Software Error as a Limit to Inquiry for Finite Agents: Challenges for the Post-human Scientist

John F. Symons and Jack K. Horner

Abstract

Finite agents must build rule-governed processes of some kind in order to extend the reach of inquiry beyond their limitations in a non-arbitrary manner. The clearest and most pervasive example of a rule-governed process that can be deployed in inquiry is a piece of scientific software. In general, the error distribution of all but the smallest or most trivial software systems cannot be characterized using conventional statistical inference theory, even if those systems are not subject to the halting problem. In this paper we examine the implications of this fact for the conditions governing inquiry generally. Scientific inquiry involves trade-offs. We show how increasing use of software (or any other rule-governed procedure for that matter) leads to a decreased ability to control for error in inquiry. We regard this as a fundamental constraint for any finite agent.

1.0 Introduction

The increasingly central role of computing technologies has changed scientific inquiry in significant ways. While the practical import of this transformation is undeniable, its implications for the philosophical understanding of scientific inquiry have been less clear. In this paper we focus on a traditional topic in the philosophy of science, namely the idea of progress in inquiry. We examine how the use of software affects progress in inquiry. We will explain why the error distribution[1] of all but the smallest or most trivial software systems cannot be characterized using

[1] An anonymous reviewer of this paper noted that software testers typically distinguish among human errors, software faults (erroneous states of the software), and software failures (when actual output does not match expected output). This paper is concerned with error distributions driven by software faults.

conventional statistical inference theory, and why this matters for our understanding of scientific inquiry. By carefully thinking about the relationship between software and statistical error correction techniques we can shed light on how inquiry conducted by finite agents is constrained. We argue that this constraint is a feature of inquiry for finite agents in general and that "in the limit" thinking about scientific progress ought to be recast in terms of a sharper focus on formally characterizing constraints and trade-offs for inquiring agents generally.

In the Limit of Inquiry

Charles Sanders Peirce famously argued that "Inquiry properly carried on will reach some definite and fixed result or approximate indefinitely toward that limit" (Peirce 1931, vol.1, 458) His faith in the convergence of inquiry towards some fixed result given sufficient time and effort was such that it formed the basis for his characterization of truth. An assertion is true, he argued, if inquiry would converge to that assertion in the limits of inquiry (assuming that inquiry is "properly carried on"). While there are good reasons to reject Peirce's characterization of truth as the output of inquiry in the limit, his "definition" of truth highlights how inquiry was central to some of the most important aspects of his philosophical work.

Peirce regarded his intuition concerning the progress of inquiry to be completely unassailable. "It is unphilosophical to suppose that, with regard to any given question, (which has any clear meaning), investigation would not bring forth a solution of it, if it were carried far enough." (Peirce 1957, 55) More recently, at least some philosophers of science have continued to embrace the idea: "There is a fundamental assumption or thesis in philosophy which says that scientific knowledge may be characterized by convergence to a correct hypothesis in the limit of empirical scientific inquiry." (Hendricks 2001, 1; Domski and Dickson 2010)

Here we argue that this assumption should be re-examined..

In this paper, we largely argue for a negative result. However, abandoning arguments that take as a premise claims about conditions in the limits of inquiry has some salutary effects for philosophy of science. As mentioned above, the most important consequence is a renewed focus on objective constraints and trade-offs for inquirers.

Thinking about inquiry in the limit requires us to assume that finite agents are capable of inquiring in ways that overcome their limitations. Past a certain point, finite agents need help to make progress. It seems reasonable to assume that the first step in going beyond our limitations is to create methods or systems that follow instructions. Such rule-governed systems are likely to be a crucial part of progress beyond our native epistemic constraints. Use of software, running on digital computers, is the most pervasive contemporary example of a tool that allows us to create systems that extend inquiry beyond human limitations. There are, of course, other ways in which we can create methods for extending inquiry. The construction of protocols for collective action, division of labor, and other aspects of so-called big science, require something like lists of rules or systems of instructions. Rules for what is to count as the *proper* or *competent* conduct of inquiry are already implicit in Peirce's account of inquiry in the limit. In this paper, we focus on software insofar as it is more amenable to formal reflection than methods or protocols in big science. While we focus on software, we regard the general result as applicable to other rule-governed systems and argue for this result elsewhere.

