




# Understanding Error Rates in Software Engineering: Conceptual, Empirical, and Experimental Approaches

Jack K. Horner<sup>1</sup> · John Symons<sup>1</sup> 

Received: 13 February 2019 / Accepted: 13 February 2019 / Published online: 21 February 2019  
© Springer Nature B.V. 2019

## Abstract

Software-intensive systems are ubiquitous in the industrialized world. The reliability of software has implications for how we understand scientific knowledge produced using software-intensive systems and for our understanding of the ethical and political status of technology. The reliability of a software system is largely determined by the distribution of errors and by the consequences of those errors in the usage of that system. We select a taxonomy of software error types from the literature on empirically observed software errors and compare that taxonomy to Giuseppe Primiero's *Minds and Machines 24*: 249–273, (2014) taxonomy of error in information systems. Because Primiero's taxonomy is articulated in terms of a coherent, explicit model of computation and is more fine-grained than the empirical taxonomy we select, we might expect Primiero's taxonomy to provide insights into how to reduce the frequency of software error better than the empirical taxonomy. Whether using one software error taxonomy can help to reduce the frequency of software errors better than another taxonomy is ultimately an empirical question.

**Keywords** Software error · Error · Philosophy of software engineering · Computer science education

## 1 Introduction

To date, there is no comprehensive theory of error in software that is both standardized and widely used in software engineering practice. In general, what might such a theory of error look like? A number of approaches are conceivable. One might understand the purpose of a theory of error to involve explaining the source or sources of error. As

---

✉ John Symons  
johnsymons@ku.edu

<sup>1</sup> University of Kansas, Lawrence, USA

such, a theory of error would provide an answer to the question of why errors occur. Alternatively, one might seek a theory that explains what the essential nature of software error is. Thus, we have at least two questions that a general theory of error could aim to answer: Why do errors occur? and What is an error? While we can imagine varying ways to understand the purpose of a theory of error, a first step in the development of any such theory is to understand clear instances of error in a well-defined and relatively well-understood domain. In this paper, we focus on errors that arise in software engineering because it is a domain in which error is a pressing concern and where considerable resources have been devoted to studying and correcting it. Empirical research into software error rates is valuable, but as we shall show, it may have significant limitations. These potential limitations are cast in sharp relief by recent philosophical work on error in computing/information systems. Primiero's 2014 taxonomy of information systems in particular is coherently and explicitly grounded in an explicit abstract theory of computing and is more fine-grained than the empirical software error taxonomies in the published literature that we reviewed and thus might provide insights into how to reduce the frequency of software error. Whether any use of any taxonomy can reduce the incidence of software errors better than another is ultimately an empirical question.

There are some general features of error that philosophers have illuminated and that can serve as the basis for theoretical engagement with software engineering practice. At the most general level, for example, Nicholas Rescher notes that error in human affairs results from our being limited creatures whose "needs and wants outrun our various capabilities" (Rescher 2009, p. 2). On this view, to err is human, and insofar as humans have a hand in it, error in software engineering practice is unavoidable. In general, technological artifacts, like software-intensive systems, are created to help us exceed our innate capacities (Humphreys 2004). As we extend ourselves, it seems that we inevitably go astray. Empirical evidence seems to show that when it comes to software engineering, we go astray at predictable rates.

Any taxonomy of software error that hopes to help reduce the frequency of software errors will presumably have to be mapped to software development practices. Accordingly, in Section 2, we provide a sketch of the structure of the software development process.

Section 3 provides an overview of the results of a wide range of empirical studies of software error published between 1978 and 2018. Section 4 provides an informal overview of Primiero's 2014 taxonomy of error in information systems. In Section 5, we compare a taxonomy from the literature on empirical software errors with Primiero's taxonomy. In Section 6, we sketch an example of how one could test whether a part of Primiero's taxonomy would help to reduce the frequency of software errors better than the "empirical" taxonomy we discuss in Section 3. We further identify practical resources that can guide the construction of experiments to help assess whether one software error taxonomy is more likely to help reduce the frequency of software errors than another taxonomy. The ultimate goals of such experiments are to help determine whether at least some taxonomies of software error rates can help to reduce the frequency of software errors better than other taxonomies can, and to help us understand, at least in some ways, how error in software can limit human inquiry.

As we discuss below, empirical investigation over the past four decades shows a good agreement among average software error rates (errors per line of code), regardless of application type—*about one to two empirically observable errors per hundred lines<sup>1</sup> of code*. One natural response to this finding is to say that it indicates the need for more rigorous testing in software engineering. However, even in software that has been tested as well as is practicable, various kinds of error in most non-trivial software projects occur at the empirically observed average rate of about one error per hundred lines of code. In addition, it is likely that, for non-trivial software that is useful in scientific, engineering, or business applications, there are hard theoretical limits to the project of error correction, in addition to the pragmatic trade-offs faced by software engineers (Symons and Horner 2017). The persistence of error in software engineering is a remarkable feature of the empirical studies that we consider. While we will have to live with error, clearly, some of these errors matter more than others. Understanding the varieties of error that arise in software will be an important part of learning to cope with life in an increasingly software-dependent society.<sup>2</sup>

## 2 Software Systems, Specifications, and Errors

The empirical and conceptual parts of this paper are restricted to errors in software systems. ISO/IEC 2017 defines a *software system S* to be a collection of functions, processes, and artifacts, abstract or concrete, that are essentially or by fiat associated with a sequence of instructions written in computer languages (such as C++, Fortran, or Ada) that execute on some hardware system, within some usage context. In this sense, a software system includes computer programs written in one or more computer languages (See Symons and Horner 2014; Horner and Symons 2014).

