Understanding error rates in software engineering: Conceptual, empirical, and experimental approaches

Jack K Horner and John Symons
University of Kansas

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. Here we argue that even in software that is developed in accordance with the best practices of contemporary software engineering, various kinds of error in software occur at the empirically observed average rate of about one error per hundred lines of code. We compare the taxonomies of empirically observed software error rates reported in the published literature with Giuseppe Primiero's (2014) taxonomy of error in information systems. The goal of such comparisons is to discover taxonomies that help to reduce the software error rate in practice.

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 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 also has significant deficiencies. These limitations become especially clear in light of recent conceptual work on error. Though philosophical research into the problem of error in software is in the very early stages, it already exceeds empirical studies in some important respects and offers some hope for clarifying ways in which engineering practice can be improved.

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 2007, 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. More recently, Giuseppe Primiero (2014) has offered a conceptual account of errors in information systems recognizing three general types of errors: mistakes, failures, and slips. Each of these has what Primiero calls *conceptual*, and *material*, subtypes. (Thus, there are six high-level types of error in that taxonomy).

Section Three provides an overview of the results of a wide range of empirical studies of software error published between 1978 and 2018. We follow our review of the literature with an informal categorization of software system errors that comprehends the range of error types that appears in that literature. Next, we compare this informal taxonomy with Primiero's taxonomy of errors in information systems. Finally, we use those results to suggest an experiment that could be used to determine how to conduct pairwise comparisons of error taxonomies. The ultimate goals here are to help reduce the software error rate in practice 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, regardless of application type – *about one to two* *empirically observable errors per hundred lines*[1] *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.

2. Software Systems, Specifications, and Errors.

The empirical and conceptual parts of this paper are restricted to errors in software systems. ISO/IEC 2008 defines a *software system* S to be a collection of processes and artifacts, abstract or concrete, that are essentially or by fiat associated with a sequence of instructions written in computer languages (such as 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, together with the specification, design, test, implementation, and operational processes (including human interactions) and artifacts associated with these activities.

A software specification, H, is a set of imperative sentences that state *what* objectives S must achieve[2], but not *how* S must achieve those objectives. For example, the imperative "By 1969, the system must 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.

[1] 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 related view, see Table 2.53 in Boehm 2000.
[2] 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.

For expository convenience, we posit that H is *not* an element of S. From a software engineering perspective, at a high level S, and thus errors that may arise in S, involve life-cycle process phases and associated products (artifacts) of S. Given H, those life-cycle process are (following ISO/IEC 2008)

- Creating a logical design (LD) of S, allocating allocating elements of H 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
- Deploying S
- Maintaining/sustaining S

Each of these phases typically generates documents that are subjected to review during the lifecycle.[3] Several variants of the lifecycle process are possible (see, for example, Stutzke 2005, esp. Chapter 10), but apart from mandating that S satisfies H, these differences make no difference for the purposes of this paper.

However S is realized, we posit that in order to be correct, S must *satisfy* H; in our view, this satisfaction relation can be characterized in model theoretic terms.[4] If S does not satisfy H, 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

[3] 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 lifecycle. The end result of this allocation process is an acyclic graph of allocations to elements of the developed system. The allocation process typically has high epistemic and contextual content.

[4] Although explicating this relationship in detail is outside the scope of this paper, we believe that the schema of this relation can be captured in the following way. "S satisfies H" iff for each model M of H, there is some model M' of S such that M is homomorphic to M'.

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 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 2008). 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 is limited to tracking the number of errors, with relatively little attention regard to the nature of those errors, as a function of time.[5]

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. Of those studies we chose eleven widely cited studies that involve quantitative data collection, reasonably well-documented methods, a broad range of error types, and sample sizes of twenty or more projects representing a variety of application types. While our sample is not exhaustive, we regard it as representative. To help assess whether our sample is representative, we examined every abstract in *IEEE Transactions on Software Engineering* that mentions software error. If the abstract mentioned software error, we reviewed the associated full paper to assess whether it contained information that should be included in a representative sample of empirical software error-rate literature. We found, based on this method, that our sample was in fact

[5] 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. Although such error tracking systems could be used to inform formal statistical analyses of error-rates, they are typically not optimized to be used this way.

representative.[6] In addition to our survey of the literature, one of us (AU#1) conducted an error analysis of the Linux operating system (excluding the kernel) using the *splint* static analyzer (University of Virginia 2018).

