Philos. Technol. (2014) 27:461–477
DOI 10.1007/s13347-014-0163-x

RESEARCH ARTICLE

Software Intensive Science

John Symons · Jack Horner

Received: 27 December 2013 / Accepted: 10 April 2014 / Published online: 9 May 2014

Abstract This paper argues that the difference between contemporary software intensive scientific practice and more traditional non-software intensive varieties results from the characteristically high conditionality of software. We explain why the path complexity of programs with high conditionality imposes limits on standard error correction techniques and why this matters. While it is possible, in general, to characterize the error distribution in inquiry that does not involve high conditionality, we cannot characterize the error distribution in inquiry that depends on software. Software intensive science presents distinctive error and uncertainty modalities that pose new challenges for the epistemology of science.

Keywords Models · Simulations · Software · Epistemology · Complexity · Post-human science

1 Introduction

A *software system* is a collection of processes and artifacts, abstract or concrete, that are essentially or by fiat associated with a sequence S of instructions written in some computer language L (ISO/IEC 2008). In this sense, a software system includes computer programs written in L, together with the design, test, implementation, and operational processes and artifacts associated with S. Hereafter, unless otherwise noted, we will restrict the term *software system* to mean S, together with any machine model M (Waite and Goos 1984, especially Chap. 3) that satisfies the semantics of L. We understand the semantics of L to be specified in model-theoretic terms along the lines presented in, for example, Chang and Keisler (1990).

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. The increasingly central role of computing technologies has changed science in

J. Symons (✉)
Department of Philosophy, University of Kansas, Lawrence, KS, USA
e-mail: johnsymons@ku.edu

J. Horner
Los Alamos, NM 87544, USA

significant ways. While the practical import of this transformation is undeniable, its implications for the philosophical understanding of scientific inquiry are less clear. Our objective in this paper is to characterize an important software-based difference between non-software-intensive science and software intensive science and to explain why this difference is relevant to philosophy of science.

Broadly speaking, the interests to date of philosophers of science in software-related topics have concerned *computational modeling* and the kinds of questions and problems that have been addressed include:

> How does the role of computational models in modern science relate to traditional debates over the nature of scientific explanation? (Woodward 2009; Alexandrova 2008; Bokulich 2011)
> What is the epistemic status of the results of inquiry that is driven by models and how do these models relate to traditional concerns regarding error and uncertainty? (Salmon 1967; Giere 1976; Mayo and Spanos 2011; Bolinska 2013)
> What are the implications of computational modeling for the ontology of scientific theories? (Chakravartty 2011)

For purposes of this paper, we consider software systems insofar as they serve attempts to realize computational models in inquiry. In this respect, we are addressing computational models in the sense that this term is typically used by philosophers of science. While we believe that attention to the role of software systems will have implications for all three of the modeling questions listed above, our focus in this paper will be on distinctive *epistemic* issues related to the role of software per se, and more specifically, about what we can know about the implications of software error distributions in scientific inquiry. Our focus is thus most closely related to, but in general is not identical with, the second of the three broad questions enumerated above.

Philosophers of science began to take note of the growing significance of computational models, simulations, agent based models, and Monte Carlo methods—in the early 1990s. Early philosophical engagement with computational models and simulations tended to be critical. So, for example, Oreskes et al. (1994) urged policy makers to avoid relying on evidence generated by software-based computational modeling techniques and challenged their scientific legitimacy. By contrast, Humphreys (1994), Winsberg (1999), and others recognized that for a range of scientific questions, computational models and simulations are often our only viable research tools. By now, it has become clear that in contexts where cognitive, economic, ethical, political, or practical barriers would otherwise loom large, software will inevitably figure in scientific inquiry for the foreseeable future (Symons and Boschetti 2013; Boschetti et al. 2012).

Because of the centrality of the concept of explanation in philosophy of science, it is unsurprising that most discussions of software in the literature have focused on determining whether and in what ways explanations derived from computational models differ from those provided by other kinds of scientific models (see for example Humphreys 1994; Guala 2002; Parker 2009; Winsberg and Lenhard 2010). In addition to debates over the nature of explanation, there has been some attention to the role of error and uncertainty in computational modeling (Winsberg and Lenhard 2010). Computational models, like all scientific models, are subject to error and

uncertainty. So, for example, as in much of traditional science, computational models depend on idealizations and like most work in the so-called special sciences, the generalizations that result from modeling are subject to ceteris paribus clauses (Batterman 2009; Symons and Boschetti 2013). In these respects, computational modeling is continuous with the traditional scientific enterprise and naturally enough, philosophers have approached computational methods with the set of distinctions and strategies inherited from traditional philosophy of science.

Some philosophers have argued (and we agree) that there is more to recent changes than simply the appearance of a more powerful set of tools for scientists. In our view, there are important new philosophical questions that arise with the use of computing technology in science. There are at least two aspects of recent changes in science that merit philosophical reflection. From our perspective, the clearest and most fundamental of these is the role of software as such in science. Strikingly, the impact of software on scientific practice has attracted almost no attention from philosophers of science. By contrast, many philosophers have addressed the second prominent aspect of this change; the appearance of powerful and relatively inexpensive computing technology for scientific modelers. Philosophers have noted that greater processing power has led to more sophisticated scientific models and that new possibilities for modeling have opened new domains for scientific inquiry. Since the mid-1990s, there has been extensive philosophical research into the role of computational models in scientific explanation.[1] While we agree that the quantitative increase in power and the accompanying widening of the scope of computational models and simulations is exciting and important, in this paper we will focus on the less widely discussed role of software in modern science.[2]

Philosophical reflection on these questions is in its early days and has been met with skepticism from traditional philosophers of science. Roman Frigg and Julian Reiss, for example, have rejected the claim that these computational models raise interesting new philosophical problems or questions (Frigg and Reiss 2009). Frigg and Reiss point out that much of the excitement among philosophers is diminished when we recognize that these apparently new questions are not unique to computational models and simulations. They argue that many of these issues have variants that arise elsewhere in the history of philosophy of science. We agree with their judgment that many of the

[1] See Symons 2008 for a discussion of how computational models have figured in discussions of the metaphysics and epistemology of science.