A system specification, *H*, is a set of imperative sentences (called “requirements”) that state *what* objectives a system must achieve, but not *how*, at least in all details, the system must achieve those objectives.<sup>3</sup> For example, the imperative “By 1969, the send a man to the moon and bring him back alive” is part of *what* a system must achieve, but “Use a Saturn rocket whose flight software is implemented in IBM assembly language” addresses part of *how* the objective might be satisfied.<sup>4</sup>

<sup>1</sup> For the purposes of this paper, a line of code is a “statement,” as “statement” is defined in the standard for a given computer language. For a closely related view, see Table 2.53 in Boehm et al. (2000).

<sup>2</sup> Petricek 2017 contains an engaging dialogue about how we might approach living with software error. For our view on what it means to live with error in the epistemology of software-intensive research, see Symons and Alvarado (2019).

<sup>3</sup> In practice, it is common to distinguish elements of a specification that are mandatory from those that are not. Typically, the qualifier “must” or “shall” is used to delineate a mandatory requirement.

<sup>4</sup> Alternately, the imperative sentences of *H* can be classified in terms of functional, vs. non-functional requirements. Informally, functional requirements define what a system is supposed to *do* and non-functional requirements define how a system is supposed to *be*. Functional requirements are usually in the form of “the system shall do <requirement>,” an individual action or part of the system, perhaps explicitly in the sense of a mathematical function. In contrast, non-functional requirements are in the form of “the system shall be <requirement>,” an overall property of the system as a whole or of a particular aspect and not a specific function. We thank an anonymous reviewer for noting these alternatives. Nothing in this paper, however, depends on whether requirements are organized under the “what/how,” vs. the “functional/non-functional,” taxonomy.

A software specification,  $H_{\text{soft}}$ , is a set of imperative sentences that state *what* objectives  $S$  (the software in the system) must achieve, but not *how*  $S$  must achieve those objectives.<sup>5</sup>

For expository convenience, we posit that  $H_{\text{soft}}$  is *not* an element of  $S$ . From a high-level software engineering perspective, errors that may arise in  $S$  involve life cycle process phases and associated products (artifacts) of  $S$ . Given  $H_{\text{soft}}$ , those life cycle processes are:

- Creating a logical (sometimes called a “functional”) design (LD) of  $S$ , allocating elements of  $H_{\text{soft}}$  to the LD, then
- Creating a physical design (PD) for  $S$  that includes extensive realizations of  $S$  in computer languages and allocation of elements of the LD to the PD, then
- Implementing the PD in computer languages to run on designated hardware, then
- Integrating and system testing  $S$ , then
- Deploying  $S$ , then
- Maintaining/sustaining  $S$

Each of these phases typically generates documents and uses processes that are subjected to review (including detection of errors) during the life cycle.<sup>6</sup> Several variants of the life cycle process are possible (see, for example, Stutzke 2005, esp. Chap. 10), but apart from mandating that  $S$  satisfies  $H_{\text{soft}}$ , these differences make no difference for the purposes of this paper.

However,  $S$  is implemented, we posit that in order to be correct,  $S$  must *satisfy*  $H_{\text{soft}}$ ; in our view, this satisfaction relation can be characterized in model theoretic terms.<sup>7</sup> If  $S$  does not satisfy  $H_{\text{soft}}$ , we will say  $S$  *has a software error*.

### 3 Review of Empirical Studies of Software Error

As early as 1969, it had become apparent to the software engineering community that the number of errored instructions in software systems was increasing primarily as a function of software system size (i.e., as the number of lines of software in the system of interest; Royce 1970). This observation suggested that a quantitative characterization of error rates in software is relevant to the management of complex programming projects. Numerous studies along these lines were published in the late 1970s and early 1980s. After this initial period of high interest, however, the number of quantitative

<sup>5</sup> We thank an anonymous reviewer for recommending we emphasize the distinction between a software specification and a system specification. In a given system, the software requirements are a proper subset of the entire set of system requirements.

<sup>6</sup> In general, the process of allocating  $H$  to the elements of the system proceeds by phases and involves “deriving” and allocating requirements from previous phases in the life cycle. The end result of this allocation process is an acyclic graph of allocations to elements of the developed system. A systematic account of allocation is beyond the scope of this paper. For our present purposes, all that matters is that allocation ultimately yields a mapping,  $A$ , in the set-theoretic sense, from the models of  $H$  into the models of artifacts and processes of the deployed system.

<sup>7</sup> We believe the satisfaction relation can be captured as follows: “ $S$  satisfies  $H_{\text{soft}}$  if every model  $M$  of  $H$  is homomorphic to some model of  $S$ .” It is not, however, essential to this paper that this specific formulation characterizes the notion of “satisfying a software specification.”

empirical studies of software error rates published per year diminished until the early 2000s. Thereafter, interest in the quantitative characterization of empirical software error rates grew again. Among the reasons for this renewed interest was the increasing codification and institutionalization of software engineering processes beginning in the late 1990s (e.g., IEEE 2000; ISO-IEC 2017). The growing importance of security, concerns about liability, and an increase in theoretical consideration of software engineering practice among philosophers and other humanists may have also contributed to the revival.

