Philosophical Problems in Software Engineering

A Personal Perspective

Jack K. Horner

Overview

- Objective
- What is software engineering?
- Software engineering standards

Objective

- To survey some entangling alliances (Washington 1796) between
  - Epistemology
  - Ethics
  - Metaphysics
  - Ontology
  - Logic

and software engineering

What is software engineering?

A widely used software engineering standard (ISO 2017) says, in effect, that software engineering is

An integrated set of software development

- practices
- procedures
- artifacts

that optimizes on some set of objectives (e.g., minimizing cost, schedule, risk)

Does software engineering have any philosophical content?

- *Advocatus diaboli*
  - Almost no software engineer believes s/he has to address philosophical topics
  - Almost no philosopher of computing pays attention to software *engineering* as such (Winsberg 2010; Oreskes, Shrader-Frechette, and Belitz 1994 are two exceptions)
  - “Shut up and calculate!” (attributed to Richard Feynman (Mermin 2004))
- What would Plato do in a jam like this? (Kevin Kline in *A Fish Called Wanda,* Cleese 1988)

What is software engineering?

- ISO 2017 presumes we have a reasonably good idea of
  - What software is
  - What might count as “objectives”
  - How given development practices, procedures, and artifacts *optimize* on a given set of objectives

What is software? (What various people have said)

- A codification, in a computer language, of something that can be executed on a physical computer (paraphrase of Piccinini 2015) [Issue: this requires us to say what a physical computer is]
- (A program is) a proof (Curry/Howard 1969) [Issue: not all programs are proofs, unless we define "proof" in a pathologically narrow way]
- (A simulation is) like an experiment (Morrison 2009, 2015)
- (A simulation is) like measurement practices (Morrison 2009)
- (Computer simulation) software is like a scientific instrument (Alvarado 2020)

What is software? (What various people have said, cont’d)

- (Computer models are) a formal extension of mathematical representation (Weisberg 2012 and Pincock 2011) [Issues: some AI software is not consistent with probability theory. What formal extension, specifically, is involved?]
- Software involves representations and imaging (Barberousse and Vorms 2009) [Issues: what do we mean, precisely, by “representation” (the entire history of epistemology?) and “imaging”?]
- (At least some) Software involves hypothesis generation testing [Issue: not all does]
- Among other things, a motley collection of computer languages, models, and engineering rules of thumb (Winsberg 2010)

What is software? (What various people have said)

- Strong pancomputation: Every physical system is a computer (Putnam 1967) [Issue: this implies that a mere rock is a computer. Is a "So what?" reply adequate?]
- Computing (including software) is purely syntactic in nature (a view motivated by Stich 1983). [Issue: this view doesn't explain how we individuate things that are, and are not, computers, in solely syntactic terms. It seems we have to invoke something outside of syntactics proper to make those distinctions]

What is software? (the Turing view)

- A set of instructions written in a Turing-complete (Turing 1936) language (e.g., the lambda calculus; all modern standardized computing languages such as C/C++, Java, Ada, Fortran)
- A Turing-complete language is one that can characterize a Turing machine (see Boolos, Burgess, and Jeffrey 2007 for a definition)
- This is the most commonly used definition of "computer" and "computer language"
- In the next few slides, let's look at whether the Turing-machine idiom captures everything we mean by "computable"

Does the Turing-machine idiom capture everything we mean by “computable”?

- Let’s assume that any adequate conception of computation must at least capture arithmetic on positive integers
- A function *f* from positive integers to positive integers is called *effectively computable* if a list of instructions can be given that in principle make it possible to determine the value of *f*(*n*) for any positive integer *n*
- The instructions must be completely definite and explicit
- This definition is hand-waving in various ways. For example, the notions of “completely definite” and “explicit” are relative to something more fundamental (such as a formal language or set-theoretic notions)
- The definition above is rigorous enough for the purposes of this talk

Does the Turing-machine idiom capture everything we mean by "computable"?