[2] Software has begun to perform some of the functions that, in the pre-software era, were considered distinctively *human* aspects of science. For example, Michael Schmidt and Hod Lipson described how their program *Eureqa* inferred Newton's second law and the law of conservation of momentum from descriptions of the behaviour of a double-pendulum system (Schmidt and Lipson 2009). More recently, Eugene Loukine and colleagues demonstrated a model that was able to predict unforeseen side-effects for pharmaceuticals that were already approved for consumption (Eugen 2012). These two papers represent very different examples of software intensive science: one is a system which is capable of generating theoretical insights and law-like relationships from a data set, while the other makes dramatic progress on a specific practical question of great importance. Examples like these indicate that across a broad swath of scientific endeavor, from highly theoretical to applied science, inquiry itself is no longer purely a matter of individual or collective *human* effort. Across the sciences, software-intensive systems are increasingly driving the direction of research and in some cases are already beginning to displace human researchers. Unlike previous improvements in scientific technology, computers not only extend our capacities, but are taking on at least some of the cognitive aspects of theoretical work in the sciences. Fundamental to understanding the character of post-human science is careful attention to the nature of its distinctive kinds of error and uncertainty.

questions raised in the philosophy of science with respect to the role of computational technologies have been addressed to some extent in earlier work.

However, in our view, philosophers of science who have been drawn to the topic are correct to see something philosophically significant in recent developments. While we do not agree with Frigg and Reiss that the existing literature in philosophy of computational modeling and simulation is "the same old stew," we accept the general point that the philosophical topics that have been addressed to date are not unique to computational models and simulations. Philosophical reflection on computational models and simulation is motivated by evidence that science is changing in ways that we are only dimly beginning to understand. The right place for philosophers to start, on our view, is by focusing on the role of software in science. The widespread use of software marks an obvious and undeniable difference between contemporary science and the science of the past and it has the advantage that its properties can be described in clear formal terms. For this reason, we think that careful attention to the role of software offers a useful point of entry into philosophical reflection on the distinctive characteristics of contemporary science.

2 Outline of the Argument

In this paper, we argue that there is no effective method for characterizing the error distribution in software systems containing more than a few hundred lines of code.[3] To the extent a scientific domain epistemically depends on software systems larger than a few hundred lines, some parts of its error distribution will remain unknowable.

In outline, our argument is:

1. An important task for the epistemology of science concerns the characterization of errors in the scientific enterprise. In a scientific domain that uses no software we *can* (although we might not be required, or choose not, to) at least partially characterize the distribution of errors in terms of conventional statistical inference theory (hereafter, CSIT; see, for example Hogg et al. 2005). Indeed, it is more than difficult to imagine a modern scientific discipline in which CSIT could not play some role in characterizing the distribution of error (Good 1983; Cox 2006; Mayo and Spanos 2011). CSIT, of course, has long been a concern of epistemology of science (Salmon 1967; Giere 1976; Mayo and Spanos 2011). All that is required for CSIT to be applicable to the characterization of an error distribution is that at least some aspect of that distribution can be characterized in terms of random variables (Chung 2001, esp. Chap. 3).
2. To the extent that software plays any essential role in a scientific domain, in contrast, we cannot be assured CSIT can be used to characterize the distribution of error arising from that role because the computer languages in which the software used in modern science is written (a) are not required to, and (b) in actual use in general do not, satisfy the conditions for the application of CSIT. This failure

[3] Of course, there are trivial counterexamples, such as the case of a program containing the same instruction repeatedly; say for example, the instruction "a=2" repeated an arbitrary number of times. Such examples are not representative of typical or even useful software and they certainly have no role in scientific inquiry.

to satisfy the conditions of applicability of CSIT arises from the fundamental role played by *conditionality* in modern computer languages.

3. Without CSIT to help characterize the distribution of errors in a software domain, there is no effective method[4] for characterizing the error distribution in that domain—in this case, in the software used by software intensive science—except by testing that software, and in general, testing all *paths* in the code.
4. Testing all paths in software systems that contain more than about 10^8 paths (about 300 lines of code) is intractable. Thus, we cannot, in general, characterize the distribution of errors in software system that are larger than about 300 lines of code. Most software systems used in science are much larger than 300 lines of code. To the extent that a scientific domain depends on a software system larger than about 300 lines of code, therefore, the unknowability of the error distribution in that software and the consequences of that uncertainty in a scientific domain that depends on such software, marks an epistemically fundamental difference between non-software intensive science and software intensive science.

