Why is it so difficult to ensure that software is reliable?

Abstract

How can we be certain that software is reliable? Is there *any* method that can verify the correctness of software for *all* cases of interest? Computer scientists and software engineers have assumed, that there is no fully general solution to the verification problem as an informal working hypothesis. In this paper, we survey approaches to the problem of software verification and offer a new proof for why there can be no general solution.

Introduction

In the computer science and software engineering communities the question of software verification is a central concern. Computer scientists have created various methods for at least partially checking the correctness of software. But could there be a *fully general* solution to the problem of verification? By a fully general solution we mean one that solves the verification problem for *all* cases of interest. Most computer scientists and software engineers have assumed as a working hypothesis that there is no such solution. In this paper, we demonstrate formally why this working hypothesis is correct.

The question of whether there is a general solution to the verification problem has two important aspects. The first is largely philosophical, in the sense that that it concerns the limits of human knowledge. The second concerns the class of possible software verification methods. This issue lies within the domain of theoretical computer science. While we argue that the verification problem has no general solution, it can be solved for relatively small (where "small" is defined below) software systems for reasons that we will explain. Specifically, we will argue that there are metalogical properties of the software verification problem that preclude a solution for all cases of interest.

The problem of software verification is relevant to epistemological questions concerning the role of computers in science. What degree of certainty are we entitled to expect of theories that depend in important ways on software? (See Boschetti et.al 2012). How does the kind of software intensive science that is currently ubiquitous differ in kind from non-software intensive science?

The verification problem also has ethical and legal consequences in situations where we must decide how much care needs to be taken in military, governmental, and commercial contexts to minimize software error. What level of software testing should we expect of a responsible manufacturer in cases where failure can lead to serious harms?

Error-free software in science and technology is ideal but as we will argue, we can never be certain that we have outside of a very restricted set of domains.

In this paper we examine approaches to the problem of error and verification in software engineering and theoretical computer science. To do this, we first state the verification problem (Section 1.0). We then critically examine possible solutions. In Section 2.0, we will explain why that verification-as-testing fails to serve as a general solution to the verification problem. Given this failure, we must turn to formal approaches, which are discussed in Section 3.0. The two principal kinds of formal approaches are axiomatic methods and model-theoretic methods. We explain why both fail to satisfy the desiderata that we believe any acceptable method for verification must satisfy. At present, axiomatic approaches are of merely historical interest to most practicing computer scientists for reasons we explain. Model-checking is a different matter. Building on the techniques of model-theory, model-checking is an approach that has produced impressive practical results and offers the best attainable purchase on the solution to the problem of verification.

Nevertheless, we will argue, any piece of software that is required to implement ordinary arithmetic, meaning virtually all business and scientific software, cannot be completely verified.

The arguments we present here are meant to close the door to a very specific kind of ambition. We acknowledge that probably no practicing scientist in the software verification community has the goal that we show is impossible. Our argument does two things. First it provides a principled formal reason in support of the commonsense assumptions concerning the limits of verification. In the scheme of things this is a less important contribution than our second point. Our work here takes an idealized vision of what is achievable via computational methods and software off the table. This is especially important for philosophical reflection on the role of computational methods in science and the ethics of technology. If our argument stands, one cannot generally assume that we have, or that we can reliably create, error-free software. These results demonstrate that the ideal of achieving error-free software should simply drop out of epistemological and ethical reflections on computational methods.

1.0 The verification problem

The problem of assessing the reliability of software can be thought of as the problem of showing the correctness of a software system. For the purpose of this paper, we use the term "software" to mean a sequence of instructions written in a computer language (e.g., C++, Java, Ada, etc.). Understanding the reliability or correctness of a software system involves determining whether that system satisfies a specification. Among other things, a specification represents the purpose for which the software is being developed. Specifications can be articulated with varying degrees of precision. Generally, the more precise the better. The problem of assessing whether the software system meets the specification is called the *software*

verification problem; from here on, we will refer to that problem more briefly as the *verification problem*. The verification problem, succinctly stated, is

(The verification problem). Given a software system *S* and specification *H*, determine whether *S* satisfies *H*.

The satisfaction relation in this context can be articulated in model-theoretic terms.[1] We will give a precise characterization of this relation in Section **3**. An intuitive understanding of 'satisfies' will suffice for our immediate purposes.

There are a wide variety of ways that software can go wrong (See Floridi, Fresco, and Primiero 2015;). Notice that our definition of the verification problem is minimal in the sense that it is not intended to rule out the system doing *more* than what we ask of it in the specification. There are cases where a piece of software behaves in ways not represented in *H* which would not count as errors on our minimal account. Doing more than the specification can be good or bad. For example an operating system does more, and is intended to do more, than what is generally specified. Most unspecified uses or functions of an operating system are not regarded as problematic. Similarly, a piece of Trojan software, such as a rootkit, can perform according to the specification that would ordinarily allow an administrator access to a system, but in so doing, it could also allows an adversary a backdoor into the system.[2]