- A numerical function of *k* arguments is *Turing computable* if there is some Turing machine that can compute it
- A function (e.g., addition, subtraction) that is effectively computable on an idealized abacus is called an *abacus computable* function.   Abacus computable functions are Turing computable. (See Boolos, Burgess, and Jeffrey 2007, Chap. 5 for a details.)
- A *recursive function* (see Boolos, Burgess, and Jeffrey 2007, Chap. 6) of a given kind K is a function that is defined by (a) a set of primitive functions (sometimes called the "basis" or the "basic" functions), together with (b) function-schema that tell us how to generate all other members of the K, starting with the basic functions.
- A *recursively computable function* is an effectively computable recursive function. Ordinary arithmetic, for example can be defined in terms of recursively computable functions (see Chang and Keisler 2012, 42). Recursive arithmetic functions are abacus computable, so are Turing compatible.

Does the Turing-machine idiom capture everything we mean by “computable”?

- All Turing computable functions are recursively computable
- *Turing’s thesis* is that any effectively computable function is Turing computable
- *Church’s thesis* states that the recursive computable functions are the effectively computable functions.
- Turing’s thesis – that all effectively computable functions are Turing computable – is equivalent to Church’s thesis (see Boolos, Burgess, and Jeffrey 2007, Chap. 6; Piccinini 2015, Chaps. 15-16).

Does the Turing-machine idiom capture everything we mean by “computable”? (The Halting Problem)

- Define a function (called the *halting function*), *h*, as follows. *h*(m,n) = 1, or 2, depending, respectively, on whether machine m, started with input n, eventually halts, or not. The Halting Problem is to find an effective procedure that, given *any* Turing machine (call that machine m), and given any positive integer n, will enable us to determine whether m, given n as input, ever halts.
- The Halting Problem cannot be solved on a Turing machine (Boolos, Burgess, and Jeffrey 2007, p. 40)

Does the Turing-machine idiom capture everything we mean by “computable”? (with a nod to logic)

- It can be shown that there is an effective procedure for producing, given any Turing machine M and an input n to M, a set of sentences G and a sentence D such that M given n will eventually halt if and only if G implies D.
- The question of whether there is an effective procedure for determining whether any given finite set of sentences implies another given sentence is called the *decision problem* (Tarski 1953, 3; Turing 1936; Turing 1938)
- It follows that if there were an effective procedure for deciding when a finite set of sentences implies another sentence, then the halting problem would be solvable.

Does the Turing-machine idiom capture everything we mean by “computable”?

- The halting problem is not solvable on a Turing machine.
- Turing’s thesis -- that any effectively computable function is Turing computable -- thus implies that the decision problem is not effectively computable, because it is not effectively computable on a Turing machine (for details, see Boolos, Burgess, and Jeffrey 2007, Chap. 11; Piccinini 2015, Chap. 15).

Do Turing-machine concepts capture everything we mean by "computable"?

- Certain non-Turing constructs, called *hypercomputers*, that are claimed to be able to solve the Halting Problem, have been proposed (for a somewhat dated introduction to the topic, see Ord 2006)
- Davis 2003 argues, in effect, that several characterizations of "hypercomputer" implicitly *assume* that a hypercomputer can solve the Halting Problem. If so, arguing that a hypercomputer can solve the Halting Problem is circular
- Turing 1938 argues (barely) that whatever a hypercomputer is, it is not a machine

Software has an identity problem (a nod to metaphysics)

- What do we mean when we say the *same* software runs on different physical computers?
- "War Story 1"
  - The Ballistic Missile Early Warning System (BMEWS) was a collection of large radars and associated computer systems designed to detect an over-the-Pole ICBM attack. In the early 1980s, the BMEWS software was re-engineered to run on a more modern computer
  - The acceptance criteria required the new system to produce the same outputs as the old, given the same inputs

Software has an identity problem

- War Story 1, cont’d
  - A engineering-oversight organization hired by the procurer (USAF) argued that in order to show that the new system produced the same outputs, given the same inputs, as the old, the new system would have to be a detailed micro-state-level emulator of the old system
  - This approach would have driven the cost and schedule of the new system beyond the procurer’s resources
  - The developer argued that
    - The requirements did NOT imply an emulator was required
    - The behavior of the old system was sometimes, without warning, non-deterministic, so the behavior of the old system was not, strictly speaking, even reproducible
    - The procurer conceded