Managing error in large, multi-person software projects is now a matter of considerable concern and attention. Despite this concern, most contemporary software development projects almost never involve statistical hypothesis testing based on quantitative error rates (DeMarco 1982; Humphrey 2008): typically quantitative error analysis in large software projects emphasizes tracking the number of errors, with relatively little attention to the nature of those errors, as a function of time.<sup>8</sup>

In practice, of course, software engineering projects catch some errors during software development by using model-checking, compilers, linkers, software development environments, formal reviews, and testing. Inevitably, however, errors escape these checks and are detected, if at all, only after the software has been deployed.

Roughly 100 empirical studies have been published since 1970 on software error rates. To help characterize the empirical software error distribution reported in the literature, we first examined every abstract in *IEEE Transactions on Software Engineering* that mentioned “software error” or “errors in software.” We had two objectives. The first was to estimate the range of empirical software error rates (expressed in errors per line of code), aggregated over “all” error types reported in the surveyed literature. We hypothesized (but did not assume) that this aggregate rate was, to coarse approximation, on average on the order of a few errors per hundred lines of code. The second objective was to assess whether the error types reported in the literature could be subsumed under a relatively small, but plausibly informative set (containing ~5 members) of taxonomic categories that (a) had been reported in the empirical software error literature and (b) was mappable to measurands obtainable during the software development phases mentioned in Section 2.

If the abstract of a paper chosen as noted above literally mentioned “software error” or “error(s) in software” we reviewed the associated full paper. If the body of such a paper reported studies (as a report on new research, or via references to the literature) of the error rates in at least five<sup>9</sup> projects, a broad range of error types, and a variety of application types, we extracted a paper identifier, the error types, their associated application type, and the frequency of those types reported that paper. In addition to our survey of the literature, one of us (AU No. 1) conducted an error analysis of the

<sup>8</sup> Large software projects typically use an error tracking system that records error reports generated by developers and testers and records actions taken (status tracking) concerning those reports (see, for example, Plutora 2018; Mantis 2019; Micro Focus 2019).

<sup>9</sup> As a rough rule-of-thumb, a  $t$  test on a sample size of 10 is often sufficient to support practical statistical inference ( $P < 0.1$ ) about the values of the parameters of *normal* distributions. The actual sample size required for a given statistical test depends on the distribution-type of the population, the particular kind of sample statistics computed, the type of test, the statistical hypothesis, the experimental design, and what is known, independently of the test of interest, about population parameters. See, for example, Crow et al. (1955); Devore (1995); Gopal (2006); and Hogg et al. (2005).

Linux operating system (excluding the kernel) software using the *splint* static analyzer (University of Virginia 2018).

Eleven papers (of the ~100 *Transactions in Software Engineering* papers, 1970 to the present, whose abstract contained the phrase “software error” or “error(s) in software” we reviewed) met all of the criteria described above.

We chose to call a summary of the indicated software error information for a given paper an “error descriptor” (for that paper). On the surface, the 11 error descriptors we abstracted from the literature appeared to span more than 100 types of software errors with almost no (literal) duplication of error types from paper to paper.<sup>10</sup> That result, where no further analysis possible, would imply that little to no interesting statistical structure could be inferred by comparing the papers we reviewed. On closer inspection, however, it became clear that all the error types reported in the literature we examined could be subsumed under a small set of empirical software error types described in Bowen (1980) and Boehm (1981), p. 383: misunderstanding of the specification, errors in logic, errors in numerical precision/accuracy, and errors in memory management. These categories can straightforwardly be mapped to the development phases outlined in Section 2 (see, in particular, Boehm 1981, p. 383). The categories in this taxonomy are briefly described in the following (Sections 3.1, 3.2, 3.3, and 3.4).

### 3.1 Misunderstanding the Specification

The kind of error most frequently reported in these studies was “Misunderstanding the specification.” Diagnosing this error type presumes that the specification is clearly and unambiguously formulated (at least for sufficiently informed, well-intentioned readers). In many software system development projects, however, the software engineers assigned to implement to a part of the specification may not be fluent in the domain the specification addresses, and often enough, do not even know who the specification writers are. This can lead to all manner of misunderstandings. An engineer could fail to understand the specification in at least two different ways. For example, the engineer could fail to understand that a particular model or theory (e.g., Newtonian mechanics) was (perhaps implicitly) presumed by the specification. Or, the engineer might incorrectly allocate or map the requirements stipulated by the specification. For example, suppose the engineer were given the requirement to compute the trajectory of a missile in real time. Suppose further that the engineer allocated this requirement to a machine that was too slow to perform the computation. This would be an error in allocating requirements.