3 Software Intensive Systems, Software Conditionality, and the Limits of Testing

In order to show how conditionality in software intensive science affects the epistemic status of inquiry we introduce a measure of conditionality that distinguishes software intensive science from non-software-intensive science. By *conditionality*, we mean that the course of a computation (i.e., the order in which software instructions are executed) is determined by, or is equivalent to constructs that can be represented by, conditional schemata, e.g., "If *x* obtains then do *y*." This *if-then* (or *if-then-else*) schema, or its equivalent, must be represented in every general-purpose (i.e., "Turing complete"[5]) modern programming language. Some modern programming languages, such Fortran and C++, which are extensively used in modern scientific practice, explicitly contain if-then-else constructs. Even Turing complete languages that do not seem to have, or on the surface would seem to rely little, on "if-then" schemata (e.g., Prolog and Haskell) must be able to represent the equivalent of that construct.[6]

In the following, we propose a precise characterization of *conditionality in a software-intensive system considered as a whole* in terms of a measure we call *path*

[4] As we use the term here, a method is *effective* for a class of problems iff (Hunter 1971, pp. 13–15)

- it consists of a finite number of exact, finite instructions
- when applied to a problem from its class, it always finishes (*terminates*) after a finite number of steps
- when applied to a problem from its class, it always produces a correct answer

[5] A computer language is Turing complete if it can be simulated on a single-tape Turing machine (Boolos et al. 2002). Being Turing complete is a condition of adequacy for being a general-purpose computer language.

[6] Here's why: The equivalent of the "if-then" schemata is realizable in a Turing machine (e.g., "not-x or y" is representable in a Turing machine (Boolos et al. 2002), which is logically equivalent to "if x, then y"). Therefore, any Turing complete language must be able to simulate the "if-then-else" schemata. *How*, specifically, one maps the Turing-machine equivalent of the "if-then" schemata into a particular Turing complete language will in general depend on the particulars of the language of interest. For the purposes of this paper, we do not need to consider the precise details of those mappings: it is enough for our purposes *that* such mappings exist.

complexity. Except where otherwise noted, we will not distinguish between an explicit realization of the "if-then" schemata in a Turing complete computer language CL per se, and the equivalent in CL of that schemata in a Turing machine.

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 (Nielson et al. 1999). We will define a *path in a software system* to be a path (Diestel 1997, p. 6) in such a graph. The *path complexity* of S, as we use that term, is the number of possible paths in that control-flow graph. The number of paths in a program increases exponentially with the number of conditional statements in S. By way of illustration, suppose a software system included the following instructions:

Line 5: If ("mass is less than 10 units" is true) then go to line 100, else go to line 6.
Line 6: Print "mass is greater than or equal to 10 units"
.
[list of instructions that have no conditionality]
.
Line 100: Set mass to 0 units

In this example, there are two paths. In one of these paths, the system skips the intermediate lines between line 5 and line 100. Those intermediate instructions are not executed. Otherwise, the system proceeds to execute the instructions on line 6. Note that the conditionality at line 5 generates two paths. Line 5 is an example of a *binary conditional statement*—a conditional statement that defines two possible paths. On average, in actual software systems, there is approximately one binary conditional statement (such as line 5) per 10 lines of software. Thus, each binary conditional statement like line 5 doubles the number of paths in a software system. Two binary conditional statements in a software system will result in 4 paths in the system, three binary conditional paths will result in 8 paths in the system, four will in 16, and so on. It is worth noting that this definition of complexity differs from *McCabe complexity*, which is the number of *independent* paths in a software system (McCabe 1976). Path complexity, as we have defined it here, captures the space of possible ways that the software system could run.

Now, consider a 1,000-line (instruction) software program that has a binary conditional statement (such as line 5 in the example above) 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. A 1,000-line program is extremely short by contemporary scientific software standards. Almost all modern scientific software systems are larger than 50,000 lines, and 100,000-line systems are common enough (Center for Systems and Software Engineering 2013; Horner 2003). Clearly, software intensive science involves programs with extremely high path complexity.

Our next task is to show why this high path complexity is philosophically significant to the practice of software intensive science. One way to evaluate the epistemic status of software intensive science is to consider the degree of confidence we should assign to the results of a system with high conditionality. 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 output of a system if we can judge it to be reliable. Given the important role of software intensive science, even in terms of the

practical role of its outputs, we need some assurance of their reliability independently of philosophical questions.

How can we determine the reliability of software systems in software intensive science? To answer this question, it is helpful to consider how an epistemic agent could cope with error in inquiry. In the case of a scientific domain that uses no software, there is a relatively straightforward approach that an agent could take. In general, the distribution of errors in such a domain can be characterized by conventional statistical inference theory (CSIT; Hogg et al. 2005, Chaps. 5–12). CSIT requires us to randomly draw (Hogg et al. 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. Typically, the sample size required to test a hypothesis of interest in such a case is small—on the order of 100.[7]

In the case of a scientific domain that does use software in an essential way, in contrast, we cannot be assured that CSIT can be used to characterize the distribution of error because the computer languages in which the software used in modern science is written (a) are not required and (b) in actual use in general do not satisfy the conditions for the application of CSIT. Those languages fail to meet the conditions of applicability of CSIT because of the fundamental role played by *conditionality* in them. More precisely, random variables, which are presumed by CSIT, as such admit of no conditionality in the above sense (Chung 2001, Chap. 3).