Software has an identity problem

- War Story 2
  - One of the programs in BMEWS computed the trajectory of a missile
  - The output of the old system, for times close to missile launch, produced a “saw-tooth” trajectory. The new system, running the same software as the old, produced a smooth trajectory
  - The procurer argued that the new system’s output didn’t match the old system’s output

Software has an identity problem

- War Story 2, cont’d
  - The developer argued that the “saw-tooth” trajectory of the old system, which contained instantaneous right-angle changes of direction, was physically impossible (those accelerations would have required infinite power)
  - The procurer conceded
  - Post-mortem: the behavior of the algorithm used to compute the trajectory was sensitive to the size of a hardware-specific structure called a computer “word”. The original computer has a 36-bit word; the new system had a 60-bit word.

Software has an identity problem (with a nod to decision theory)

- War story 3
  - Late one night in early Spring 1982, the new and old BMEWS systems at Fylingdales Moor, North Yorkshire (UK), were running side-by-side on the same live radar data
  - The staff at the site was understandably skeptical of the new system because it had yet to prove its reliability
  - Suddenly, the old system reported an incoming ICBM
  - The new system reported, in contrast, a “ghost” -- a radar return from an aurora
  - Aurora or nuclear war: What would Kant do in a jam like this?

Software has an identity problem

War story 3, cont'd

- Fortunately, the site staff had decades of experience with this scenario. They knew
  - Auroras can produce radar returns, some of which look like missile tracks
  - Auroral activity is cyclical (period = ~11 years), and was near its peak at the time
  - An ICBM attack consisting of a single missile was extremely unlikely
  - The way the software determined whether something was an attack depended on how many times a particular segment of the software could be run in ~0.03 second. The more times that segment could be run in that interval, the better the characterization.
  - The newer machine could run the segment about 100 times faster than the old machine could.
- The site staff decided to believe the new system. Minutes later, they shut off the old system permanently

Software has an identity problem (COVID-19 example)

- The Imperial College London (ICL) COVID-19 simulator (ICL 2020) has been widely used in the UK and US to justify interventions (masking, social distancing, school closures) during the pandemic. Whether the simulator produces correct results has fundamental public-health and socioeconomic consequences
- The ICL simulator team made available to the public two versions of “the” simulator, neither of which is identical to the version used for official public-policy decision-making

Software has an identity problem (COVID-19, cont'd)

- Do the results of the three versions differ in any important way?
  - The ICL team says it doesn't guarantee anything about the publicly available versions
  - We have limited information about the outputs of the non-public version
  - Various authors (Horner and Symons 2020b; Eglen 2020; Rice, Wynne, Martin, and Ackland 2020) have evaluated the publicly available versions of "the" simulator and raised significant concerns about identity of the code and the reproducibility of its results

Does software need hardware?

- Can software be characterized without reference to hardware?
- It is theoretically possible to encode in a pure energy regime anything a Turing machine could do. This tells us that a physical realization of Turing machine does not in principle have to have a *material* character
- Suppose there were a purely mathematical function that could produce all the formal domain/range relationships that a physical computer could. There is nothing contradictory in such a notion. Why wouldn’t that function be a “computer”? (an idea suggested in Bradbury 1950)

Does software need hardware?