### 3.2 Errors in Numerics

Many application types require a representation of real numbers (e.g., in the representations of forces, distances, and temperatures; more generally, continuous real-valued quantities). In general, modern digital computing systems can only approximate the true value of real number quantities of interest. In some applications, the difference between the digital approximation and the true value of the quantity of interest does not matter and can be ignored. But it often does, and this is a relatively common source of

<sup>10</sup> There is no widely used (“standardized”) taxonomy for software error types in the literature.

error. More specifically, it is not uncommon to find errors in deployed software arising from the following sources (the list is neither mutually exclusive nor jointly exhaustive; see University of Virginia 2018 for a more comprehensive and more fine-grained (C-language-oriented) list):

- a. The developer does not understand, or provide protection against, the numerical accuracy and precision limitations in a numerical algorithm.
- b. The developer is not aware of, or does not pay attention to, the limitations of numerical type conversions in the computer language. For example, a developer might try to replace the value of a 64-bit-precision variable with the value of a 128-bit-precision variable. The precision of the result could be 64 bits, 128 bits, or ill-defined. In many software development environments, this kind of error, even if it is undefined in the language of interest, is not reported by the compiler.
- c. The developer is not aware of smallest positive value representable in the computing environment (in computing jargon, often called the “platform epsilon”). Every computing environment has a smallest positive value that is representable in that environment. Any attempt to compute a value smaller than this may result in an undetected error,<sup>11</sup> in some cases with disastrous results.
- d. Most scientific software (e.g., of the kind used in hydrodynamics, electromagnetics, acoustics), a software system accepts (as inputs, or asserts as constants) values derived from measurements that are external to the software. Those measurements have whatever accuracy and reproducibility limits they do. If an engineer does not fully reflect these accuracy and reproducibility limits in software, the software can generate results that appears to have higher accuracy or precision than the measurements themselves have. Hatton, for example, studied this problem in detail and concluded that across a wide range of deployed scientific application software systems, the output had, when represented in scientific/exponential notation, at most *one* significant digit, even though the output often reported seven or more digits when expressed in exponential notation form (Hatton 1997).

### 3.3 Incorrect Logic

Arguably, logic errors could arise in almost any aspect of any taxonomy of errors. Given that programmers are not logically omniscient, we cannot expect to prevent logic errors in a system with even moderate levels of complexity. Among the more common types of logic errors are:

- a. The developer does not understand the logical structure of what is being represented. Suppose, for example, that a payment tracking system is intended to separately track paying by cash, check, or credit card. Suppose that the software engineer lumps paying by cash with paying by check, not realizing that the requirement is to track payment mode each on its own terms.

---

<sup>11</sup> The performance cost of detecting every possible occurrence of this of error, in most computing systems, is prohibitive.



- b. In a mutually exclusive list of logical cases, the developer fails to manage one or more cases. Suppose, for example, that in the payment tracking system mentioned in a, the software engineer fails to track “paying by cash.”

### 3.4 Memory Management Errors

All of today’s general-purpose computing systems contain physical memory (hardware in which information is stored and from which it is retrieved) that must be managed. At some level, these computing regimes require management of memory allocation (designating specific memory for use by a program), referencing (identifying a location in memory), writing to and reading from memory, and deallocation (releasing previously allocated memory for other uses) in order to satisfy speed (“time complexity” (Aho et al. 1983, 2006, Section 1.4)) or space (“space complexity” (Aho et al. 1983, Chap. 12)) constraints. In some languages or applications, this kind of detail is managed automatically, by the operating system, a run-time environment, or the compiler/linker. In such cases, the software engineer does not have to explicitly manage memory. In some languages or environments, memory management must be done by the application programmer.

We “rebinned” the error types in the first set of descriptors to the four categories described in Sections 3.1, 3.2, 3.3, and 3.4 and, for each record, transcribed or computed the average error rate, and the standard deviation of that rate where data was available to support such a calculation, *per line of code*, for each of the projects reported in the papers we analyzed. The range of software error rates per line of code obtained from this analysis was 0.5 to 3.0 per hundred lines of code, consistent with our initial hypothesis about this rate. Table 1 contains a summary of our results for the 11 papers from the literature and the Linux source code error analysis performed by AUT No. 1.

### 3.5 Limitations and Caveats

Most of the projects referenced in Table 1 were developed under best practicable engineering standards.<sup>12</sup> The error rates in Table 1 are those found in the referenced software projects *after* the software became operational.

The taxonomy described in Sections 3.1, 3.2, 3.3, and 3.4 certainly does not exhaust the varieties of errors and failures that software projects encounter. Empirical studies that focus on deployed software will tend to miss many important ways that things can go wrong. For example, it has been estimated that about 15% of software projects are terminated (“fail”) before, or shortly after, completion of development because of errors in the specification, i.e., the specification does not actually state what the procurer/user needs/wants (DeMarco 1982, p. 3; Stutzke 2005, p. 9). No data from projects that “fail” in this sense are reflected in Table 1.

In addition, computing hardware is susceptible to failure. Some of these failures are transient and are detected and automatically corrected by the computing environment.

---

<sup>12</sup> Those standards have evolved over the surveyed range of publications dates (1970 to the present), but none of those changes materially affect our analysis.



**Table 1** Some descriptive statistics of empirically observed software error rates obtained at the completion of system test, for a range of software application types