If we cannot, in general, apply CSIT to characterizing the distribution of errors in a software system that is used in a scientific domain because of the fundamental role conditionality plays in the computer languages used in that kind of science, then, in general, the only known effective procedure for characterizing the error distribution is to test every path in that system.[8]

Having to test every path in software system that contains more than $\sim 10^8$ paths is, for practical reasons, intractable. To see this, let us define a "test case" to be a regimen of artifacts, conditions, and processes under which a software system could be exercised to determine whether the system produces a particular expected result under that regimen. A test case would include a set of inputs for, and outputs expected from, the software system. Suppose we could define a test case for, execute, record, and analyze 1 path per second through a 1,000-line system that has on average, one binary branch per ten lines. Note that in general this rate is humanly unattainable because we must first design test cases and analyze test results. Returning, for example, to our case of the 1,000-line system with a path complexity of $\sim 10^{30}$, let us compute the time it would take to exhaustively exercise all paths, i.e. test, all paths in the system. The estimated lifetime of the universe is ~10 billion years, or about 10^{17} s. So it would take $\sim 10^{30}/10^{17} = \sim 10^{13}$ lifetimes of the universe to sequentially exercise[9]

[7] More precisely, the sample size required to attain a given confidence level is a function of the distribution of interest.

[8] One can distinguish several kinds of testing in terms of properties of control-flow graphs (Nielson et al. 1999). By "testing every path" in a software system, we mean "executing, and analyzing the results of that execution of, all edges and all combinations of condition-edges in the control-graph representation of the software system of interest."

[9] 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 these are special cases. In general, we must consider cases in which we must execute the tests serially.

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. Even within the space of relevant behaviors for applications there will be untested paths in those systems.

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 the sciences.

The foregoing considerations allow us to provide a precise sufficient condition for determining whether a scientific domain is to count as software intensive science. A scientific domain should be counted as an instance of software intensive science (SIS) if it requires the use of a software system that has path complexity of at least 10^8 (one hundred million). By contrast, non-software intensive science (NSIS) is scientific domain that does not use a software system that has a path complexity of at least 10^8. At first blush, the 10^8 path threshold in our NSIS/SIS distinction might seem arbitrary. However, the bounding considerations we discuss above show that it has the right order of magnitude. 10^8 is a clear general practical upper bound on the path complexity of code all of whose paths can be exercised during testing.[10]

This path-based distinction, furthermore, grounds a fundamental epistemic distinction: we can in principle and practice characterize the error distribution in an NSIS (even if we have to test every path in that system), but we cannot, in general, characterize the error distribution in a SIS.

Worse is true. Even if CSIT could in principle play the same role in SIS and NSIS, the sampling required by CSIT for SIS is intractable. Here's why. Nothing prohibits the errors in an SIS from being independent (Chung 2001, Section 3.3). If so, the Central Limit Theorem (CLT; Chung 2001, Chap. 7) tells us that the distribution of those errors could converge to a normal distribution (Hogg et al. 2005, Df. 3.4.1) as the size of the system increases. The dimensionality, D, of the space of paths in such a software system is the path complexity of that system. Theory tells us that the sample size K required the assessment of the hypothesis

(H) The distribution of errors is a normal distribution.

at a given confidence level in such a system, scales *exponentially* as D, and thus, exponentially as the path complexity, that is[11]

[10] High path complexity is not the only aspect of SIS that has not received adequate attention to date by philosophers. As one anonymous referee for this paper points out, the high variability in the methods, algorithms, and language choices evident in SIS also has no counterpart in NSIS, leading to, among other things, fundamental questions of commensurability among different software systems that nominally concern the same subject matter. For example, there are at least 10 widely used numerical methods for solving systems of partial differential equations, and the results they produce are in general not identical (Morton and Mayers 2005). In addition, simply changing the computer language in which an algorithm is realized is not, for some pairs of languages such as Fortran 77 and C, even well-defined because the language standards do not provide an adequate basis for inter-language translation of certain numerical types (ANSI 1977; ISO/IEC 2005; Feldman et al. 1990). Problems of this kind have led to serious errors whose origin is quite difficult to isolate in practice. (No such problems arise in NSIS.) All these issues clearly bear on the reliability of software and scientific inferences based on the use of software. These topics merit careful treatment in their own right. Here, however, we focus on the distinctively high conditionality of SIS.

[11] For a derivation, see Hogg et al. 2005 (Sections 2.6 and 9.4).

$$K = a^D \quad (1)$$

where a is a parameter that is determined by the confidence level at which we wish to assess H. For example, if D=1, and we wish to assess (H) at the 90 % confidence level, then a=100 and K=100. If all else remains, the same and D=2, then K=10,000.

Now to show that we have drawn a random sample from the population P of software paths in a given software system, we must *serially* test (because we must exhaustively compare each element of the sample with all other elements of the sample) whether the combination of results is a random sample. A lower bound on the time in which any information (a "signal", in the sense of Reichenbach 1958) can be obtained from a material comparing system is the time it takes light to traverse the classical electron radius ($\sim 3 \times 10^{-15}$ m). Given Eq. 1, therefore, even if we could test, in the time it takes light to traverse the radius of an electron ($\sim 10^{-23}$ s), whether a given element of a candidate sample is part of a random sample, the time it would take to serially verify a large enough random sample, to be such, in the smallest software system in an SIS, would be $\sim 100^{100,000,000}$ s. This implies that in general it is not practically possible to draw a random sample, tested to be such, from a software system in an SIS, in order to assess whether (H) is likely to be correct at the 90 % confidence level.

Viewed as algorithms,[12] science that contains no software (i.e., the simplest kind of NSIS) contains little if any conditionality. For example, in their canonical forms the classical theory of electromagnetism (Maxwell 1891, esp. Part IV, Chap. IX) and Newton's theory of motion (Newton 1726) contain no conditionality. Such theories are traditionally regarded as paradigm representations of scientific theories. Algorithmically, there is only one path through Newton's or Maxwell's theory, so their path complexity is 1. Arguably, a large part of the rational satisfaction that we derive from science that contains no software may be due to their lack of conditionality and the resulting lack of path complexity.