- Any computing functionality that is attributable to software could in principle be implemented in hardware alone. This suggests that at least the computing functionality of software is not inherently distinct from the functionality of certain hardware configurations.
- (Facetious?) “Really good software doesn’t need hardware” (Watkin 1995)

Software engineering standards

- A set of prescriptions for phased software development (ISO 2017). These phases are
  - Specification
  - Logical design
  - Physical design
  - Implementation
  - Verification (often called "Test")
  - Maintenance

Software engineering standards

The economic and risk-management rationale for a phase-structured approach to software development and management are based on two major premises (Boehm 1981, 38):

I. In order to create a “successful” software product, we must, in effect, execute all of the phases at some stage anyway
II. Any different ordering of the phases will produce a less successful software product

Software engineering standards (Rationale I)

- *We must execute all the phases at some point anyway.*
- Follows directly from questions that inevitably arise in the development of any software system:
  - What objectives must the software achieve? (Specification phase)
  - How do we ensure that everyone who helps to develop part the software understands how his/her part of the software integrates with the rest of the software? (Logical and physical design phases)
  - How do we determine that the software is doing what is supposed to do? (Verification/Test phase)

Software engineering standards (Rationale II)

- *If we execute the phases in any order but that described above, cost, schedule, and risk are higher.*
- Derives directly from empirical studies of the costs of fixing an error in a software system as a function of the phase in which the error is detected and corrected
- These studies show that in a large (> ~50,000 source lines of code (SLOC; Boehm, Abts, Brown, Chulani, Clark, Horowitz et al. 2000, 395)) or highly technical software project, a typical error is 100 times more expensive to correct in the maintenance phase than in the specification phase; in small projects (< ~10,000 SLOC), a typical error is 20 times more expensive to correct in the maintenance phase than in the specification phase (Boehm 1976; Boehm 1981, 40).

Software engineering standards

- Each of the development phases imposes requirements on, or equivalently, allocates requirements to, the processes and products of one or more successor phases.
- Taken end-to-end, the resulting requirements-allocation induces a hypergraph (Berge 1973) that spans the elements (documentation, processes, and code) in the system.
- Fodder for philosophy: in practice, there is no effective procedure for allocating requirements to products and processes

Software engineering standards

Fodder for philosophy, cont’d

- In theory, one could cast all the artifacts and processes of a phase-structured software development regime in a formalized language and force requirements allocation to be rendered as “derivations” in a formal derivation system expressed in that language (see, for example, Turner 2011; Perry et al. 2015)
- Magnusson 1990 proposed such a scheme for Ada. In that scheme, the specification was written as a system of Ada *package specifications* (a package specification in Ada is a formal construct of the Ada language).

Software engineering standards

Fodder for philosophy, cont’d

- Successful compilation of this system would demonstrate the consistency and completeness of the whole in terms of the Ada language definition. (A joke among Ada developers: “If you can get an Ada system to compile, there is no need to test it.”)
- Unfortunately, any such scheme requires stakeholders to be fluent in the formal system

Software engineering standards

- Documentation is crucial to ensuring the transparency, explainability, and reproducibility of software
- Even though this point seems self-evident, it is sometimes argued a software listing by itself is sufficient to determine what that software is *intended* to do. This view is incorrect because
  - The syntax and semantics of programming languages are far from sufficient to determine the intended application semantics (what the code is intended to do) of a given body of software.
  - Any program, regardless of what the code *seems* to be about, could be used solely to show that the machine on which it runs will in some sense cycle the program, without regard to anything else that program might be intended to do (JKH: mention two examples)

Software engineering standards

- Obviously, there is no guarantee that using a software development process of the kind described in this section will yield an error-free product
- Empirical studies of software error and its causes strongly suggest, however, that if such a framework is not used, with very high probability, software will contain at least 10 times as many errors as software developed within such a framework (Boehm 1973; Boehm 1976; Myers 1976; Boehm 1981, 40).

Specification

- The principal function of the specification phase of a software project is to generate an agreement (called the *specification*) among stakeholders that states what objectives a software system must achieve.
- Among other things, the specification is intended to reflect the results of the negotiation of stakeholder values, including *ethical* and *normative* considerations (see, for example, Horner and Symons 2020b)
- Fodder for philosophy: Ethical issues in computing is a vigorous area of research in philosophy of computing. (See, for example, any issue of *Minds and Machines* in the last 10 years.)