For each of these twelve studies, we recorded the general application type (the domain in which the software is used, whether the software was an application or part of an operating system, the general types of error recorded, the average fraction of lines sampled that contained at least one error, and the standard deviation of the fraction of the errored lines. (We will explain the taxonomy of error types below.) Our findings from the empirical literature, and from our splint analysis of Linux, are summarized in Table 1.

[6] This result is not especially surprising, because at least two of the sources contained in our sample (e.g., Boehm 1981 and Humphreys 2008) themselves contain broad summaries of published empirical software error rates.

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[7]
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, Lipow, and Nelson 1978
Various	Misunderstanding the specification, logic, numerics	0.03 (average over all app. types)	not available	Boehm 1981, p. 383
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
Various	Misunderstanding the specification, logic, numerics	0.02 (average over all app. types)	not available	Humphrey 2008, p. 5[8]
Operating system	Logic, memory	0.005	not available	Malkawi 2014[9]

[7] The error rates stated in Charette 2005 are an order-of-magnitude estimate.
[8] Humphrey 2008 states this data in a high-level summary form only.

kernels	management, task management			
Linux operating system, excluding the kernel	Logic, incorrect type conversion, memory management	0.02	0.01	Unpublished *splint* results obtained by AU#1, 2018)

In our survey, we discovered that the taxonomies used in published empirical software error studies typically varied, and on the surface appeared to be incommensurable, from study to study. On further analysis, however, it became clear to us that the range of error types reported in the empirical software error literature could, with only a slight abstraction,[10] be represented in just a few higher-level types of error. In our review we categorized these higher-level error types as follows: misunderstanding of the specification, logic, numeric, and memory management. This agrees with various taxonomies of empirical software error discussed in Bowen 1980 and Boehm 1981 (p. 383), although it is not strictly identical to any one of them. In the following, we illustrate what the categories in our taxonomy mean.

3.1 Misunderstanding the specification

By far, the kind of error most frequently reported in these studies was "Misunderstanding the specification". 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 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

[9] Malkawi 2014 states that the owners of the data prohibit reporting the identity of the operating system(s) of interest.
[10] Discovering an "appropriate" abstraction not a mechanical activity, and whether a given abstraction A is "better than" another, B, depends on the purpose the abstraction must serve.

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). 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 a relatively common source of 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 2008 for a more comprehensive, and more fine-grained (C-language-oriented) list):

a. The developer doesn't understand, or provide protection against, the numerical accuracy and precision limitations in a numerical algorithm.

b. The developer is not aware of, or doesn't pay attention to, the limitations of numerical type conversions in the 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, in some cases with disastrous results.

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.

b. In a mutually exclusive list of logical cases, the developer fails to manage one or more cases.[11]

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 (locating a location in memory), writing to and reading from memory, and deallocation (releasing previously allocated memory for use by other programs) in order to satisfy speed ("time complexity" (Aho, Hopcroft, and Ullman 1983, Section 1.4)) or space ("space complexity" (Aho, Hopcroft, and Ullman 1983, Chap. 12)) constraints. In some languages or applications, this kind of detail is often 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.

4. Limitations and caveats

Most of the projects referenced in Table 1 were developed under best practicable engineering standards. The error rates in Table 1 are typical of those found in software projects *after* the software became operational. The rough taxonomy we have sketched above certainly does not exhaust the varieties of errors and failures that software projects encounter. Empirical studies of this kind 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

[11] There is straightforward programming technique that helps to mitigate the effects of this kind of error – essentially, it requires including a logical case clause that handles anything that lies outside the cases explicitly enumerated/diagnosed by the program. The technique isn't a panacea, of course. A software engineer must use it religiously, and something reasonable must be done by a program if the program finds itself in a non-enumerated logical state.

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, 3; Stutzke 2005, 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. Many 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.

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 2007, 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.

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.

Developers rarely publish metrics, including error metrics, on their projects (DeMarco 1982). There are many reasons for this. Not least, 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.[12] In addition, 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.