Our characterization of the problem restricts the challenge of verification to the determination of whether the system satisfies the specification. This is not meant to be an exhaustive list of necessary and sufficient conditions for the ways that things can go wrong in software engineering or in the deployment of technology. Rather, it is a necessary condition for determining the correctness of *S* that it at least satisfy the specification *H*. By analogy, we can say that a functional heart is one that pumps blood throughout the body of an organism in a way that leads the organism to thrive. If it fails to do so, then we can say that it is not a functional heart. Whether the beating heart also has a pleasant sounding rhythm or can function as a symbol of romantic love is a not relevant to whether the heart is performing what philosophers of biology called its

[1] Characterizing the satisfaction relation in the verification problem in model-theoretic terms may seem to differ from the way some computer scientists characterize verification. Emerson (2008, 28), for example, suggests the verification problem is determining "whether or not the behavior of *M* meets the specification *h*" where *M* is the program and *h* is the specification. Our approach does not depend on behavioral properties, as such, of a program. Instead, we characterize the satisfaction relation in terms of a function that relates the models of a program/software system to models of the specification (see Section 3 for more detail).

[2] The root toolkit example shows how difficult it is in practice to distinguish some specification from implementation issues. In this case a specification issue – failure to prohibit Trojan horses – with an implementation issue – a piece of software that answers the specification requirement to allow administrator access to the system, but also allows malicious actors to compromise the system.Trojan To put that point more sharply, we could answer the requirement to cut a piece of wood with a saw or a stick of dynamite. Both would do the job; the dynamite would surely cause much collateral damage.

proper function (See for example Millikan 1989). Given our restriction we can characterize a necessary condition for software error as follows:

(DE) *An error in a software system* S *is a failure to satisfy* H.

Given (DE), verifying a software system *S* is equivalent to showing that *S* will not fail to satisfy *H* in the sense of (DE).

Verifying software correctness requires using a method. What should such a method look like? First, we would like a method that scales well, i.e., a method whose complexity increases no more than "proportional to the size of the software system". For example, the complexity of the method applied to a software system containing 200 "expressions" should be no greater than twice that of a software system containing 100 such expressions. We especially want to avoid situations in which the complexity of the method grows exponentially with the size of the software system. Second, we want a method that does not have to rely on lucky guesses or inspiration. Third, to help minimize the labor involved in the application of the method, it should be automatable. Fourth, we want the method to be able to handle the case in which more than one process is running at the same time since this is a feature of many modern software systems. Finally, we want a method that runs to completion in a time we care to wait.

To summarize, the following seem like reasonable features that we should expect from such a verification method. It should:

(D)[3]

- (D1) Scale less than exponentially in problem "size"
- (D2) Not have to rely on "inspiration" when applied
- (D3) Be automatable
- (D4) Capture concurrency[4]
- (D5) Complete its task in a time we care to wait.

Emerson 2008 (pp. 27 and 35) suggests some desiderata of adequacy that partially overlap (D3) and (D5) above. The criteria in Emerson 2008, however, are strongly (and in some particulars, exclusively) oriented to *formal* methods of verification (see Section 2.3 of the current paper).

[3] Collectively, D2, D3, and D5 significantly overlap what we mean when we say a verification procedure is *algorithmic*. In particular, an algorithmic procedure is an effective procedure, and an effective procedure by definition implies (D2), (D3), and (D5). One might, therefore, replace the union of (D2), (D3), and (D5) with a desideratum requiring an approach to verification to be algorithmic. Articulating (D) as shown, however, supports some informative distinctions among the relative strengths of approaches that have been taken to the verification problem, and for this reason we choose to adopt (D) in the more expansive form shown.

[4] Two programs, A and B, are concurrent if at least some portion of those programs execute at the same time.

In this paper, we show how the task of verification faces fundamental challenges in satisfying (D). To help explain these challenges, we survey existing approaches to the verification problem and draw some lessons. We emphasize that this survey (especially Section 3) provides only the level of detail required to support our general thesis. It does not, nor need it, given the arguments in Section 3, contain the kind of detail that would typically be included in a comprehensive, general technical review of verification methods.[5].

2.1 Verification and the Halting Problem

The theory of computation is the formal study of how computing systems compute. In order to ensure that this objective is well defined, computer science requires a clear characterization of a computing system. For the purposes of this paper we will restrict ourselves to the most widely accepted characterization of a computer: the Universal Turing Machine. A Universal Turing Machine is regarded as the minimal system that could serve as a general-purpose "computer" (Turing 1936; Boolos, Burgess, and Jeffrey 2007).

Given this interpretation of "computer" we already know, in one sense, that a fully general algorithmic solution to the verification problem is not possible. Here's why: Let *P* be an arbitrary program running on a Universal Turing Machine and let *H* be the requirement that *P* should eventually halt. Stated as a verification task, the requirement can be framed as follows:

> Given a software system *S* (*P* running on a *UTM*) and specification *H* (tell whether *P* halts), determine whether *S* satisfies *H*.