Specification (with a further nod to ethical considerations)

- There are various ways ethical considerations are related to software engineering
  - How the software is to be used, which often requires us to take into account what humans can, or are likely, to do
  - Whether we can trust the behavior of computing systems (this is largely an epistemological question, but it affects how we might choose to use computing systems in ethical contexts)
  - Whether a computing system could be an ethical agent
    - A variant of the Turing Test suggests the answer could be "yes"
    - But it is far from obvious that ethical notions are reducible to Turing-machine notions

Specification (ethics, cont’d)

- Rationale I for ISO 2017 is largely an empirical generalization based on cost and schedule metrics. Those metrics come from a collection of projects in which ethical considerations do not play a particularly deep role
- It’s not clear how well Rationale I, and therefore the argument for ISO 2017-like software engineering, would fare for non-cost/schedule optimizations

Specification, cont’d

- Optimizations in tradeoffs in software systems are
  - likely to be nonlinear, e.g., in order to optimize on the entire set of objectives, we may not be able to optimize each of the objectives (e.g., there are tradeoffs among time, money, and risk)
  - often only partially ordered – in general, it is not possible to compare optimizations across all contexts of interest

Logical design

- Objective: to generate an abstract description, called a Logical Design Document, of a system that satisfies the requirements of the specification.
- The abstract description that satisfies the specification assumes no particular implementation in hardware, software, or human procedures.
- Various languages can be used to express the logical design. In current practice, the Unified Modeling Language (see, for example, Rumbaugh, Jacobson, and Booch 1999) is often used for this purpose
- Fodder for philosophy: in practice, the mapping between the specification and the logical design is almost never well-defined in an effective procedure

Physical design

- Objective: to generate a concrete description, typically called the Physical Design Document, or Detailed Physical Design Document, of how specific machines, software, and human processes, and their interactions, will satisfy the requirements allocated to them from prior phases
- The software-specific component of the Physical Design Document is often called the Software Design Document, or SDD
- No software is generated is generated during this phase
- Fodder for philosophy: in practice, the mapping between the requirements allocated from prior phases and the physical design is almost never well-defined in an effective procedure

Implementation

- Objective: to implement on actual machines, and in software and human procedures, an operational product that satisfies the requirements allocated to it from prior phases.
- The software developed during the implementation phase is typically required to satisfy certain programming-language-specific standards (sometimes called “coding guidelines”), that are inherited by allocation from the specification phase. These standards prescribe programming-language-specific practices that are, and proscribe practices that are not, acceptable
- Fodder for philosophy: there is intense debate about what programming-language-specific standards should be enforced

Verification

- Objective: to determine whether the product generated in the Implementation phase satisfies all requirements allocated to the software. This can be viewed as part of the question of whether we should trust software (Alvarado 2020; Boschetti and Symons 2011; Boschetti, Fulton, Bradbury, & Symons 2012)
- Verification is typically performed at various software-build levels.
- Fodder for philosophy: This phase is fraught with philosophical problems, as the following slides identify in more detail

Verification (the Halting Problem, again)

- A specification requiring that a software system S
    - be a Turing machine, and
    - contain the equivalent of the halting function

cannot be verified because the Halting Problem

cannot be solved on a Turing machine

Verification issues (Symons and Horner 2014)

- *Could we test all paths in a program?* (a path is a sequence of instructions that can be executed in a program, beginning with program start and ending with program exit)
- Testing all paths in a program is in general intractable for all but the smallest (< ~300 SLOC) programs.
- Assume a 1000-SLOC program, with one binary branch ("if X, do Y"), on average, per 10 lines, and assume all branches are on all paths
  - The number branches in the code is $2^{1000/10} = \sim 10^{30}$
  - If we could formulate, execute, and evaluate one test per second, it would take 10^{17} lifetimes of the universe to execute all paths in this program
- A 1000-SLOC program is tiny by today's standards (e.g., Windows 10 is about 15 million SLOC)

Verification

- *Can we test parts of programs in parallel?*
- Not all programs can be decomposed to independently verifiable components. For example, large hydrodynamic simulators are state-history-sensitive: we have to run the computation, from initial conditions, for an unknown time (Kuzmin and Hämäläinen 2014)
- Even if we could parallelize tests, coordination of the results would be speed-of-light-limited
- We can always imagine a program large enough that it could not be tested in the lifetime of the Universe because of speed-of-light coordination limitations