Software and error

Symons and Horner have argued (Symons and Horner 2014, Horner and Symons 2014) that the error distribution of all but the smallest or most trivial software systems cannot be characterized even using conventional statistical inference theory (CSIT). There are at least two senses in which the characterization of the error distribution might not be attainable: (a) because of distinctively human limitations (e.g., the speed at which humans can work is limited by human physiology, the human species might not endure forever, etc.), or (b) because *no* finite agent who is constrained only by mathematics[2] and the temporal finiteness of the universe in which that agent acts would be able to attain the goal.

Symons and Horner 2014 emphasized (a); in this paper, we argue that (b), which arguably implies (a), also holds.

For the present purposes, we define a "post-human agent" (PHA) to be a finite agent whose "representations" are constrained only by

[2] By "mathematics", we mean "mathematics as we know it", i.e., anything definable in set theory.

- set theory (Bernays 1968)
- model theory (Chang and Keisler 1990)
- conventional statistical inference theory (CSIT; Hogg, McKean, and Craig 2005)
- the speed-of-light communication/coordination limit (Reichenbach 1957; Susskind 2014; Deutsch 1997, Chap. 6; Lloyd 2000)
- the temporal finiteness of the universe in which the agent acts

By "Y is a representation of X" we mean "Y is a *function* of X (in the sense of set theory)".

Note that a PHA, thus defined, has no distinctively human limitations (other than, possibly, some mathematics, *if* that mathematics is distinctively human).

There is, of course, a celebrated limitation to what a PHA could know about the error distribution of a software system. In particular, the *halting problem* (Turing 1937, Turing 1938) states that it is not *in general* possible to decide whether some software systems running on a canonical computer (i.e., a Turing machine (Boolos, Burgess, and Jeffrey 2007)) will halt. If we require that a software system S halt, else be in error, then a PHA cannot in general determine whether S will halt and thus cannot in general determine the error distribution of S. It can be proven that some software systems will halt; it can also be proven that some software systems will not halt.

We will argue that, in general (from here on, we will use the phrase "in general" to mean "for all cases of interest"), a PHA cannot determine the error distribution of a software system, even if that software system is *not* subject to the halting problem.

2.0 Outline of the argument

In outline, our argument is:

1. To the extent that software plays any essential role in a domain of inquiry, a PHA cannot in general determine that even conventional statistical inference theory (CSIT; Hogg, McKean, and Craig 2005) can be used to characterize the distribution of error in that software.

2. Without CSIT to help characterize the distribution of errors in a software domain, there is in general no known effective method (Hunter 1971) for characterizing the error distribution of that software -- except by testing that software, and in general, testing all *paths* (defined late) in that software.
3. Defining and validating test cases for, executing those test cases, and analyzing the results of the testing of all paths in software systems is intractable. We call this the "path-complexity catastrophe".
4. Even "model-based" approaches (Baier and Katoen 2008) to proof of software correctness cannot, if powerful enough to capture the full flow-control structure of S, overcome the path-complexity catastrophe.
5. Therefore, by (1) – (4), a PHA cannot characterize the distribution of errors in software systems, even if those systems are *not* subject to the halting problem. This is a fundamental limit imposed by software on (scientific) inquiry.

We now argue for (1) – (4) in turn.

3.0 The argument in detail

In this section, we argue for each of the premises listed in Section 2.0 in detail.

Premise 1. *A PHA cannot, in general, determine whether a software system S satisfies the conditions for the applicability of CSIT.*

Computer scientists call a system *software intensive* if "its software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole." (IEEE 2000) By this definition, much of contemporary science involves software intensive systems.

In an application of CSIT, a finite sample is randomly drawn from what is typically presumed to be a population P. Statistics (e.g., the sample average and standard deviation) are computed based on properties of the sample. Tests are then performed on the sample statistics.

Based on a statistical inference rule R that *presumes that the (error) distribution of P is the distribution of a random variable (r.v.)*[3], and is defined in terms of the results of those tests, we are allowed to draw a probabilistic conclusion about whether P has properties that can be expressed as a function of the sample statistics (e.g., that the population mean has the value of the sample average).

We note that an r.v. X is defined in terms of, among other things, a probability measure. Thus for X to be an r.v., among other things it must satisfy the conditions of having a probability measure. A probability measure is a real-valued (discrete or continuous) distribution (Chung 2001, Section 1.1) function (Chung 2001, p. 21). Hereafter we will assume the definition of an r.v. given in Chung 2001 (p. 34, reproduced in Footnote 3 of this paper.) That definition is what we mean when we say that if X is an r.v., then "X must have/be the distribution of a random variable".