Turing proved that no algorithm that can determine whether *P* halts for *all* possible program input-output pairs for *P*. This is the Halting Problem (Turing 1936). Thus, for at least some cases, if the specification *H* contains a requirement that a software system halt, (D) cannot be satisfied for all cases of interest. The Halting Problem assumes that a computer is a Universal Turing Machine. That assumption excludes consideration of systems capable of implementing hypercomputation, which some authors have argued could or should be considered as a "computer".[6]

Modern general-purpose computing languages are *Turing complete*, i.e., programs written in them are equivalent to sets of instructions that can execute on a Universal Turing Machine. Such languages are ultimately defined in such a way that they can fully describe anything a Universal Turing Machine can do.

[5] For a detailed survey of the latter kind, see Emerson 2008 and Clark, Bloem, Veith, and Henzinger 2018

[6] Copeland et al 2016 make the case that hypercomputation should be taken seriously as a candidate for what is meant by "computation", given hypercomputation's compatibility with the Church-Turing thesis. We will not defend our choice to exclude hypercomputation in this paper, however see Davis (2006) for reasons to be skeptical.

The Halting Problem is a well-known limiting result that stands as one of the intellectual landmarks of computer science. However, from an engineering perspective, one can imagine granting that the Halting Problem is an insurmountable obstacle to verification while simultaneously regarding it as a special case that can be ignored in practice. One might, for example, focus on the practical task of developing methods for determining whether a system satisfies a specification in the following way: One could simply stipulate that one is excluding verification tasks that involve the kinds of self-referential or meta-level features that characterize the Halting Problem. Restrictions of this kind are implicitly what happens in engineering practice. Naively, it might seem that once provably unachievable specifications have been ruled out, the verification problem is a stratightforward *testing* problem. It turns out however, that verification-as-testing has intractable difficulties, albeit of a very different kind than the Halting Problem as we shall explain in following section.[7]

2.2 Verification-as-testing

In software engineering, most efforts to address the verification problem and minimize error involve testing. Testing involves significant challenges.[8] To see why, let S be a sequence of instructions written in some computer language L. The abstract executable structure of S can be represented as a control-flow-graph[9] (Nielson, Nielson, and Hankin 1999; Baier and Katoen 2008). We define a *path* in a software system to be a path (Diestel 1997, p. 6) in such a graph. We define the *path complexity* of S to be the number of possible paths in that control-flow graph. Path complexity, thus defined, captures the space of possible ways that the software system could run to completion.[10,11] The number of paths in a program increases at least exponentially with the number of conditional statements in S.[12] Consider, for example, a 1000-line (instruction) software system that has a binary conditional statement every 10 lines on average. The number of paths through such a program, and hence its path complexity, is $2^{1000}/10 = \sim 10^{30}$. In general, the path complexity of a program of M lines that has a binary decision branch on average every N lines is $2^{M/N}$, where $M > N$. We call this exponential scaling of the number of paths in S with the number of control (for our

[7] This problem has been known, at least informally, since the earliest days of software testing.
[8] Among philosophers, Jim Fetzer was the first to point out that software verification characterized as a testing problem poses challenges (Fetzer 1988) that are intractable in practice.
[9] A control-flow statement in S is a statement that can, based on a condition that may not always obtain, change the order of execution of the statement in S.
[10] Note that this definition requires that S can run to "completion". Some software systems, such as operating systems, by design "run forever", and thus have no "completion".
[11] This definition of path complexity is different from *McCabe complexity*, which is a count of the number of *independent* paths in S (McCabe 1976).
[12] A conditional statement is a statement of the form "If X, do Y". A Turing complete language, in addition to implementing conditional statements, must also implement loops. For the purpose of this paper, we can restrict the analysis to a program whose control statements are "if-then" statements only. Why? To show that verification-as-testing fails to satisfy (D1), it is sufficient to show that even if S contained only if-then control constructs, verification-as-testing would fail to satisfy (D1). (Accommodating loop control constructs in S only increases complexity.)

purposes, conditional) statements in S the *path-complexity catastrophe*; in control-flow-graph terms, it is equivalent, for some software systems, to what is sometimes called the "state-explosion" problem (Valmari 1998). [13]

A 1000-line program is extremely short by contemporary standards. For example, it is not uncommon for large scientific simulators to contain ~10^5 lines of code (Horner 2003). The UNIX/Linux operating systems each contain at least ~10^6 lines of code. Facebook's software system reportedly contains ~61 million lines of code!

Why is high path complexity significant for verification-as-testing? One way to begin to answer this question is to consider the degree of confidence we should assign to the results of a system. A natural way of thinking about the appropriate degree of confidence we would give to these systems is in terms of their reliability. We can be more confident in the behavior of a system if we can judge it to be reliable.

How can we determine the reliability of software systems through testing? In an empirical scientific domain that uses no software (e.g., measuring the temperature of a material object, using only a thermometer), there is a relatively straightforward approach that an agent could take. Typically, the distribution of errors (in the case of the thermometer, the distribution of errors presumed to be contained in a set of measurements) in such a domain can be characterized by conventional statistical inference theory (CSIT) (Hogg, McKean, and Craig 2005, Chaps. 5-12). CSIT requires us to randomly draw (Hogg, McKean, and Craig 2005, Df. 5.1.1) a sample from the population of interest, then apply statistical tests to the sample to assess the probability that a specific hypothesis (H) about the population holds. Often the sample size required to test a hypothesis of interest in such a case is small -- on the order of 100.