Verification issues (Symons and Horner 2017, 2020a)

- *Could we (statistically) verify a program using conventional statistical inference theory* (CSIT, Hogg, McKean, and Craig 2005)*?*
- Verification is equivalent to characterizing the distribution of errors in a program
- CSIT requires that the variables of interest (here, errors) be characterized in terms of *random* variables (Hogg, McKean, and Craig 2005)
- Using random variables to verify a program requires us to know, *logically priori to* performing tests of statistics defined in terms of (estimators of) those variables, what the execution-path-space (= the set of all possible paths) is (Hogg, McKean, and Craig 2005; Chung 2001, Section 3.1)
- But we have no method of determining what that path-space is without first *empirically* discovering the paths in the software. That problem effectively backs us into a problem that scales the same way that executing all the paths in the code does
- Thus, CSIT cannot, in general, characterize such errors in all programs of interest

Verification issues (Horner and Symons 2019)

- *Could we verify programs by building them in such a way that they are provably correct (an approach generically called "model checking"* (see Clarke, Henzinger, Veith, and Bloem 2018; Perry et al. 2015)*?*
- This approach has had some notable successes. For example, it revealed that a simulator used to determine whether a building could survive a large earthquake gave, in one case, a fatally wrong result
- In practice, however, all applications of model checking to date have involved the use of software development environments that contain millions of lines of code (e.g., operating systems, compilers, editors) that were not developed using model-checking

Verification (Horner and Symons 2019)

- *Can finite agents exhaustively verify programs?*
- Assume that
  - In order for a software system S to satisfy a specification H, there must be a homomorphism from the set of models (Chang and Keisler 2012) of H to the set of models of S, and
  - H requires S to implement Robinson arithmetic (a subtheory of ordinary arithmetic)
- The Löwenheim-Skolem Theorem implies that there are an infinite number of non-isomorphic models of arithmetic
- Thus, to exhaustively verify S, we must verify that an infinite number of non-isomorphic models of arithmetic are isomorphic to a subset of the models of S
- This cannot be achieved in less than an *infinite* number of verification steps. No agent capable of performing only a finite number of verification actions (each of which takes at least some finite time, t_{min}, to perform) could perform such a verification
- Almost every program is required to implement ordinary arithmetic, so this is a limit (for finite agents) to the verification of almost every program

Verification (are simulators special?)

- Simulators often cannot, for various (e.g., ethical, monetary, time-critical) reasons, be verified by empirical experiments. What does it mean to verify a simulator in those contexts?
- *A thorny case*. In the US, large (100K – 1M SLOC) simulators are used to verify the safety and efficacy of nuclear weapons (NNSA 2016). These simulators are verified, in part, by comparing their results to those of other simulators. Is this an infinite regress, or is there, a la Aquinas 1270 (Pt. I, Q2, Art. 3), a "First Simulator"?
- Is this concern more about how we make justifiable inferences in the absence of conclusive information (the problem of induction), than it is about whether simulators present distinctive verification problems?

Maintenance

- This phase iterates the phases described above after the product is deployed, as needed
- Maintenance policies and procedures are documented in a Maintenance Manual
- If maintenance is “so conceived and so dedicated” (Lincoln 1863), any philosophical problem in the maintenance phase is no more than a variant of a philosophical problem in prior phases

Summary and conclusions

- Software engineering intersects in substantive ways with
  - Ethics
  - Epistemology
  - Ontology
  - Logic
  - Metaphysics

Acknowledgements

- This talk benefited from discussions with
  - John Symons
  - Ramón Alvarado
  - Tony Pawlicki
  - Dick Frank
  - Dick Stutzke
  - Tim Beeson
  - Larry Cox
- For any errors that remain, I am fully responsible

How to get a copy of these slides

- Download (in PDF) from my personal website:
  - http://jkhorner.com/PHILOSOPHY/Philosophical_Problems_in_Software_Engineering.pdf or
  - Access http://jkhorner.com/, select "PHILOSOPHY" on the splash page, then click on the relevant link on the resulting page
- Or contact me by email: jhorner@cybermesa.com

