J. K., 2020, «Why there is no general solution to the problem of software verification», *Foundations of Science*, 25, pp. 541-557.

Tedre, M., 2011, «Computing as a science: A survey of competing viewpoints», *Minds and Machines*, 21, pp. 361-387.

Termine, A., Antonucci, A., Facchini, A., Primiero, G., 2021, «Robust model checking with imprecise Markov reward models», in *International Symposium on Imprecise Probability: Theories and Applications*, pp. 299-309, PMLR.

Turing, A. M., 1949, «Checking a large routine», In *Report of a Conference High Speed Automatic Calculating-Machines, 22-25 June 1949*. University of Cambridge, Mathematical Laboratory, 1950.

Turner, R., Turner, R., 2018, *Computational artifacts*, Springer Berlin Heidelberg.

Tymoczko, T., 1979, The four colour theorem and its philosophical significance, *The Journal of Philosophy*, 76, 2, pp. 57-83.

Van Leeuwen, J., 1990, *Handbook of theoretical computer science. Volume B: formal models and semantics*, Elsevier and MIT.

Wittgenstein, L., 1958, *Philosophical Investigations*, Third Edition. New York: Macmillan.

Varzi, A., 2019, «Mereology», in *The Stanford Encyclopedia of Philosophy*, Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/spr2019/entries/mereology/>.

Williams, D.C., 1953, «On the Elements of Being: I», *Review of Metaphysics*, 7, 1, pp. 3-18.