In the case of a domain that uses non-trivial software in an essential way, we cannot, in all cases of interest, be assured that CSIT can be used to characterize the distribution of error. It has been shown (Symons and Horner 2017) that it is not possible to ensure, in all cases of interest, that the errors in a software system are characterizable by random variables (for a definition of "random variable", see Chung 2001, Chapter 3). CSIT requires (here, error) distributions to be defined in terms of random variables, so we cannot, for all software systems of interest, be assured that CSIT is applicable. It is always possible, furthermore, to extend (perhaps unintentionally) any software system whose error distribution is characterizable by a distribution of random variables to a software system the distribution of whose errors is not characterizable as a distribution of random variables (Symons and Horner 2017).

If we assume that verification is testing, and we cannot apply CSIT to testing, we must test all paths in a software system in order to characterize its error distribution. But that approach is intractable. To get at least an informal sense of this problem, again consider the 1000-line program mentioned above. Suppose that we could test one path per second and that the program contained on average, a binary branch per 10

[13] For a complete discussion of the path complexity catastrophe see Symons and Horner (2014).

lines. Exhaustively testing all paths in such a program would take more ~10^{13} lifetimes of the Universe to test all paths in the code.

It might be objected to the above that the "path-complexity catastrophe" is largely determined by the relatively slow speed of human action or comprehension. One might imagine, such an objection might go, an entirely automated testing regime in which no human involvement is required.

Although it is difficult to discern what a test regimen completely devoid of human involvement could be (Turing 1950), let's entertain the notion that there might be such a scheme. In that case, we note that the test regimen must nevertheless involve coordinated collecting of test results at a given point in space (Cover and Thomas 2006; Hennessy and Patterson 2007; Reichenbach 1957). That communication is speed-of-light limited (Reichenbach 1957). Let's suppose, for example, that the average distance such communication must traverse is 1 meter, permitting us to replace the time to execute a test case in the analysis above to ($1/(3 \times 10^8$ m/sec) ~) 3×10^{-9} sec. In this regimen, therefore, it would take "only" ($10^{13} \times 10^{-9}$ ~) 10^4 lifetimes of the universe to test all paths in a non-trivial 1000-line software system. Thus, even if the time to execute a test case were limited only by speed-of-light communication, the path-complexity catastrophe would persist on speed-of-light scale.

To address this concern, it has been suggested that parallelizing tests (i.e., executing those tests at the same time) on a sufficiently large computer could, in theory, make the path-complexity catastrophe go away. However, even if this were a solution to the problem for some software systems, it is significantly limited for those software systems whose testing is state-history-dependent (e.g., large climate simulators), because the software sequences to be tested are not decomposable to anything smaller than a sequence that produces an entire system trajectory.

For at least some software systems, even maximally parallelizable testing cannot make the path-complexity catastrophe go away. Here's why. The minimum time, t_{coord}, to coordinate at a given spatial location, P, the reports of tests executed in parallel at M disjoint spatial locations x1, x2, …, xM, is ~ Md/c, where

- $d > 0$ is the mean of normally distributed one-way distances between P and the xi, $i = 1, 2, \ldots, M$
- $xi \cap P = \emptyset$ for each i
- $\emptyset$ is the null set (null region)
- c is the speed of light

(For a more detailed discussion, see Amdahl 1967). Note that for any d, as $M \rightarrow \infty$, $t_{coord} \rightarrow \infty$. Thus, no matter what value *d* has, there is a positive lower bound to t_{coord}, and a corresponding upper bound on the

path-complexity of some software system that can be tested in any finite time. This argument generalizes to the case in which there is merely some finite time – not necessarily determined by light time-of-flight -- required to coordinate results among M separate tests, provided c is finite.

In summary, verification-as-testing cannot satisfy at least (D1) and (D5) for all cases of interest. Given that testing will not succeed, we must explore alternative strategies.

2.3 Formal methods of verification

What can be done to overcome the limits of verification-as-testing? It would seem that formal methods, roughly analogous to formal methods in logic (see for example Chang and Keisler 2012; Gries 1981) could provide purchase on the verification problem. Such approaches provide at least mathematically well-defined frameworks within which a variety of desirable meta-level properties (e.g., consistency and completeness) can be characterized by finite procedures. Is there some equivalent strategy for proving the correctness of a software system? There have been some impressive results in this program.

In this section, we sketch some examples of how the program of formal verification has been pursued. This overview closely follows Emerson's 2008 retrospective and is not intended to be exhaustive or original. Our purpose here is only to introduce and illustrate some of the main achievements of the formal verification approach (for a fuller recent survey of these topics, see Clark, Bloem, Veith, and Henzinger 2018). We will argue that formal methods of verification variously meet at least some of (D1) – (D5) for at least some software systems. But ultimately, as we argue in Section 3.0, no verification method can satisfy (D5) for all cases of interest, and that result limits the detail that is proportionate to include in the more-or-less historical overview that follows (i.e., in Sections 2.3.1 and 2.3.2).