Against the above analysis, it might be argued that some software systems can be tested by "model checking" (Littlewood and Strigini 2000; Peled et al. 2008). Model checking is an algorithm that checks whether a model satisfies a set of temporal logic formulas. *If* a software system S instantiates a model whose error distribution is well characterized, that suggestion 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.[13]

The point is well taken. It is not, however, an argument against the claim that we have no practical way of characterizing the error distributions of software systems *in general*. Indeed, for model checking to characterize the distribution of errors in a software system, it must *presume* that the software system of interest instantiates a well behaved model. In actual practice, much software demonstrably does not have this

[12] An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function (Boolos et al. 2002).

[13] This does not imply, of course, that the software does not 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.

property. In general, moreover, showing that a software system instantiates a model whose error-distribution is well characterized has a time order comparable to that having to test all paths in that system.

It might be objected to the view we have presented that it is unreasonable to demand that all paths in the software in a SIS be tested before that program can be used. This concern is certainly correct, but it does not in general change the limitations of our knowledge of the distribution of errors in a software system. In friendly cases, we may be able to certify the reliability of scientific software for specific applications under very specific conditions—those that are identical to the conditions of our test cases. Given that this is the correct (and only feasible) approach to take to testing scientific software, however, we are still left with the intractable problem of characterizing, and avoiding the execution of, untested software paths.

It has sometimes been suggested that there is a general software development methodology, which, if rigorously followed, would help to greatly reduce the error-frequency in software (ISO/IEC 2008; Black et al. 2012). Although there is some truth to this suggestion, it says much less than it might at first seem. There is no known effective, or even generally accepted, theory or method for producing correct software (Graham et al. 1973). This is unlike NSIS, in which we *do* have generally accepted—indeed, mandated—method and theory (Kuhn 1970, Sections II–V).

Alternately, it has been suggested that it is possible to create software systems that automatically test other software systems (Fewster and Graham 1999). In this scheme, a "test harness" running on a computer replaces human labor in software testing. Using automated testing, it would seem that software systems of arbitrary size could be tested at the speed a computer can run, thereby obviating, if not collapsing in principle, the NSIS/SIS distinction.

Unfortunately, there are at least two problems with this proposal.

First, a test harness is a software system which itself must be tested. The testing of the test harness is subject to the same limitations of software testing mentioned in the NSIS/SIS distinction. There may be, of course, special cases in which the test harness adds no path complexity. These are special cases, however, not general solutions to the NSIS/SIS complexity problem.

Second, there is no general way to automate the creation of test harnesses for all software testing regimes. This implies that the test-harness approach at the very least is not a general way of eliminating the NSIS/SIS distinction, even though it might be a labor-leveraging approach for testing *some* software systems.

It has also been suggested that the limitations of software testing could be mitigated by using "program-slice" testing. A program slice K is the set of all statements in a set of computer language statements S that may affect the values of a set V of variables at some point p in S (Silva 2012). In program-slice testing, one tests a sequence of increasingly inclusive conditions that could affect the values of the members of V. In theory, this scheme would eventually test all possible conditions which could affect the values of the members of V.

Although this kind of approach to testing can in theory test all possible paths in a software system, it unfortunately requires testing at least all paths in S, and thus is subject to exactly the limitations discussed above.

It is of course possible, in the barest logical sense of that term, that some as-yet-unspecified testing technique might be discovered that would solve the problems we

have raised. Such a claim, however, is not a definite, evaluable hypothesis or argument as it stands. And in any case, the burden of proof lies with any proposal that asserts a specific testing technique can overcome the limitations identified above.

Once a software system becomes a part of practice in a scientific domain, our knowledge of what the system, and more generally, anything that depends essentially on what that system does, is subject to the limitations explained above (Sorenson 2011).

These issues have practical as well as theoretical import. In practice, the severity of problems generated by untested parts of software depends on the consequences of that software's behavior.

Of course, not all paths need to be tested in all practical applications. For example, we would not need to test the behavior of an airbag control system in a car for speeds over 300 miles per hour. Nevertheless, it is important to recognize that we know nothing about the behavior of the system in these untested cases.

In other regimens, we may have no choice but to test all paths in the system. For example, the software in a pacemaker might induce fibrillation if the pacemaker happens to enter an untested path. Consequently, in the case of pacemaker software, we have no choice but to test all possible paths. Because of this, typical pacemaker software by design has very low path complexity. However, there are experimental pacemakers that can be controlled over the Internet and they have SIS-level path complexity.

Similarly, given United States' commitment to observe the Comprehensive Nuclear Test-Ban Treaty (United Nations 1996), the USA must rely heavily on simulation to verify the safety and efficacy of nuclear weapons (National Coordination Office 2013). In this case, the software is tantamount to a proxy for the weapons proper, requiring very high confidence that the software reflects all behavior of such weapons in which one has an interest (Gustafson 1998).[14] In contrast, if the software is simply a "throw-away" experiment for summing, say, the first N integers, we might not care what it does if, say, we accidentally input real-valued data into that software.

One might be tempted to further refine the NSIS/SIS path-complexity threshold, but it is more than difficult to imagine what of consequence could come from such an effort. In any case, nothing in this paper would be sensitive to such a refinement.

No matter how we decide to restrict our tests, the implications of these restrictions must be well-understood and appreciated by the users of the software system. These restrictions mark the limits of the reliable domain of application of the software system. Beyond this domain of application we cannot reliably know the behavior of the system nor can we use the system to ground assertions about the world.