References

- Alvarado, R. (2020). *Computer Simulations as Scientific Instruments* (PhD. Diss. University of Kansas).
- Aquinas. (~1270, first published 1485). *Summa theologica*. Vol. I. Trans. by Fathers of the English Dominican Province, 1948. Christian Classics.
- Barberousse, A., & Vorms, M. (2014). About the warrants of computer-based empirical knowledge. *Synthese* 191.15, 3595-3620.
- Berge, C. (1973). *Graphes et Hypergraphes*. English translation: *Graphs and Hypergraphs*. North-Holland Publishing Company.

References

- Boehm, B.W. (1973). Software and its impact: a quantitative assessment. *Datamation*, May 1973, 48-59.
- Boehm, B. W. (1976). Software engineering. *IEEE Transactions on Computers*, December 1976, 1226-1241.
- Boehm, B. W. (1981). *Software Engineering Economics*. Upper Saddle River NJ: Prentice-Hall.
- Boehm, B. W., Abts, C., Brown A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D., & Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River NJ: Prentice-Hall.

References

- Boolos, G. S., Burgess, J. P., & Jeffrey R. C. (2007). *Computability and Logic.* Fifth Edition. Cambridge.
- Boschetti, F., & Symons, J. (2011). Why models' outputs should be interpreted as predictions. In *International Congress on Modelling and Simulation (MODSIM 2011)* MSSANZ: Perth, WA.
- Boschetti, F., Fulton, E., Bradbury, R., & Symons, J. (2012). What is a model, why people don't trust them and why they should? In M. R. Raupach (Ed.), *Negotiating our future: Living scenarios for Australia to 2050* (pp. 107–118). Australian Academy of Science.
- Bradbury, R. (1950). *The Martian Chronicles*. Doubleday.
- Cleese, J. (1988). *A Fish Called Wanda.* Metro-Goldwyn-Mayer. Film.

References

- Chang, C. C., & Keisler, H. J. (2012). *Model Theory.* Third Edition. Dover.
- Chung, K. L. (2001). *A Course in Probability Theory.* Third Edition. Academic Press.
- Clarke, E.M., Henzinger, T. A., Veith, H., & Bloem, R., eds. (2018). *Handbook of Model Checking*. Springer.
- Davis, M. (2003). The Myth of Hypercomputation. In A. Shlapentokh (ed.). *Miniworkshop: Hilbert's Tenth Problem, Mazur's Conjecture and Divisibility Sequences*. MFO Report. 3. Mathematisches Forschungsinstitut Oberwolfach.
- Eglen, S. (2020). CODECHECK report comparing ICL 2020c and some tables in ICL 2020b. https://zenodo.org/record/3865491#.XuIc-W5FyUk. Accessed 11 June 2020.
- Hogg, R. V., McKean, J. W., and Craig, A. T. (2005). *Introduction to Mathematical Statistics.* 6th edition. Prentice Hall.

References

- Horner, J. K., & Symons, J. F. (2020a). What Have Google's Random Quantum Circuit Simulation Experiments Demonstrated about Quantum Supremacy? Forthcoming in Hamid R. Arabnia, Leonidas Deligiannidis, Fernando G. Tenetti, and Quoc-Nam Tran, eds. *Advances in Software Engineering, Education, and e-Learning*. Springer Nature. A draft is available at arXiv:2009.07359.
- Horner, J. K., & Symons, J. F. (2020b). Software engineering standards for epidemiological modeling. Forthcoming in *History and Philosophy of the Life Sciences*. A draft can be obtained from arXiv:2009.09295.
- Howard, W. A. (1980) [original paper manuscript from 1969]. "The formulae-as-types notion of construction", in Seldin, J. P., & Hindley, J. R. (eds.), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press. pp. 479–490.