2.3.1 The "axiomatic" approach to formal verification

One way to formalize the verification problem is to cast it as a question about whether a given software system S is equivalent to a theorem in a theory, where that theory is formulated as a set of axioms that captures a specification H. In this paradigm, one manually constructs proofs of correctness for (deterministic) programs that start with an input and terminate with an output. To do this, a computer program is first translated into a set of sentences in the formal language L in which H is expressed. We then attempt to show that the resulting set of sentences has a proof in an axiom system (that determines H). This approach to formal verification is called "axiomatic" verification. Floyd 1967, for example, provided some basic principles for this approach by proving "partial correctness" in such a framework, as well as articulating

termination and total correctness forms of liveness properties (Emerson 2008, p. 29).[14] Extending this idiom, Hoare 1969 provided an axiomatic basis for verification of partial correctness using axioms and inference rules in a formal deductive system (Emerson 2008, p. 29).

The Floyd-Hoare framework provided many useful insights. The approach facilitated the investigation of important meta-theoretic properties such as soundness and (relative) completeness, as well as compositionality. This framework, however, turned out to have limited utility in practice for several reasons (Emerson 2008, p. 29). First, the approach scaled exponentially in the number of terms in the theorems to be proven, thus failing to satisfy (D1). Second, the approach often required discovering, in non-mechanical ("inspired") ways, proof strategies specific to the problem of interest, and failure to discover a proof did not imply that there was no proof possible, thus failing to satisfy (D2). Third, there was no known way to automate the approach, thus failing to satisfy (D3). Fourth, the framework could not express the temporal aspects of concurrent programs, thus failing to satisfy (D4).

Pnueli 1977 extended the Floyd-Hoare approach to capture least part of concurrency. Pnuelli proposed, as a working hypothesis, that temporal logic could be used for reasoning about concurrent programs.[15] To capture concurrency, he defined a temporal logic-based system that included as basic temporal operators F (sometimes), G (always), X (next-time), and U (until) (Emerson 2008, p. 30). Besides these basic temporal operators applied to propositional arguments, Pneuli's system permitted nested and boolean combinations of subformulae (Emerson 2008, p. 31).

Like the Floyd-Hoare framework, Pneuli 1977 took an axiomatic approach to verification. Like the Floyd/Hoare approach, Pneuli's approach cannot satisfy (D1) and (D2). Nevertheless, Pneuli's incorporation of temporal logic into the formal description of software systems, it turned out, provided powerful resources for addressing (D4).

Temporal logic comes in two broad flavors (Emerson 2008, pp. 31-32, See also Venema 2001):

(1) Linear Time Logic (LTL (Pneuli 1977))

> In LTL, an assertion h is interpreted by default to apply to a single path. (Here, "path" means a sequence of instructions that represents one way that a system could run.) When interpreted over a program there is an implicit universal quantification over all paths of the program.

(2) Branching Time Logic (BTL)

[14] A liveness property asserts that program execution eventually reaches some desirable state (Owicki and Lamport 1982).

[15] Concurrent systems are often reactive systems (i.e., they execute in response to a stimulus (e.g., from a sensor) outside those programs). Reactive systems are often nondeterministic, so their non-repeatable behavior is not amenable to testing. Their semantics can be given as infinite sequences of computation states.

> An assertion h of a *branching time logic* is interpreted over computation trees. A branching time logic has a universal future-time quantifier A (for all futures) and an existential future-time quantifier E (for some future) paths. These quantifiers allow us to distinguish between AFP (along all futures, P eventually holds and is thus inevitable)) and EFP (along some future, P eventually holds and is thus possible).

One widely used branching time logic is known as Computation Tree Logic (CTL). Its basic temporal quantifiers are A (for all futures) or E (for some future) followed by one of F (sometime), G (always), X (next-time), and U (until); compound formulae are built up from nestings and propositional combinations of CTL subformulae (Emerson 2008, p. 32, See also Huth and Ryan 2004).

CTL and LTL do not have the same expressive power (Emerson 2008, p. 32). There is an ongoing debate as to whether linear time logic or branching time logic is better for formal verification objectives (Emerson 2008, p. 32).

One prominent logical framework familiar to logicians and that can capture CTL is the mu-calculus (Kozen 1983). The mu-calculus provides operators for defining correctness properties using recursive definitions and least fixpoint and greatest fixpoint operators. Least fixpoints correspond to well-founded or terminating recursion, and are used to capture liveness or progress properties asserting that something does happen. Greatest fixpoints permit infinite recursion. They can be used to capture safety or invariance properties. The mu-calculus is very expressive and flexible. It is still in wide use in formal verification methods (Emerson 2008, Section 4).

2.3.2 Model-checking

Casting the verification problem in a temporal logic does not, by itself, overcome all the problems faced by axiomatic approaches. Framing the problem in terms of temporal logic does nothing to address (D1) – (D3), and there is no guarantee that it would satisfy (D5). To help address these issues, Clarke and Emerson 1981 observed that if in contrast to the axiomatic approaches to proof of correctness, we *derived* a software system S directly from H we might be able to overcome at least some of the problems of the axiomatic approach. This proposal – formally deriving software from a specification – is often called the "synthesis" approach to formal verification of software.