In order for CSIT to be able to characterize the error distribution in an application domain for a PHA, the error must be representable as (a distribution of) a random variable (Chung 2001; Bevington and Robinson 2002). In a Turing complete language, we can always construct a statement sequence C' whose error distribution is *not* representable by a random variable. More precisely, the errors in is a software system S could fail to characterizable by an r.v. by failing to

---

[3] A random variable (r.v.) is defined as follows (Chung 2001, p. 34). Let

1. Ω be a space (Chung 2001, p. 16)
2. F be a Borel field (Chung 2001, p. 388) of subsets of Ω
3. P be a probability measure (Chung 2001, p. 21) on F
4. $<\Omega, F, P>$ be a probability space (Chung 2001, p. 23)
5. $R^1 = (-\infty, +\infty)$ be the finite real line
6. $R^* = [-\infty, +\infty]$ be the extended real line
7. B^1 be the Euclidean Borel field (Chung 2001, pp. 387-388) on R^1
8. B^* be extended Borel field (Chung 2001, p. 388)
9. A set in B^* be a set in B possibly including one or both of $-\infty$ or $+\infty$

A *real, extended-valued random variable* is a function X whose domain is a set Δ in F and whose range is contained in R^* such that for each B in B^*, we have

$$\{\omega : X(\omega) \in B\} \in \Delta \cap F$$

where $\Delta \cap F$ is the trace (Chung 2001, p. 23) of F on Δ.

In many cases of interest, $\Delta = \Omega$..

satisfy any of the > 10 conditions of the definition of an r.v. C' could be conjoined (perhaps unintentionally) to a software system S whose error distribution can be characterized by the distribution of a random variable, to create a software system S'. No agent (PHA or otherwise) can legally apply CSIT to S'. If S' is sufficiently large, a PHA, who is finite, cannot in general know whether the errors in S' are representable as a distribution of a random variable, because the PHA's finite sampling of S', a la CSIT, might fail to detect the features of S' whose error distribution is not characterizable by a random variable. Therefore, in general a PHA cannot know that CSIT is applicable to software systems.

We note that this limitation does *not* presume any specific type (e.g., normal, uniform, binomial, Poisson, etc.) of probability distribution, but holds for *all* distribution types, because all CSIT tests (see, for example, Definition 5.1.2 of Hogg, McKean, and Craig 2005) require that a distribution of an r.v. be in play.[4]

Against this view, it has sometimes been argued that we can apply CSIT without restriction to any system, so Premise 1 cannot be correct.

Although it seems compelling, the objection stands on a critical, implicit assumption: that the population to which CSIT is being applied is characterized by (the distribution) of a random variable. Every statistical inference rule requires this assumption, though in typical practice the assumption is often implicit. That assumption is a reasonable working hypothesis for many populations, and in the absence of evidence to the contrary, is plausible. Notice, however, that unless we know the distribution is that of a random variable, the assumption is, in the case of every population that has not been exhaustively surveyed, a posit, not something that we actually know to be the case about the population of interest. (If the population has been exhaustively surveyed, we have no need for CSIT.)

We can use CSIT to test, it might be countered, whether a population has a distribution of a random variable. And if that is so, we can at least identify cases in which we can't use CSIT.

Let's unpack this objection. There are two cases to consider. In the first case, suppose there were such a test and it showed that the population of interest cannot characterized by the distribution of random variable. Then CSIT, or at least all of CSIT with the exception of the test

[4] Thanks to the anonymous reviewer who recommended this clarification.

of whether the population has a distribution of a random variable, would be inapplicable. But in that case, Premise 1 obtains.

Now suppose that the test indicated we had grounds to believe that the population of interest, P, is characterized by the distribution of a random variable. In CSIT, we randomly draw a finite sample from P. Suppose that, unknown to us, P has a subpopulation, SP, that has a distribution of a random variable, and another subpopulation, SP', that does not have such a distribution. S', considered in the argument for Premise 1, above, is an example of this kind of "hybrid" population. Now nothing precludes a sampling from selecting only members of SP. (We hope that sampling doesn't produce this result, but nothing prohibits it from doing so. Such a sampling would not be a *random* sampling,[5] though that fact would be unknown to us.) If our sample happens to contain only members of SP, CSIT will tell us that P has, with some probability, population parameters with values defined in terms of the sample statistics. It's important to emphasize that in this case, applying CSIT, presumes, contrary to fact, that P has the distribution of a random variable.