Application type	Types of error	Fraction of sampled lines errored, average	Fraction of sampled lines errored, standard deviation	References
Banking	Misunderstanding the specification, logic, numerics	0.01 (10-year total)	0.011 (10-year total)	Banker et al. 2002, p. 34
Military towed marine sensor array (SURTASS)	Misunderstanding of specification, logic, numerics	0.03	Not available	Thielen 1978
Various	Various	0.02	Not available	Charette 2005 <sup>a</sup>
Various	Misunderstanding of specification, logic, numerics	0.02 (average across all app. types)	Not available	Jones 2008, pp. 433–434
Various Java and C++ applications	Misunderstanding the specification, logic, numerics	0.02 (average over all app. types)	0.01	Phipps 1999
Various	Misunderstanding the specification, logic, numerics	0.02	Not available	Thayer et al. 1978
Various	Misunderstanding the specification, logic, numerics	0.03 (average over all app. types)	Not available	Boehm 1981, p. 383 <sup>b</sup>
Various	Misunderstanding the specification, logic, numerics	0.03 (average over all app. types)	Not available	Jones 1978
Various US-produced software	Misunderstanding the specification, logic, numerics	0.02 (average over all app. types)	Not available	Jones 1981 <sup>c</sup>
Various	Misunderstanding the specification, logic, numerics	0.02 (average over all app. types)	Not available	Humphrey 2008, p. 5 <sup>d</sup>
Operating system kernels	Logic, memory management	0.005	Not available	Malkawi 2014 <sup>e</sup>
Linux operating system, excluding the kernel	Logic, incorrect type conversion, memory management	0.02	0.01	Unpublished <i>spint</i> results obtained by AU No. 1, 2018)

<sup>a</sup> The error rates stated in Charette 2005 are an order-of-magnitude estimate

<sup>b</sup> Boehm 1981 reports this data as a function of project phase; the summary spans ~200 projects

<sup>c</sup> Jones 1981 reports this data in summary form only; the summary spans ~100 projects

<sup>d</sup> Humphrey 2008 states this data in a high-level summary form only; the summary spans ~500 projects

<sup>e</sup> Malkawi 2014 states that the owners of the data prohibit reporting the identity of the operating system(s) of interest

Some are not. No computing hardware failures are considered in the empirical studies we evaluated, nor are they discussed further in this paper. Hardware failure is an unavoidable reality for software-intensive systems insofar as software must be implemented in physical devices and these devices are subject to the contingencies of the physical world.<sup>13</sup>

Furthermore, a software error has at least two dimensions: the type of the error, considered independently of how the software system is used, and the consequences of the error in the context in which the software system is used (see MISRA 2013; Rescher 2009, p. 4). An error in one usage context might be of no consequence, but that same error might be fatal in another usage context. The taxonomy in Table 1 does not sharply distinguish these two dimensions.

Not least, developers rarely publish metrics, including error metrics, on their projects (DeMarco 1982). There are at least two reasons for this. First, it takes time and money to analyze error metrics, and almost no software procurement specifically pays a developer to attend to software error metrics analysis.<sup>14</sup> Second, a published error metric discloses something about the performance of the developer, and this can create the perception of a developer weakness or even lead to legal liabilities.

These limitations and caveats are not intended to be a criticism of the empirical work conducted to date. Regardless of how one approaches a characterization of software error, it is likely impossible to find a taxonomy that strictly partitions (i.e., provides a mutually exclusive, jointly exhaustive decomposition of) software error space. A typographical error, for example, could be associated with essentially any kind of error.

### 3.6 Interpreting the Results of Empirical Studies

At least three significant observations can be made about the empirical studies of software error summarized in Table 1. Most strikingly, there is good agreement among averages of aggregate empirical software error rates reported from 1978 to 2018, regardless of application type—*about one to two empirically observable errors per hundred lines of code*. This is strong indication that the empirically observed error rate in software development has been roughly constant over nearly 40 years of software engineering practice, despite explicit community wide efforts to reduce that rate. Note also (see Table 1) that standard deviations of the empirical error rates, where determinable from the published literature, are large relative to the average error rates. This suggests that the empirical software error rate data has, at least in some studies, relatively high mean deviation (i.e., the mean deviation is about the size of the sample average; Crow et al. 1955, p. 13). Third, the *splint* error measurements (performed by AU No. 1) on Linux are unexpected. Why? Linux is used in, or used in the development of, roughly 10 billion devices worldwide. We would expect a widely used operating system to have among the lowest empirical software error rates. A *splint* analysis suggests that this expectation is misplaced—Linux, exclusive of the kernel, has an empirical error rate of 2% (on average, there are two errored C-language statements per 100 C-language

<sup>13</sup> We discuss the role of such unavoidable constraints due to the character of physics in Symons and Horner (2017).

<sup>14</sup> In a competitive environment, of course, a developer that paid *no* attention to software error would go out of business.

statements, exclusive of the kernel, making the Linux error rate (exclusive of the kernel) comparable to that of non-operating-system code. The consequences of these errors in any given usage regimen, of course, is a separate question.

Analysis of empirical studies of error rates shows some striking features, as illustrated in Table 1. However, the error types shown in Table 1 may not shed sufficient light on how we might best reduce error rates. It seems plausible that individuating and understanding the kinds of errors that arise, their prevalence, their seriousness, and ideally their sources, might help us to make progress on reducing the prevalence of the errors that we care about. Whether a given taxonomy of software errors will help to reduce the frequency of software errors better than another taxonomy is ultimately an empirical question. We regard Giuseppe Primiero's 2014 taxonomy of errors in information systems as a promising alternative to the empirical taxonomy of software error shown in Table 1. In the following (Section 4), we briefly review Primiero's taxonomy.