The synthesis approach, as such, does not guarantee that (D1), (D2) and (D5) are satisfied. One way to help to meet (D1) and (D5) is to require that the description of S define a *finite* state graph M. M can then searched, via pattern specifications (which can be of arbitrary complexity), to determine whether M satisfies H.

Clarke and Emerson 1981 proposed, in particular, that we exploit the "small model property"[16] possessed by certain decidable temporal logics. Exploiting the small model property of these decidable temporal logics has at least two further virtues: The method is sound: if the input specification is satisfiable, the method generates a *finite* global state graph that is a model of the specification, from which individual processes of S can be derived. The method is also complete: If the specification is unsatisfiable, it would be possible to determine that is unsatisfiable: given any finite model M and CTL specification H one can algorithmically check that M is a genuine model of H by evaluating (verifying) the basic temporal modalities over M based on the fixpoint properties. Composite temporal formulae comprised of nested subformulae and boolean combinations of subformulae of CTL could be verified by recursive descent. These features: CTL, fixpoint properties, and recursion – became the foundation of what is now called "model checking" (Emerson 2008).

How well does model-checking work? In practice, model-checking is typically implemented in a *model checker*. A model checker is a software tool that helps to assess whether a software system S is a model of a specification H. Model checkers that are formulated in CTL can be quite useful in practice, especially when applied to finite-state concurrent systems. Moreover, CTL has the flexibility and expressiveness to capture many important correctness properties. In addition, a CTL model checking algorithm has reasonable efficiency: it is polynomial in the specification size (i.e., it satisfies (D1)).

The fundamental accomplishment of model checking has been the enabling of broad scale formal verification. Today many industrial-application systems have been verified using model checking. Model checkers have verified protocols with millions of states and hardware circuits with at least 10^{50} states (Clark, Bloem, Veith, and Henzinger 2018).

For example, model-checking has been used to verify (Clark, Bloem, Veith, and Henzinger 2018)

- a cache coherence protocol
- the bus arbiter for the PowerScale multiprocessor architecture
- a high-level datalink controller
- a control protocol used in Philips stereo components
- an active structural control system to make building more resistant to earthquakes[17]

For at least some programs, model-checking evidently satisfies (D1)-(D4). In addition, model checking supports both verification and refutation of correctness properties. Since most programs do contain

[16] A system K has the small model property if and only if any satisfiable formula in K has a "small" finite model, i.e., a model whose size is a polynomial function of the formula size.

[17] The model checker used in this case found errors in the original design of the system. Some of these errors would have made buildings *less* resistant to earthquakes.

errors, an important strength of model checkers is that they can readily provide a counter-example for at least some classes of errors.

Despite its power, neither model checking nor any other verification method can satisfy (D5); the requirement that the method complete its task. in all cases of interest, as we now proceed to argue.

3.0 Can any method solve the verification problem for all cases of interest?

Model-checking satisfies (D1) - (D5) for at least *some* software systems. We will now argue that any software system that (a) is written in a Turing complete language and that (b) must implement at least Robinson arithmetic[18] (Mostowski, Robinson, and Tarski 1953)[19] has, as a consequence of the Löwenheim-Skolem theorem (Löwenheim 1915; Skolem 1920), an infinite number of non-isomorphic models and thus cannot be verified in a finite time,[20] i.e., cannot satisfy (D5).

To show this, we first posit that in any verification regimen, the following must hold (where S is a software system and H is a specification):

(A) *S satisfies a specification H up to model-identity* if each model (in the sense of Chang and Keisler 2012, p. 32) of H is homomorphic to some model of S.

Corresponding to (A), we posit that verification must satisfy:[21]

(Q) *A verification V verifies that S satisfies H up to model-identity* if V verifies that each model (in the sense of Chang and Keisler 2012, p. 32) of H is homomorphic to some model of S.

We can motivate (A) and (Q) as follows.

First, we note that it is more than difficult to imagine that there could be a sense of "S satisfies H" in which at least one model of H is not homomorphic to some model of S.

It might, however, be objected that (A) and (Q), which require that each model of H is homomorphic to some model of S is too strong. Why, such an objection might go, couldn't we just show that

[18] Robinson arithmetic is "ordinary" (Peano) arithmetic without the Peano induction axiom.
[19] Virtually all business and scientific software must implement arithmetic.
[20] Note that a theory of arithmetic (or anything else) that is not finitely axiomatizable cannot be implemented on a finite Universal Turing Machine. No second-, or higher-, order theory of arithmetic, for example, can be implemented on a Universal Turing Machine. (See for example Chang and Keisler 2012, Chapter 1.)
[21] Although beyond the scope of this paper, it's worth noting that criteria (A) and (Q) rest on a theory of verification that does not appear to be limited to software regimes as such, and thus might help to characterize verification in ordinary empirical science.

(O) One model B of H is homomorphic to some model of S?

On the surface, (O) seems sensible; in fact, (O) reflects the way "verification" is sometimes, if not often, performed in practice (Black, Veenendaal, and Graham 2012). There is, however, a fundamental limitation to this approach.