4 Some Types of Error Unique to Software

The fundamental role played by high conditionality/path complexity in SIS changes the epistemic landscape (that is, our ability to characterize the distribution of errors in the software in SIS) of science in ways indicated above. In general, these errors could be anything expressible in the computer languages deployed. In software used in science,

[14] The existence of a requirement for high confidence does not, as such, imply that this requirement is satisfied.

numerical errors often play a prominent role and occur with relatively high frequency. These numerical errors, in general, will occur under unknown conditions in SIS, because in general not all paths in a typical software program can be tested. In the following, we sketch a few examples of this class of errors. Numerical errors of the kind described in this section could of course occur in an NSIS if that NSIS contains software. In a software-containing NSIS, assuming all paths in the system are tested, the probability of detecting most of the occurrences of errors of these types is much higher than the probability of detecting most (or even more than a vanishingly small fraction) of the occurrences errors of these types in a SIS.

All the errors described in this section can be regarded as examples of what in Primiero's error taxonomy (2013) are called "material failures," and more specifically, "misaddressed resources." Misaddressed resources, in that taxonomy, are resources that are required but are addressed by incorrect or insufficient instructions. More specifically, the kinds of error described below concern varieties of incorrect or insufficient numerical "type" management. For a more comprehensive and systematic account of the kinds of error which could escape detection because of the practical limitations of testing, see Primiero 2013.

4.1 The Problem of Precision Management

In all modern computing languages, we can (approximately) represent any real value, R, in a canonical normalized form (sometimes called "scientific notation" form),

$$R = \pm M.NNN\ldots \times 10^{K} \qquad (C)$$

where

M is an integer between 1 and 9, inclusive

NNN… is a finite sequence of non-negative integers

K is a non-zero integer

Let Z be a digit in M.NNN …. The number of *significant digits* in an instance I of form (C) is the number of digits in M.NNN…, counting from the left, such that the values obtained under repeated determination (i.e., by repeated measurement or computation) of I lie in the range determined by Z±1. The number of significant digits is a reproducibility measure (Halmos 1950) closely related to the variance of a probability distribution and to a statistical confidence interval (Hogg et al. 2005). In the jargon of computer science, the number of significant digits is called the *precision* of R.

In the absence (and perhaps even in the presence) of comprehensive precision management, the precision generated by scientific software degrades rapidly as a function of program size. In a sobering empirical study of this phenomenon, Hatton (1997) demonstrated that ~15 randomly chosen scientific software systems each containing at least 50,000 lines of code, distributed across diverse scientific application domains, yielded results with at most one significant digit—regardless of application domain and authorship. Hatton conjectured that this striking error-rate invariant arose from potentially diverse sources ranging from failure to model error distributions in inputs, to failure to manage error-propagation within computation.

Whether a single-significant-digit result is acceptable typically depends on the application. Let's suppose, for example, that the temperature of the coolant in a nuclear

reactor must be kept between 500 and 525° Fahrenheit if the reactor is to operate safely. If the control software that manages this temperature has only one significant digit, then the actual temperature control is at best 500±100°, which would be a disaster.

More surprising still were the results Hatton obtained by analyzing the number of significant digits in the outputs of software systems composed entirely of *randomly generated* sequences of instructions. As the size of such "random"-programs goes to "infinity," he discovered that the number of significant figures in the outputs of these randomly generated sequences asymptotically approaches 1 (Hatton 2013). Programs, as small as 15,000 lines of code, clearly exhibit this tendency.

What should we infer from such results? Surely not that it is impossible to build a program that can produce more than one significant digit. Rather, it is a clear warning that in absence of aggressive error management, as a function of program size, programs will devolve to little more than noise generators.

A naïve attempt to fix this kind of problem—adding error-checking code to a program—is a devil's trade. The error-checking code itself must be tested, so in general adding error-checking code to existing software adds more de facto untestable paths to the code.

4.2 Mapping Real-Valued Quantities to Discrete-State Machines

Many scientific theories are expressed in terms of real-valued quantities (quantities whose magnitudes are real numbers). However, because digital computing machinery has no way of representing real-valued quantities exactly (technically, computing machinery can represent, at best, integers), computing representations of real values are integer approximations of real values. At the simplest level, these approximations can give rise to round-off and truncation errors.

4.3 Errors Arising from Approximations of Numerical Operations

Operations on approximated real values, *as operations*, generate distinctive kinds of errors. Any arithmetic operation, such as multiplication, on two approximated real-valued operands, generates error that is a function of that operation as such, and of the uncertainty in the operands (Taylor 1982, esp. Chap. 3). If not comprehensively managed, sequences of simple arithmetic operations (which typically occur in modern computing regimes at the rate of 10^9–10^{18} per second) on approximations of real operations can yield results that contain no information, solely as a consequence of lack of error management of these operations.

4.4 Numeric Errors Arising from Type Mismanagement

In every modern programming language, one must specify[15] the size of machine storage allocated for a representation of numeric elements (e.g., variables and literals).

[15] In some computer languages, this can be done by implicitly accepting the default specification. So-called "interpreted" languages, which include many of the widely scripting languages in the UNIX family of operating environments, determine "type" only during execution. Type management in these contexts is obviously fragile.