Premise 2. *Without CSIT, a PHA has no known effective method for characterizing the error distribution of a software system S – except by testing all paths in S.*

By a "path" in a software system S, we mean any maximal chain of statements in S whose flow-control structure (Nielson, Nielson, and Hankin 1999) is determined by instantiations the "if X, do Y" (aka "binary branch") schema.[6] (A Turing complete language must be able to implement such a schema.) By the "path complexity" of S, we mean the number of paths in S. To date, the only known candidate effective methods for characterizing the error distribution in a software system are CSIT and (exhaustive) testing of the paths in that system. Therefore, without CSIT, exhaustive path testing is the only known effective method for characterizing that distribution.

---

[5] A *random sample* is well defined only if the sampling is from a population *characterized by a random variable*. Note that in order to draw a random sample, we must be able to determine, independently of the sampling process, that the (here, error-) population being sample has the distribution of a random variable. See, for example, Hogg, McKean, and Craig 2005, Df. 5.1.1.

[6] Some variant of this argument would carry no matter what flow-control constructs were used to define a "path". The "if…then" schema is sufficient for our purposes.

It has been suggested[7] that instead of defining complexity in terms of flow-control constructs, we might define complexity in terms of "testing all possible inputs" to a software system S. We note, however, that the phrase "testing all possible inputs" actually means "testing all possible input-sets that could make a difference in the executable paths of S". It is true that in some cases, the set of all possible inputs to S that would make a difference in what paths might be executable in S might be small. But for the purposes of this paper, our objective is to consider all possible software systems, and that scope includes the case of a software system that would require us to test as many possible inputs as there are paths in S. Thus, in the most general case, the "testing all possible inputs" approach devolves to testing all paths in S.

Premise 3. *In general, defining and validating test cases for, executing those test cases, and analyzing the results of the testing, of all paths in a software system S is intractable.*

For the sake of illustration, consider a 300-*line* (by "line" we mean "source line of code" in the sense of Boehm et al. 2000, pp. 77-81) software system S with an average of one binary branch per 10 lines.

In order to understand the practical import of Premise (3), it helps to first look at S from a *human* point of view. Having to test every path in software system that contains more than ~10^8 paths is, for practical reasons, intractable. To show this, let's compute the time it would take to test all paths in a 1000-line system. The estimated lifetime of the universe is ~10 billion years, or about 10^{17} seconds. So it would take ~$10^{30}/10^{17}$ = ~10^{13} lifetimes of the universe to test[8] each path through a typical 1000-line software system that contains one binary branch, on average, per ten lines. Exhaustive path testing for typical software systems is therefore intractable for humans. Even within the space of relevant behaviors for applications there will be

[7] Thanks to an anonymous reviewer for this suggestion.

[8] The path-test cases for *some* software could be executed in parallel (Hennessy and Patterson 2007, p. 68); in theory, given a large enough parallel machine, all path-tests in such a case could be executed in the same time it takes to execute one test case. But we must first define such cases, validate them, and then analyze the results of exercising those cases; all of these activities are limited by human speed. For a detailed discussion of why attempts to overcome this limitation cannot succeed, see Symons and Horner 2014. This result can be generalized up to speed-of-light communication constraints: see comments at the end of Section 2.0.

untested paths in those systems.[9] This path-based consideration, furthermore, determines a fundamental limit on what we can know about the error distribution in a software system.

It might be objected that this "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 imagine what a test regimen completely devoid of human involvement could be (Turing 1950; Symons, Boschetti, Fulton, and Bradbury 2012), let's entertain the notion that there might be such a scheme. In that case, we note that the test regimen must nevertheless involve synchronized 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 (Misner, Thorne, and Wheeler 1973; Reichenbach 1957). Let's suppose 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 $(1/(3 \times 10^8 \text{ m/sec}) \sim) 3 \times 10^{-9}$ sec. In this regimen, therefore, it would take "only" $(10^{13} \times 10^{-9} \sim) 10^4$ lifetimes of the universe to test all paths in a 1000-line software system. Thus, even if the time to execute a test case were limited only by speed-of-light communication, some variant path-complexity catastrophe would persist.