## 4 An Overview of Primiero's Taxonomy of Errors in Information Systems

Primiero's (2014) taxonomy of errors in information systems recognizes that computing systems have both epistemic (informational/semantic content) and computational ("mechanical operational"/"instructional") features, and these are related in various ways. An adequate account of error in computing systems must take both these features and their relationship into account (see Piccinini 2015, esp. Chaps. 1–7, for a critical survey of these issues.) To achieve this end, Primiero begins by characterizing information systems in terms of an informational semantics that is cast in a procedural idiom (analogous to the idiom of Abstract State Machines).<sup>15</sup> The procedural idiom includes two main operations on informational contents: *access* and *use*. In that idiom, local *validity* (i.e., "true in a given context") of informational content "is explained in terms of the instructions needed to reach a given state; global validity is given by the set of states the system goes through to reach a given goal" (Primiero 2014, p. 251). "Moving to reach a goal is explained by *accessing* the information at a given starting state and *using* that information by performing syntactic transformations on it, obtaining the next state." (Primiero 2014, p. 251).

Primiero 2014 then defines a taxonomy of errors whose highest-level types are *mistakes*, *failures*, and *slips*, as follows.

On Primiero's account, "[m]istakes are errors involving the description and the design of the problem to be solved, or the specification to be implemented. [...] resulting from a faulty or incorrect explanation and presentation of the object for which a validation procedure is required." (Primiero 2014, p. 259). Mistakes have both conceptual and material subtypes.

A *conceptual mistake* is a mistake in which a [...] pair (P, G), where P is a procedure and G is a goal, contains or refers to an ill-defined "category"<sup>16</sup> A (Primiero 2014, p.

<sup>15</sup> Characterizing how the epistemic content of a computing system could be cast in a procedural idiom involves some interesting, and likely difficult, questions, such as "Can we reduce 'knowing' to a procedural idiom?" Addressing those questions is beyond the scope of this paper.

<sup>16</sup> By "category" in this context, we take Primiero to mean "type," in the sense of the reserved term "type" his formal characterization of information systems.

259). For example, consider a specification that requires as its subroutine an algorithm to compute Newtonian velocity. To define the domain of such function in a way that accommodates Einstein-relativistic behavior would represent a conceptual mistake.

A *material mistake* involves “[...] the structural design of the strategy, where a pair (P,G) is given that includes elements operations or processes that do not constitute a strategic (sub-) goal to G. [...] This means that at the design level (e.g., in the physical design of the software) there is an error implementing the system requirements specification.” (Primero 2014, p. 259). For example, suppose the system specification required computing leap years (which must be divisible by 4) but the implementation tried to compute leap years by testing to see whether the years were divisible by 3.

Primero defines failures as “[...] errors explicitly referring to the rules used in the evaluation and resolution of the problem (respectively, the validation of the specification) or related to the resources these rules have to access.” (Primero 2014, p. 260). As with mistakes, Primero explains how failures have both conceptual material subtypes. *Conceptual failures* involve “[...] problems in the selection and formulation of rules or strategies.” (Primero 2014, p. 260).

Material failures involve “[...] problems related to the accessibility of the resources required for the correct execution of a procedure for the problem or specification at hand.” (Primero 2014, p. 260). All memory management errors in the sense of Table 1, for example, are material failures.

The least clearly delineated type of error discussed by Primero is what he calls *slips*. According to Primero, slips are “errors generated by the applications of rules that are appropriate to the given goal, but that do not match some formal criteria.” (Primero 2014, p. 261). Slips have both conceptual and material subtypes.<sup>17</sup> *Conceptual slips* are “practical errors related to algorithm design, i.e. where the selection of range and domain, the order of rules applied and subrecursion definitions are chosen for an efficient algorithm” (Primero 2014, p. 262). For example, we might try to apply a second-order-valid numerical integration routine that requires a fourth-order-valid integrator.

## 5 Comparing Empirical Software Error Studies with Primero’s Taxonomy of Errors

With these definitions in hand, we can now compare the empirical software error taxonomy of error types in Table 1 with Primero’s taxonomy of errors in information systems (see Table 2).

Two observations immediately follow from Table 2. First, it is clear enough that the error types described in the literature on empirical software error rates (i.e., the error types shown in Table 1) correspond in various ways to some of the error types in Primero’s taxonomy. That correspondence, however, is not always one-to-one, even where the empirical, and Primero’s, categories at least partially overlap. Second, there are error types in Primero’s taxonomy for which there is no clear counterpart in the

<sup>17</sup> Primero infelicitously defines slips as “material errors” (Primero 2014, p. 261), thus making the subcategories of conceptual and material slips less clear than they should be. We simply strike “material” from the original definition of slip here for the purposes of our exposition.

**Table 2** Correspondence between empirical software error types shown in Table 1 and elements of Primiero's (2014) taxonomy of information system errors. S = misunderstanding the specification, N = numerics, L = logic, M= memory management, as defined in Table 1. ND = not well-differentiated in the empirical software error literature

Primiero's (2014) taxonomy		Correspondence to the taxonomy of errors in Table 1
Mistakes	Conceptual	S (ND)
	Material	L (ND)
Failures	Conceptual	S
	Material	N (ND), L(ND), M
Slips		ND
	Conceptual	ND
	Material	ND

empirical software error types of Table 1. That observation implies that the set of error types in the existing empirical studies that we reviewed are, relative to Primiero's taxonomy, less fine-grained and thus *could* fail to capture some important features of software error. In this sense at least, philosophical analysis can provide a more discriminating differentiation of error types and dimensions than we find in the existing empirical software error literature.