Most languages provide for several different storage size-allocations (technically, these allocations are part of *type specification* in the language of interest). For example, in both Fortran and C, we can allocate a smaller, or a larger, space to a given real ("floating-point") variable. Fortran and C enforce rules for operations (e.g., addition) that act on sets of operands that do not all have the same numeric type. Some of these rules are more or less intuitive, but some are arcane. It is uncommon for software engineers to pay much attention to numeric type differences of operands in a given operation, but inattention to such detail can lead to quite unexpected results—error by any other name. A widely used N-body gravitational simulator, *nbody6*, for example, has ~10,000 instances of mixed-numeric-type ("mixed-precision") arithmetic within its ~30,000 lines of Fortran. Not surprisingly, removing these mixed-precision operations produces software whose results are not the same as the mixed-precision version (Horner 2013).

The above kinds of numeric error can in principle be partially mitigated—but, for the reasons cited above, only partially—under aggressive type management. The severity of problems generated by the types of numeric errors mentioned above when they occur in untested parts of software depends on the consequences of that software's behavior, and can range from trivial to profound, as noted above. A general theory of the consequences of untested software paths is highly desirable, but beyond the scope of this paper.

5 Conclusions

There can be little doubt that science in the industrialized world has become deeply dependent on computing technology and that this dependence has changed the practice of science in significant ways. It is important to critically examine the changes in the epistemic character of inquiry that result from increased reliance on software intensive systems. Part of the task of the epistemology of science is to characterize the kinds and extent of errors in scientific knowledge.

What we have done is to explain why the kinds of errors associated with contemporary scientific practice stand in sharp contrast with those found in traditional science. In a scientific domain that contains no software, we *can* at least partially characterize the distribution of errors in terms of CSIT. CSIT, of course, has long been a concern of epistemology of science. To the extent that software systems play an essential role in contemporary science, we cannot be assured that CSIT can be used to characterize the distribution of error. This is because the languages used in software systems that are essential to the practice of science admit a kind of *conditionality* that cannot be captured using CSIT.

Without CSIT to help characterize the distribution of errors in software systems used in inquiry, there is no effective procedure for characterizing those distributions except by testing every path. Testing all paths in a typical software system containing more than 10^8 paths is intractable. We, therefore, distinguish a non-software-intensive science (NSIS) from a software intensive science (SIS) in terms of this limit.

This path-based delineation grounds a fundamental epistemic distinction: we can characterize the error distribution in an NSIS; but in general, we cannot characterize the error distribution in a SIS. A SIS therefore contains, in ways an NSIS cannot,

distinctive error and uncertainty modalities that present new challenges for the philosophy, and indeed, the responsible practice, of scientific inquiry.

Acknowledgments This work benefited from discussions with Sam Arbesman, George Crawford, Paul Humphreys, and Tony Pawlicki. We are grateful to the reviewers of earlier versions of this paper for extensive and insightful criticisms. For any errors that remain, we blame the path complexity of our (biological) software.

References

Alexandrova, A. (2008). Making models count. *Philosophy of Science, 75*(3), 383–404.

ANSI. (1977). American National Standard Programming Language Fortran. *ANSI, X3*, 9–1977.

Batterman, R. W. (2009). Idealization and modeling. *Synthese, 169*(3), 427–446.

Black, R., van Veenendaal, E., & Graham, D. (2012). *Foundations of software testing ISTQB certification.* Cengage Learning EMEA.

Bokulich, A. (2011). How scientific models can explain. *Synthese, 180*(1), 33–45.

Bolinska, A. (2013). Epistemic representation, informativeness and the aim of faithful representation. *Synthese, 190*(2), 219–234.

Boolos, G., Burgess, J., & Jeffrey, R. (2002). *Computability and Logic* (4th ed.). Cambridge: Cambridge University Press.

Boschetti, F., Fulton, E. A., Bradbury, R. H., & Symons, J. (2012). What is a model, why people don't trust them, and why they should. In *Negotiating our future: Living scenarios for Australia to 2050*, Vol 2, 107–119). Australian Academy of Science.

Center for Systems and Software Engineering, University of Southern California. (2013). *COCOMO II.* http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html.

Chakravartty, A. (2011). Scientific realism. In *Stanford encyclopedia of philosophy.* E. Zalta (Ed.). http://plato.stanford.edu/entries/scientific-realism/.

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

Chung, K. (2001). *A course in probability theory* (3rd ed.). New York: Academic.

Cox, D. (2006). *Principles of statistical inference.* Cambridge: Cambridge University Press.

Diestel, R. (1997). *Graph theory.* New York: Springer.

Eugen, L. (2012). Large-scale prediction and testing of drug activity on side-effect targets. *Nature, 486*(7403), 361–367.

Feldman, S. I., Gay, D. M. Maimone, M. W., & Schryer, N. (1990). A Fortran to C Converter. AT&T Bell Laboratories technical report.

Fewster, M., & Graham, D. (1999). *Software test automation*. Reading: Addison-Wesley.

Frigg, R., & Reiss, J. (2009). The philosophy of simulation: hot new issues or same old stew? *Synthese, 169*(3), 593–613.

Giere, R. (1976). Empirical probability, objective statistical methods, and scientific inquiry. In C. A. Hooker & W. Harper (Eds.), *Foundations of probability theory, statistical inference, and statistical theories of science* (Vol. 2, pp. 63–101). Dordrecht: Reidel.

Good, I. J. (1983). *Good thinking: The Foundations of probability and its applications*. University of Minnesota Press. Republished by Dover, 2009.

Graham, R. M., Clancy, G. J., Jr., & DeVaney, D. B. (1973). A software design and evaluation system. *Communications of the ACM, 16*(2), 110–116. Reprinted in E Yourdon, (Ed.), *Writings of the Revolution*. New York: Yourdon Press, 1982 (pp. 112–122).