References

- Imperial College London (ICL). (2020). https://github.com/mrc-ide/covid-sim/blob/master/src/. Accessed 10 May 2020.
- ISO/IEC/IEEE. (2017). *ISO/IEC/IEEE 12207:2017. Systems and software engineering – Software life cycle processes.* https://www.iso.org/standard/63712.html. Accessed 26 May 2020.
- Kuzmin D and Hämäläinen J. (2014). *Finite Element Methods for Computational Fluid Dynamics: A Practical Guide.* SIAM.
- Lincoln, A. (1863). The Gettysburg Address. In Boritt, G. (2008). *The Gettysburg Gospel: The Lincoln Speech That Nobody Knows.* Simon & Schuster.

References

- Magnusson, J. (1990). Personal communication.
- Mermin, N. D. (2004). Could Feynman have said this? *Physics Today* 57, 5, 10. https://doi.org/10.1063/1.1768652.
- Morrison, M. (2009). Models, measurement and computer simulation: the changing face of experimentation. *Philosophical Studies* 143, 33-57.
- Morrison, M. (2015). *Reconstructing reality*. Oxford: Oxford University Press.
- National Nuclear Security Administration (NNSA). (2016). Maintaining the Stockpile. https://www.energy.gov/nnsa/missions/maintaining-stockpile. Accessed 24 October 2020.
- Ord, T. (2006). The many forms of hypercomputation. *Applied mathematics and computation* 178.1, 143–153.

References

- Oreskes, N., Shrader-Frechette, K., & Belitz,K. (1994). Verification, Validation, and Confirmation of Numerical Models in the Earth Sciences. *Science* 263, 641–646.
- Alexander, P., Pike, L., Loscocco, P. G., & Coker, P. G. (2015). Model checking distributed mandatory access control policies. *ACM Transactions on Information and System Security (TISSEC)* 2(18).
- Piccinini, G. (2015). *Physical Computation: A Mechanistic Account*. Oxford.

References

- Pincock, C. (2011). Modeling reality. *Synthese* 180, 19-32.
- Putnam, H. (1967). The mental life of some machines. In S. Hook (ed.), *Dimensions of Mind: A Symposium*. Collier. pp. 138-164.
- Washington, G. (1796). *The Address of Gen. Washington to the People of America on His Declining the Presidency of the United States.* American Daily Advertiser.
- Putnam, H. (1988). *Representation and Reality.* MIT Press.
- Rice, K., Wynne, B., Martin, V., and Ackland G. J. (2020). Effect of school closures on mortality from coronavirus disease 2019: old and new predictions. *BMJ* 2020, 371. https://doi.org/10.1136/bmj.m3588.
- Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language Reference Manual.* Addison-Wesley.
- Stich, S. (1983). *From Folk Psychology to Cognitive Science: The Case Against Belief.* MIT Press.

References

- Symons, J. F., & Horner, J. K. (2014). Software intensive science. *Philosophy and Technology* 27, 461-477.
- Symons, J. F., &Horner, J. K. (2017). Software error as a limit to inquiry for finite agents: challenges for the post-human scientist. In Powers, TM, ed. *Philosophy and Computing: Essays in Epistemology, Philosophy of Mind, Logic, and Ethics*. Springer. pp. 85-98.
- Symons, J. F., & Horner, J. K. (2019). Why there is no general solution to the problem of software verification. *Foundations of Science*. https://doi.org/10.1007/s10699-019-09611-w.
- Tarski, A. (1953). A general method in proofs of undecidability. In Tarski, A., Mostowski A., & Robinson R. M. *Undecidable Theories*. Dover reprint, pp. 1-35.
- Turing, A. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 42, 230–65.

References

- Turing, A. (1938). *Systems of Logic Based on Ordinals* (PhD thesis). Princeton University. doi:10.1112/plms/s2-45.1.161
- Turner R. (2011). Specification. *Minds and Machines* 21(2), 135–152. doi:10.1007/s11023-011-9239-x.
- Watkin, H. (circa 1995). Personal communication.
- Weisberg, M. (2012). *Simulation and similarity: Using models to understand the world*. Oxford.
- Winsberg, E. (2010). *Science in the age of computer simulation.* Chicago.