## 6 How Do We Determine Whether Settling on One Taxonomy Reduces the Frequency of Software Errors Better than Another?

Whether any particular taxonomy, as realized in a given set of practices, can help to reduce the frequency of software errors better than another taxonomy (as realized in a given set of practices), are empirical questions that are answerable only by experiment. In such an experiment, we must first define what experimental question we wish to pose, use that to help identify data is to be collected, how that data is related to the taxonomy of interest, under what protocol the data will be collected and analyzed, and what "better" means.

We can sketch an example of such an experiment. In this experiment, let us suppose we want to gain some insight into whether lumping all "misunderstandings of the specification" into a single category helps to reduce the frequency of software errors better than tracking misunderstandings of the specification under multiple subcategories. We might perform the experiment in a software engineering class of the kind typically offered at the undergraduate computer science level. We divide the class into two development teams, A and B. Both teams are assigned the "same" software development problem. Both teams have formal tracking and resolution of software errors that arise in the course of the development process. During error tracking and resolution, Team A, which we will call the "empirical taxonomy-oriented" team, would classify "misunderstandings of the specification" in a single category. Team B (which we will call the "Primiero-taxonomy-oriented team"), in contrast, would classify "misunderstandings of the specification" under various subcategories, including, for example, "inherently ambiguous specification language," "software engineer failure to understand logical implications of a clearly written specification," and "software

engineer failure to choose an algorithmic approach relevant to the specification.” What Teams A and B do with the software error data they acquire during development is left to the teams to decide. Both teams complete the project, and the frequency of software errors in the two approaches is recorded. The experiment is repeated a sufficient number of times to ensure that the sample size is large enough to ground conventional statistical hypothesis at a significance level of interest (see, for example, Crow et al. 1955; Devore 1995; Gopal 2006; Hogg et al. 2005).<sup>18</sup>

A comprehensive account of how to construct experiments to help determine whether one error taxonomy, as used in practice, can help to reduce the frequency of software errors better than another taxonomy, as used in practice, is beyond the scope of this paper. Fortunately, there are several good engineering practice-oriented sources available on the subject. Stutzke 2005 (esp. Chaps. 16 and 25), for example, provides an excellent overview of the issues that must be addressed. Grady 1992 provides useful checklists and guidance on implementing a practical software metrics (including error metrics) program. McGarry et al. 2002 provide good definitions for many error-related quantities of interest. Florac and Carleton (1999) identify possible measures and discuss selecting, collecting, and analyzing software (including software error) metrics.

## 7 Discussion

Primiero’s (2014) characterization of information systems can support a more general notion of information system error than the “mistake/failure/slip” taxonomy that he provides. One could argue, more generally, that *any* aspect of any computing system that is inconsistent with any feature of a given formal characterization of information systems could be called an “error in an information system.” Even within Primiero’s taxonomy of errors, the “mistake/failure/slip” types represent just one of several logically possible partitioning of the space of error types (in the generalized sense of “error in an information system” mentioned in the previous sentence).<sup>19</sup> How one carves up the space of the taxonomies of error that can be defined in terms of a given characterization of an information system will depend on the interests of the “partitioner.” Those interests may include, but are not limited to, the concerns of the community of software engineering practitioners.

## 8 Conclusions

The reliability of software is intimately related to, if not fully determined by, the distribution of errors in software systems and the uses to which that software is put.

<sup>18</sup> In a typical undergraduate *computer science* degree program, a *software engineering* course is offered as an elective, and typically only one section of that course is taught per year. In an undergraduate *software engineering* degree program, such as that offered at Carnegie Mellon University, multiple sections of software engineering courses are offered every semester.

<sup>19</sup> Primiero’s interest in partitioning information system error space into mistakes, failures, and slips appears to be twofold. First, that particular partition corresponds in some natural ways with the way software engineering practice talks about errors (Primiero 2014, p. 258). Second, Primiero’s classification corresponds reasonably well with at least one classification of errors in science (Primiero 2014, p. 258).

Even in software that has been tested as well as is practicable, various kinds of error in software occur at the empirically observed average rate of about one error per hundred lines of code. Reducing the frequency of software error is a major goal of software engineering practice. Empirical software error rate studies provide at least some insight into the nature of software error.

In this paper, we compared the taxonomies of empirically observed software error rates reported in the published literature with Primiero's taxonomy of error in information systems. This comparison shows that although philosophical research into the problem of error in software has barely begun, this kind of analysis of software error can provide a more fine-grained than the taxonomy of software errors than that provided by the taxonomy of empirical error we selected in Section 2, in several important respects. Philosophical analysis can help to formulate software error taxonomy comparison hypotheses for statistical testing, while at the same time grounding those hypotheses in an epistemically and logically transparent model of computation.

**Acknowledgments** We wish to thank the anonymous reviewers of this paper for insightful, constructive suggestions. For any errors that remain, we are solely responsible.

**Funding Information** John Symons' work is supported by The National Security Agency through the Science of Security initiative contract no. H98230-18-D-0009.