To see this, let H and S be described as above. Suppose there were a model A of H that was not homomorphic to some model of S. This would imply that there is a sense in which H expresses a requirement that is not addressed by S.

Consider now the following:

1. By (Q), in order to verify that S satisfies H up to model-identity, we must verify that every model of H is homomorphic to some model of S.
2. Let H be the requirement to implement a finite first-order axiomatization of at least Robinson arithmetic on a Universal Turing Machine. This requirement is implied by any requirement to implement ordinary arithmetic on a Universal Turing Machine. Note that the specifications of virtually all business and scientific software imply this requirement.
3. The Löwenheim-Skolem theorem (Löwenheim 1915; Skolem 1920) implies that there are an infinite number of non-isomorphic models of any finite first-order axiomatization of Robinson arithmetic. Therefore, by (1), (2), and Löwenheim-Skolem, verifying that S satisfies H up to model-identity requires verifying that that each of the infinite number of non-isomorphic models of H is homomorphic to some model of S.
4. Now note that S could be anything – S is merely an *attempt*, and *possibly* a *flawed* one, to satisfy H. Because we cannot a priori presume any properties of S, in general, given that a model B satisfies H, we cannot, for arbitrary B, infer that B is homomorphic to a model of S except by verifying, in a distinct action, that B is homomorphic to a model of S. Such a verification activity takes some non-zero time.
5. For all models of H, let $t_{min} > 0$ be the shortest time required to show a model of H is homomorphic to a model of S.
6. By (3), (4), and (5), for at least some software systems, we must perform an infinite number of distinct verification actions to verify that S satisfies H up to model-identity. These actions will collectively take (t_{min} times infinity =) infinite time. Thus, we cannot verify in a finite time (D5) that an arbitrary S satisfies an arbitrary H up to model-identity.

We conclude, therefore, that, even were the Halting Problem solvable, no verification method can satisfy (D5) for all cases of interest.

4. Conclusion

The most important implication of the results of Section 3.0 concerns the scope of software specifications, and more specifically, whether a specification that constrains "arithmetic" in a way could, if possible, make the specification satisfiable. The Halting Problem notwithstanding, all approaches to software verification considered above are tractable for small software systems (nominally, ~100 lines of code). For example, there is no need for an ordinary thermostat system in a home to perform more than a very small set of operations. One could even imagine such systems operating with finite look-up tables for all the operations that they would need in order to satisfy a specification. Even in cases involving arithmetical operations, one could test a system that is required to perform that arithmetic in a small finite domain. It is highly unlikely, however, that restricting specifications for software along these lines will be acceptable in the scientific context or even in most internet-enabled products in a consumer context.

Understanding the trade-offs involved as we weigh the importance of verifying software correctness with the benefits of large-scale software are not matters that we ought to leave solely to engineers and corporate leaders. Elsewhere, it has been argued that philosophers need to carefully reflect on the nature of science when the reliability of our most important instruments is impossible to determine with confidence (Symons and Horner 2014, Symons and Alvarado 2016). Perhaps more importantly, philosophers also need to reflect on the ethical implications of creating ever larger, more connected, systems with more layers of interdependence and vulnerability. Responsibility for failure is increasingly difficult to assign. The cost of detecting critical error can be impractically high and the consequences of failure to detect critical error can be lethal. In the Toyota unintended acceleration (UA) case (Koopman 2014), for example, we now know that verifying the as-built software is intractably difficult. At the same time, software systems like those in the UA case provide us with highly efficient and generally excellent cars. Is the loss of a handful of lives an acceptable price to pay for cars with more desirable features? How should one approach the risks associated with purchasing such a vehicle? What duty does the manufacturer have to explain the presence of error in its product? These are not questions that computer scientists or engineers are equipped to answer.

How do we balance our desire to be connected electronically to our toasters and toothbrushes, or to fill our homes with smart speakers/microphones/cameras with unverifiable and potentially serious

vulnerabilities that these systems introduce in our lives and social systems? The first step in beginning to answer these questions involves understanding the limits of our ability to minimize error in these systems.

5.Acknowledgments

TO BE ADDED

References

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings* (30): 483–485. doi:10.1145/1465482.1465560.

Baier, C., & Katoen, J. P. (2008). *Principles of model checking.* MIT Press.

Black R., Veenendaal E., & Graham G. (2012). *Foundations of Software Testing ISTQB Certification.* Cengage Learning EMEA.

Blum EK, Paul M, and Takasu S (eds). (1979). Mathematical Studies of Information Processing: Proceedings of the International Conference, Kyoto, Japan, August 23-26, 1978 Lecture Notes in Computer Science 75. Springer.

Boolos, G., Burgess, J., & Jeffrey, R. (2007). *Computability and Logic* (5th ed.). Cambridge UK: Cambridge University Press.

Boschetti, F., Fulton, E. A., Bradbury, R., & Symons, J. (2012). What is a model, why people don't trust them, and why they should. *Negotiating our future: Living scenarios for Australia to*, *2050*, 107-119.

Chang, C., & Keisler, J. (1990). *Model Theory.* North-Holland.