How does this example fare from a PHA point-of-view? A 300-line system is tiny by today's standards. Several software systems contain $\sim 10^6$ (e.g., Boeing 777 flight software) – $\sim 10^7$ lines (e.g., Microsoft Windows). A 10^6-line system (with a binary branch per 10 lines on average) has a path complexity of $\sim 10^{30,000}$. In comparison, there are $< 10^{100}$ fundamental particles in the known universe (Lloyd 2000).

Regardless of how little (non-zero) time, on average, it takes to test a path (i.e., ignoring *human* performance limitations as such), it is always possible to construct a software system whose paths cannot all be tested in any specific finite time. (It would suffice to add one if-then statement to a system that is the largest we can test.)

---

[9] In practical terms, a typical 300-line program with one binary branch per 10 lines, on average, would have a path complexity of 2^{30}, or roughly 10^8. Statistically speaking, therefore, a 300-line program is the largest "typical" program whose paths can be exhaustively tested in 5-10 years. This is a reasonable upper-bound on the time that would be spent testing a software system given typical funding cycles in technology projects.

Even parallelized (concurrent) testing does not provide, in general, a solution to the path-complexity catastrophe, because such testing is not always possible (e.g., testing of state-history-dependent dynamical simulations (such as large climate simulators) is not decomposable to anything smaller than an entire trajectory).

Even when parallel testing is possible, it is limited, in general, by speed-of-light-coordination of the test trajectories (i.e., at a single information-collection point; Reichenbach 1957; Susskind 2014; Deutsch 1997, Chap. 6; Lloyd 2000) of the tests. The minimum time, t_{coll}, to collect at a given spatial location, P, the reports of tests executed in parallel at M disjoint spatial locations $x_1, x_2, \ldots, x_M$, is $\sim Md/c$, where

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

(For a more detailed discussion, see Amdahl 1967). Note that for any d, as $M \rightarrow \infty$, $t_{coll} \rightarrow \infty$. Thus, no matter what value d has, there is a positive lower bound to t_{coll}, and a corresponding upper bound to the path-complexity of a software system that can be tested in any given time.[10]

Premise 4. *Even "model-based" approaches to proof of software correctness cannot, if powerful enough to capture the full flow-control of S, overcome the path-complexity catastrophe.*

Against Premise 3, it has been argued that it is reasonable to believe that software systems could be tested by "model checking" (Littlewood and Strigini 2000; Peled, Pelliccione,

[10] It is sometimes argued (Millis and Davis 2009; Nimtz and Stahlhofen 2007; Nimtz 2006) that quantum computing (Nielson and Chuang 2011) could overcome the path-complexity catastrophe because an arbitrary number of test cases could be executed in parallel on a quantum computer. Given the Copenhagen Interpretation of quantum mechanics (Bohm 1951; Messiah 1958), however, the collecting of test results requires communication of those results from the x_i to P, and this regimen is limited by synchronized speed-of-light communication and coordination of the independent test executions (Reichenbach 1957; Susskind 2014), not by the quantum theory. To date, no coherent, testably distinguishable alternative to the Copenhagen Interpretation exists, and there is strong empirical and theoretical evidence that there can be none (Bell 2004).

and Spoletini 2008; Angius 2013; Angius 2014; Baier and Katoen 2008). In outline, model checking is an algorithm that determines whether a model satisfies a special set of formulas (e.g., temporal logic formulas (Kröger and Merz 2008)) that is equivalent to a transition diagram/description of the behavior of S (Baier and Katoen 2008; Emerson 2008). *If* such a model of S has a well-characterized error distribution, the argument goes, then we can use model checking to characterize the distribution of errors in S. Alternately, we could construct a test case for a software system in such a way that if the system produced a particular result F that could not be a consequence of a given model M whose error distribution was known, we could infer that the software did not instantiate M, and therefore, we *could not infer* that the software had the error distribution of M.[11]

Now a problem immediately arises in transition diagrams/descriptions that fully (in the sense of isomorphism) characterize S: they have at least the path complexity of S[12] -- a problem known as the *state explosion problem* (Valmari 1998).

Much effort has been expended in the model-checking community to try to overcome the state explosion problem. All of these efforts hold as a working hypothesis that some way to abstract the transition-diagrams to a description that *is* tractable in testing can be found (see Angius 2013 and Emerson 2008 for a survey of these efforts).