Guala, F. (2002). Models, simulations, and experiments. In *Model-based reasoning* (pp. 59–74). Springer

Gustafson, J. (1998). Computational verifiability and the ASCI Program. *Computational Science and Engineering 5*, 36–45. http://www.johngustafson.net/pubs/pub55/ASCIPaper.htm.

Halmos, P. (1950). *Measure theory.* D. Van Nostrand Reinhold.

Hatton, L. (1997). The T experiments: errors in scientific software. *IEEE Computational Science and Engineering* 4, 27–38. Also available at http://www.leshatton.org/1997/04/the-t-experiments-errors-in-scientific-software/.

Hatton, L. (2013). Power-laws and the conservation of information in discrete token systems: Part 1: General theory. http://www.leshatton.org/Documents/arxiv_jul2012_hatton.pdf.

Hennessy, J., & Patterson, D. (2007). *Computer architecture: A quantitative approach* (4th ed.). New York: Elsevier.

Hogg, R., McKean, J., & Craig, A. (2005). *Introduction to mathematical statistics* (6th ed.). Upper Saddle River: Pearson.

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

Horner, J. K. (2013). *Persistence of Plummer-distributed small globular clusters as a function of primordial-binary population size. Proceedings of the 2013 International Conference on Scientific Computing* (pp. 38–44). Athens: CSREA Press.

Humphreys, P. (1994). Numerical experimentation. In *Patrick Suppes: Scientific philosopher* (pp. 103–121). Kluwer.

Hunter, G. (1971). *Metalogic: An introduction to the metatheory of standard first-order logic*. Berkeley: University of California Press.

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

ISO/IEC. (2005). *ISO/IEC 9899: TC2—Programming languages – C—Open standards.*

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

Kuhn, T. (1970). *The structure of scientific revolutions. Second edition, enlarged* (2nd ed.). Chicago: University of Chicago Press.

Littlewood, B., & Strigini, L. (2000). Software reliability and dependability: a roadmap. *ICSE '00 Proceedings of the Conference on the Future of Software Engineering* (pp. 175–188).

Maxwell, J. (1891). *A treatise on electricity and magnetism.* Third edition (1891). Dover reprint, 1954.

Mayo, D., & Spanos, A. (2011). Error statistics. In P.S. Bandyopadhyay & M. R. Forster (volume Eds.). D. M. Gabbay, P. Thagard & J. Woods (general Eds.), *Philosophy of statistics, Handbook of philosophy of science*, Volume 7, *Philosophy of statistics*. (pp. 1–46). Elsevier.

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

Morton, K. W., & Mayers, D. F. (2005). *Numerical solution of partial differential equations*. Cambridge: Cambridge University Press.

National Coordination Office for Networking and Information Technology Research and Development. (2013). DoE's ASCI Program. http://www.nitrd.gov/pubs/bluebooks/2001/asci.html.

Newton (1726). *The Principia.* Edition of 1726 (Trans: Motte, A.). 1848. Prometheus reprint, 1995.

Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of program analysis*. Heidelberg: Springer.

Oreskes, N., Shrader-Frechette, K., & Belitz, K. (1994). Verification, validation, and confirmation of numerical models in the earth sciences. *Science, 263*(5147), 641–646.

Parker, W. S. (2009). II—Confirmation and adequacy-for-purpose in climate modelling. *Aristotelian Society Supplementary Volume*, *83* (1).

Peled, D., Pelliccione, P., & Spoletini, P. (2008). Model checking. In B. Wah (Ed.). *Wiley encyclopedia of computer science and engineering*

Primiero, G. (2013). A taxonomy of errors for information systems. *Minds and Machines*. doi:10.1007/s11023-013-9307-5.

Reichenbach, H. (1958). *The philosophy of space and time*. (Trans: Reichenbach, M., & Freund, J). New York: Dover.

Salmon, W. (1967). *The foundations of scientific inference*. Pittsburgy: University of Pittsburgh Press.

Schmidt, M., & Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science, 324*(5923), 81–85.

Silva, J. (2012). A vocabulary of program slicing-based techniques. *ACM Computing Surveys* 44, Article No. 12.

Sorenson, R. (2011). Epistemic paradoxes. In E. Zalta (Ed.), *Stanford encyclopedia of philosophy.* http://plato.stanford.edu/entries/epistemic-paradoxes/.

Symons, J. (2008). Computational models of emergent properties. *Minds and Machines, 18*(4), 475–491.

Symons, J., & Boschetti, F. (2013). How computational models predict the behavior of complex systems. *Foundations of Science, 18*, 809–821.

Taylor, J. (1982). *An introduction to error analysis: The study of uncertainties in physical measurements* (2nd ed.). Sausalito: University Science.

United Nations. (1996). Resolution adopted by the general assembly:50/245. Comprehensive Nuclear-Test-Ban Treaty.

Waite, W. M., & Goos, G. (1984). *Compiler construction*. New York: Springer.

Winsberg, E. (1999). Sanctioning models: the epistemology of simulation. *Science in Context, 12*(2), 275–292.

Winsberg, E., & Lenhard, J. (2010). Holism and entrenchment in climate model validation. In M. Carrier & A. Nordmann (Eds.), *Science in the context of application: Methodological change, conceptual transformation, cultural reorientation*. Dordrecht: Springer.

Woodward, J. (2009). Scientific explanation. In E. Zalta (Ed.), *Stanford encyclopedia of philosophy*. http://plato.stanford.edu/entries/scientific-explanation/.