Chung, K. L. (2001). *A Course in Probability Theory.* 3rd Edition. New York: Academic Press.

Clarke, E. M. and Emerson, E. A. (1981). Synthesis of synchronization skeletons for branching time temporal logic. In Logic of Programs. Lecture Notes in Computer Science 131. Springer, 52–71.

Clarke, E. M., Bloem, R., Veith, H., Henzinger, T. A. (eds). (2018). *Handbook of Model Checking.* Springer.

Copeland, J., Dresner, E., Proudfoot, D., & Shagrir, O. (2016). Time to reinspect the foundations?. *Communications of the ACM*, *59*(11), 34-38.

Cover, T. M., and Thomas, J. A. (2006). *Elements of Information Theory*. Second Edition. Wiley.

Davis, M. (2004). The myth of hypercomputation. In *Alan Turing: Life and legacy of a great thinker* (pp. 195-211). Springer, Berlin, Heidelberg.

Diestel, R. (1997) *Graph Theory*. New York: Springer-Verlag,

Emerson, E. A. (2008). The beginning of model checking: A personal perspective. In Grumberg, O., & Veith, H., eds (2008). *25 Years of Model Checking - History, Achievements, Perspectives*. Vol. 5000 of *Lecture Notes in Computer Science*. Springer.

Floridi, Luciano, Nir Fresco, and Giuseppe Primiero. "On malfunctioning software." *Synthese* 192.4 (2015): 1199-1220.

Floyd, R. W. (1967). Assigning meanings to programs. In Schwartz, J. T. (ed.). *Proceedings of a Symposium in Applied Mathematics. Mathematical Aspects of Computer Science,* Volume 19, pp. 19-32.

Gries, D. (1981) *The Science of Programming*. New York: Springer-Verlag,

Fetzer, J. H. (1988). Program verification: the very idea. *Communications of the ACM*, *31*(9), 1048-1063.

Hennessy, J., & Patterson, D. (2007). *Computer Architecture: A Quantitative Approach*. Fourth Edition. New York: Elsevier.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM* 12, 576-580.

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

Horner, J. K. (2003). The development programmatics of large scientific codes. *Proceedings of the 2003 International Conference on Software Engineering Research and Practice,* 224-227. Athens, Georgia: CSREA Press.

Michael Huth; Mark Ryan (2004). *Logic in Computer Science*. Cambridge: Cambridge University Press

IEEE. (2000). *IEEE-STD-1471-2000. Recommended practice for architectural description of software-intensive systems. http://standards. IEEE. org.*

Koopman, P. (2014). A Case Study of Toyota Unintended Acceleration and Software Safety. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf. Accessed 17 April 2018.

Kozen D. (1983). Results on the propositional μ-calculus. *Theoretical Computer Science* 27, 333–354.

Löwenheim, L. (1915). Über Möglichkeiten im Relativkalkül. *Mathematische Annalen* 76 (4): 447–470, doi:10.1007/BF01458217. A translation to English can be found in Löwenheim, Leopold (1977), "On possibilities in the calculus of relatives", *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931* (3rd ed.), Cambridge, Massachusetts: Harvard University Press, pp. 228–251.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering* 2, 308–320. Also available at http://www.literateprogramming.com/mccabe.pdf.

Mostowski A., Robinson, R M., and Tarski A. (1953). Undecidability and essential undecidability in arithmetic. In Tarski A., Mostowski A., and Robinson RM. *Undecidable Theories.* Dover reprint.

Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of Program Analysis.* Springer.

Owicki S and Lamport L. (1982). Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems* 4, 155-495.

Pneuli, A. (1977). The temporal logic of programs. *Foundations of Computer Science*, 46-57.

Reichenbach, H. (1957). *The Philosophy of Space and Time.* Translated by Maria Reichenbach. Dover edition.

Skolem, T. (1920), Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen. *Videnskapsselskapet Skrifter, I. Matematisk-naturvidenskabelig Klasse* 6: 1–36. An English translation can be found in Skolem, T. (1977), "Logico-combinatorical investigations in the satisfiability or provabilitiy of mathematical propositions: A simplified proof of a theorem by L. Löwenheim and generalizations of the theorem", *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931* (3rd ed.), Cambridge, Massachusetts: Harvard University Press, pp. 252–263.

Symons, J.F., & Alvarado, R. (2016). Can we trust Big Data? Applying philosophy of science to software. *Big Data & Society*, *3*(2), 2053951716664747.

Symons J. F., & Horner, J. K. (2014). Software Intensive Science. *Philosophy and Technology 27*(3), 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, T. (ed.) *Philosophy and Computing: Essays in Epistemology, Philosophy of Mind, Logic, and Ethics,* Springer. pp. 85-97

Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 42. pp. 230–65.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind* LIX, 433–460.;

Valmari, A. (1998). The state explosion problem. *Lectures on Petri Nets I: Basic models. Lectures in Computer Science* 1491, 429-528.

Venema, Y., (2001) "Temporal Logic," in Goble, Lou, ed., *The Blackwell Guide to Philosophical Logic*. Blackwell. pp. 259-281