Model-checking research has produced some notable successes. Cast in broad terms, that research program has shown that in order to avoid the state explosion problem, the expressions that can be allowed in a software system must be a proper subset of the expressions that are legal in a Turing complete language (Emerson 2008). If we can show that a software system S is written using only this privileged subset of expressions, then the correctness of S is guaranteed. In at least one case, these methods have used to prove that some low-level device-driver software is correct (Emerson 2008).[13]

---

[11] This does not imply, of course, that the software system of interest doesn't have the same error distribution as M: it merely means that we would not have a warrant to make the inference that the software has the error distribution of M, on the basis of the procedure.

[12] The set of paths of a software system S, as we have defined them, is isomorphic to a subset of transition trajectories of S, so a transition-space model will have a complexity at least as large as that of path complexity.

[13] Device-drivers are typically quite small software systems because they are typically required to run very fast. In the cases in which model-based proofs of correctness showed the drivers operated correctly, it is worth noting that

Several observations are motivated by these results.

First, the restriction of a Turing complete language to a proper subset of that language implies that we cannot, in that restricted language, express everything that can be implemented in a Turing machine, which is the canonical definition of a "computer" (Boolos, Burgess, and Jeffrey 2007). So imposing such a restriction is tantamount to reducing the scope of what we regard as "computable". There is no inconsistency in this state of affairs, but it brings into sharp relief the harsh trades between model-based proofs of correctness and computability.

Furthermore, to date all restrictions of a Turing-complete language that have been determined to be able to sustain a model-based proof of correctness prohibit the expression of even first-order logic.

The prospects of finding a model-based proof of correctness for all possible software systems have further problems. To see this, let's unpack what we mean by testing a software system to determine whether it conforms to a restriction of a Turing complete language. Ultimately, such testing (verification) must show that a software sequence S is realized on a Turing machine in a way that satisfies a software specification R. (In finite computing equipment, of course, we can't fully implement a true Turing machine, because a true Turing machine requires an infinite "tape".) In practice, we typically do not verify each step of the mapping from a given software system S to its implementation in a specific Turing machine. Instead, we posit, based on other/previous testing, that at least some of the intermediates of this mapping are unproblematic. For example, we assume that if S is written in a Turing complete language, its Turing-translate – i.e., the realization of S on a Turing-machine (generated by a compiler or assembler), is an error-free translation.

Those assumptions are far from innocuous, for several reasons.

First, there is no known error-free compiler for any of the widely used, standardized computing languages (e.g., Fortran, C++, Ada, and Java).

Second, all known Fortran, C++, Ada, and Java compilers have a path complexity greater than 10^8. For example, a C++ compiler, mapped to a Turing machine, must contain a parser

---

these drivers could also be proven correct without using model-based methods (e.g., by manual inspection), in human-tractable time.

whose path complexity of at least 10^{45}, just to accommodate the keywords of C++. Exhaustive path testing of such compilers is, by the arguments above, intractable.

Not least, in any non-trivial application of model-based proof of software correctness, we would have to use a tool, T, implemented in software, to check whether some other software system S conformed to the language restriction of interest (Emerson 2008). Using T to determine whether T is subset-conforming would be circular. So we must verify T is subset-conforming without using T. We might use some other program, T', to verify that T is subset-conforming, but if we did, we would then have to ask how it is we can assume T' is subset-conforming. To avoid an infinite regress, we have to have determined that some software system T'' is subset-conforming. In general, we cannot assume that CSIT will help us here, so we must exhaustively test all branches of T''. But that kind of testing is subject to the path-complexity limitations described in Section 3.0, which was the very problem we were trying to avoid by using model-based proofs of correctness.

There is only one escape from this regress: T'' might be small enough that it is possible to test all its branches. We have strong evidence that we won't be so lucky: to date, the path complexity of any software system capable of parsing sequences written in subset-conforming languages is comparable to the minimum complexity of a parser for the C++ language ($\sim 10^{45}$).

4.0 Discussion and Conclusions

The results of the previous section imply that a PHA cannot, in general, characterize the distribution of errors in software systems,[14] even for software systems that are not subject to the halting problem.

It strongly appears that more could be discovered by relaxing various constraints on the definition of a PHA. For example, suppose we remove the constraint on a PHA that communication has to be speed-of-light limited. Suppose, in particular, a PHA could test a specific software sequence simply by "glancing" at it and determining in a time t_{glance} that the

[14] This does not imply that scientific domains that use software are somehow less reliable than those that do not. Indeed, because in general we cannot characterize the error distribution of a scientific system that uses software, we would have no warrant (at least in CSIT terms) to compare it(s) error distribution to a scientific domain that does not use software.

sequence conformed to some pattern A. If the PHA acts in a temporally finite universe whose lifetime is L, the largest software system that could be tested by this "pattern recognition" method in L is L/t_{glance}. Any finite software system larger than L/t_{glance} could not be tested by the PHA.