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

## References

- Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: principles, techniques, and tools*. 2nd edition. Addison Wesley.
- Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (2002). Software errors and software maintenance management. *Information Technology and Management*, 3, 25–41.
- Boehm, B. W. (1981). *Software engineering economics*. Prentice Hall.
- Boehm, B. W. et al. (2000). *Software cost estimation with COCOMO II*. Prentice Hall.
- Bowen, J. B. (1980). Standard error classification to support software reliability assessment. *Proceedings of the National Computer Conference*. pp. 697–705.
- Charette, R. N. (2005). Why software fails. *IEEE Spectrum*.
- Crow, E. L., Davis, F. A., and Maxfield, M. W. (1955). *Statistics Manual. NAVORD Report 3369 NOTS 948*. U. S. Naval Ordnance Test Station, China Lake, CA. Dover reprint, 1960.
- DeMarco, T. (1982). *Controlling software projects*. Yourdon Press.
- Devore, J. L. (1995). *Probability and statistics for engineering and the sciences*. Fourth Edition. Duxbury Press.
- Florac, W. A., and Carleton, A. D. (1999). Measuring the software process: statistical process control for software process improvement. Addison-Wesley.
- Grady, R. B. (1992). *Practical software metrics for project management and process improvement*. Prentice-Hall.
- Hatton, L. (1997). The T-experiments: errors in scientific software. *IEEE Computational Science and Engineering*, 4(2), 27–38.
- Hogg, R. V., McKean, J. W., and Craig, A. T. (2005). *Introduction to mathematical statistics*. Sixth Edition. Prentice Hall.
- Homer, J. K., & Symons, J. F. (2014). Reply to Primiero and Angius on software intensive science. *Philosophy and Technology*, 27, 491–494.



- Humphrey, W. S. (2008). The software quality challenge. *Cross Talk: The Journal of Defense Systems Engineering*.
- Humphreys, P. (2004). *Extending ourselves: computational science, empiricism, and scientific method*. Oxford University Press.
- IEEE. (2000). *IEEE-STD-1471-2000. Recommended practice for architectural description of software-intensive systems*. <http://standards.ieee.org>. Accessed 10 November 2018.
- ISO/IEC. (2017). 12207:2017. Systems and software engineering — software life cycle processes.
- Jones, T. C. (1978). Measuring programming quality and productivity. *IBM Systems Journal*, 17, 39–63.
- Jones, T. C. (1981). “Program quality and programmer productivity: a survey of the state of the art”. *ASM Lectures*.
- Jones, C. (2008). *Applied software measurement: global analysis of productivity and quality*. 3rd edition. McGraw-Hill.
- Gopal, G. K., (2006). 100 statistical tests. Sage Publishing.
- Malkawi, M. (2014). Empirical data and analysis of defects in operating systems kernels. Proceedings of the 24th IBIMA conference. Milan, Italy, 6–7 November 2014. Available online at [https://www.researchgate.net/publication/281278326\\_Empirical\\_Data\\_and\\_Analysis\\_of\\_Defects\\_in\\_Operating\\_Systems\\_Kernels](https://www.researchgate.net/publication/281278326_Empirical_Data_and_Analysis_of_Defects_in_Operating_Systems_Kernels). Accessed 8 June 2018.
- Mantis. (2019). *MantisBT*. <https://mantisbt.org/>. Accessed 29 January 2019.
- McGarry, J. et al. (2002). *Practical software measurement: objective information for decision makers*. Addison-Wesley.
- Micro Focus. (2019). HP ALM/Quality Center. <https://www.microfocus.com/en-us/products/application-lifecycle-management/overview>. Accessed 29 January 2019.
- Motor Industry Software Reliability Association (MISRA). (2013). *Guidelines for the use of the C language in critical systems*. <https://www.misra.org.uk/>. Accessed 30 June 2018.
- Petricek, T. (2017). Miscomputation in software: learning to live with errors. *The Art, Science, and Engineering of Programming*, Vol. 1, Issue 2, Article 14. <https://arxiv.org/ftp/arxiv/papers/1703/1703.10863.pdf>. Accessed 29 January 2019.
- Phipps, G. (1999). Comparing observed bug and productivity rates for Java and C++. *Journal of Software: Practice and Experience*, 29, 345–358.
- Piccinini, G. (2015). *Physical computation: a mechanistic account*. Oxford.
- Plutora, Inc. (2018). Plutora. <https://www.plutora.com/>. Accessed 29 January 2019.
- Primiero, G. (2014). A taxonomy of errors for information systems. *Minds and Machines*, 24, 249–273.
- Rescher, N. (2009). *Error: (on our predicament when things go wrong)*. University of Pittsburgh Press.
- Royce, W. W. (1970). Managing the development of large software systems: concepts and techniques. *Proceedings, WESCON*, August 29170.
- Stutzke, R. D. (2005). *Estimating software-intensive systems: projects, products, and processes*. Addison Wesley.
- Symons, J. F., & Alvarado, R. (2019). Epistemic entitlements and the practice of computer simulation. *Minds and Machines*, 1–24.
- Symons, J. F., & Horner, J. K. (2014). Software intensive science. *Philosophy and Technology*. <https://doi.org/10.1007/s13347-014-0163>.
- Symons, J. F., & Horner, J. K. (2017). Software error as a limit to inquiry for finite agents: challenges for the post-human scientist. In: Powers T. (eds) *Philosophy and computing*. Philosophical Studies Series, vol 128. (pp. 85–97) Springer.
- Thayer, T. A., Lipow, M., & Nelson, E.C. (1978). *Software reliability: a study of large project reality*. North-Holland.
- Thielen, B.J. (1978). SURTASS code Review Statistics. Hughes-Fullerton IDC 78/1720.1004.
- University of Virginia, Department of Computer Science (2018). *Secure Programming Lint (splint)*, v3.1.2. <http://www.splint.org/>. Accessed 2 July 2018.