Although a systematic investigation of what happens when we relax constraints on a PHA is beyond the scope of this paper, an intriguing prospect would be to allow a PHA be able to comprehend, in less than the lifetime of the universe in which the PHA exists, whether a software system of arbitrary finite size conforms to any finite pattern of interest. Posed this way, a PHA is a variant of Spinoza's God (Spinoza 1677, Part I, Prop. XI), cast in the language of the transfinite cardinals (Bernays 1968, Chapter VII).

References

Aho AV, Lam MS, Sethi R, Ullman JD (2006). *Compilers: Principles, Techniques, and Tools.* 2nd edition. Addison Wesley.

Amdahl, GM (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 JP (2008). *Principles of model checking*. MIT Press.

Bernays P (1968). *Axiomatic Set Theory*. Dover, 1991.

Bevington P, Robinson DK (2002). *Data Reduction and Error Analysis for the Physical Sciences*. McGraw-Hill.

Boehm BW, Abts C, Brown AW, Chulani S, Clark BK, Horowitz E, Madachy R, Reifer DJ, Steece B (2000). Software *Cost Estimation with COCOMO II.* Prentice Hall.

Boolos GS, Burgess JP,Jeffrey RC (2007). *Computability and Logic* (5th ed.). Cambridge.

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

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

Cormen TH, Leiserson CE, Rivest RL, Stein C (2001). *Introduction to Algorithms*. Second Edition. MIT Press and McGraw–Hill.

Deutsch D (1997). *The Fabric of Reality*. Allan Lane.

Domski M, Dickson M (eds.) (2010). *Discourse on a New Method: Reinvigorating the Marriage of History and Philosophy of Science.* Open Court.

Emerson EA (2008). The beginning of model checking: a personal perspective. In *25 Years of Model Checking*. Grumberg O and Veith H./ (Eds.) Springer. https://www7.in.tum.de/um/25/pdf/Emerson.pdf.

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

Horner JK, Symons JF. (2014). Reply to Primiero and Angius on software intensive science. *Philosophy and Technology* 27: 491-494.

Hunter G (1971). *Metalogic: An Introduction to the Metatheory of Standard First-Order Logic*. University of California Press.

ISO/IEC (2008). *12207:2008. Systems and software engineering — Software life cycle processes.*

Lloyd S (2000). Ultimate physical limits to computation. http://arxiv.org/abs/quant-ph/9908043v3.

Nielson F, Nielson HR, Hankin C. (1999). *Principles of Program Analysis*. Springer.

Nimtz G (2006). Do evanescent modes violate relativistic causality? *Lecture Notes in Physics* 702:509.

Nimtz G, Stahlhofen, A (2007). Macroscopic violation of special relativity. arXiv:0708.0681 [quant-ph].

Peirce CS. (1931). Charles Hartshorne and Paul Weiss, eds.. *Collected Papers of Charles Sanders Peirce, Vols. 1–6.* Harvard University Press, Cambridge, MA.

Peirce CS. (1957). How to make our ideas clear. In Vincent Tomas ed. *Essays in the Philosophy of Science.* The Liberal Arts Press, New York.

Reichenbach H. (1957). *The Philosophy of Space and Time*. Dover.

Spinoza B (1677). *The Ethics.* Trans. by R. H. M. Elwes (1883). Dover edition (1955).

Susskind L. (2014). Computational complexity and black hole horizons. http://arxiv.org/abs/1402.5674.

Symons JF, Horner JK (2014). Software intensive science. *Philosophy and Technology* 27 (3): 461-477.

Turing A (1937). On computable numbers, with an application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society*, Series 2, Volume 42:230–265. doi:10.1112/plms/s2-42.1.230.

Turing A (1938). On computable numbers, with an application to the *Entscheidungsproblem.* A correction*. Proceedings of the London Mathematical Society*, Series 2, Volume 43:544–546. doi:10.1112/plms/s2-43.6.544.

Valmari A (1988). The state-explosion problem. *Lectures on Petri Nets I: Basic models. Lectures in Computer Science* 1491:429-528. Springer.