5. Interpreting the results of empirical studies

[12] In a competitive environment, of course, a developer that paid no attention to software error would go out of business.

At least three significant observations can be made about the empirical studies of software error to date. Most strikingly, there is good agreement among average 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.* In addition, these studies suggest that this error rate has not appreciably changed in nearly 40 years of software engineering practice. Note also that standard deviation of the empirical error rates, where reported, are large relative to the average error rates. This suggests that the empirical software error-rate data has high dispersion. Third, the *splint* error measurements (performed by AU#1) on Linux are especially worth noting. 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 assumption is false – 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 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 show some striking features, as we have seen. However, the error types shown in Table 1 may not shed sufficient light on how we might best reduce error rates. It is likely that understanding the kinds of errors that arise, their prevalence, seriousness, and ideally their sources, will allow us to make progress on reducing the prevalence of the errors that we care about. To begin to address this challenge we need a taxonomy of errors that is at least logically complete (within an appropriate universe of discourse). We consider Giuseppe Primiero's 2014 taxonomy of errors in information system as a productive start on such a project.

6. An overview of Primiero's taxonomy of errors in information systems

Primiero's taxonomy of error in information systems depends heavily on a formalized characterization of information systems. To render his account of error clear enough for the purposes of this paper, we must first recount some of that formalism. Primiero's approach begins with the recognition 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. 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).[13] 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, 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, 251).

The *language* of Primiero's characterization of information systems has countably many *types*. *Terms* are objects in types. A *formula* in that language is the expression that an object is of a given type. *Sets of formulas* are also included in the language. *Operations* work on terms to define non-atomic formulas. *States* correspond to models of the language where types are declared valid (Primiero 2014, 251).

More precisely, the *alphabet* of the language of the informational semantics of Primiero's 2014 characterization of information systems is

a. types := {A, B, C, …}

b. processes := {a_1, a_2, a_3, …}

c. data stacks := {Γ_1 , Γ_2 , Γ_2 , …}

d. states := {S_1 , S_2, …}

e. operations := {r_1, r_2, …, r_n}

Types express propositions that are true by terms. A term can be defined by a program accessed and executed at some state; in that case, its type corresponds to the specification declaring validity of a content by that program. In this representation, a specification is based on a functional interpretation of the system. More formally, Primiero defines (2014, 252):

[13] 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.

Let S be a set of states and $S \in$ S. A *specification* α (= *A-valid)* is the content made valid by accessing and using a program of type A at state $S \in$ S.

In a state (Γ_i , α) a process a_i is executed, instantiating a specification α under a set of other states. To every non-terminal state S in Γ_i, a finite set of rules applies to reach content valid at S'. A *model for the system* evaluates every S by transition to some S'.

Let Γ_i be a sequence of processes. A *goal* G is a state S = (Γ_i , α) for a system if α is the intended specification of the system, i.e., the processes in Γ_i are supposed to make the type A in α valid (Primiero 2014, 252).

A *data stack* Γ_i is a (possibly empty) sequence of processes or programs that is accessed and executed from states S_1 to S_i to reach a specification α at state S_{i+1}. (Primiero 2014, 252)

ϕ is a *strategy* if ϕ is a collection of rules or instructions that, given an initial state, are used by the system to reach a goal G by accessing and using the information valid at intermediate states. (Primiero 2014, 252)

A *procedure* is a triple P = <ϕ, S, G>, where ϕ is an operation, S is a set of states, and G is a goal reached via ϕ. (Primiero 2014, 252).

Given this characterization of an information system, Primiero 2014 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." (2014, 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"[14] A. (Primiero 2014, 259). For

[14] 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.

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." (Primiero 2014, 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.

Primiero 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." (Primiero 2014, 260) As with mistakes, Primiero explains how failures have both conceptual material subtypes. *Conceptual failures* involve "[…] problems in the selection and formulation of rules or strategies." (Primiero 2014, 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." (Primiero 2014, 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 Primiero is what he calls *slips*. According to Primiero, slips are "errors generated by the applications of rules that are appropriate to the given goal, but that do not match some formal criteria." (Primiero 2014, 261). Slips have both conceptual and material subtypes.[15] *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" (Primiero 2014, 262). For example, we might try to apply a second-order-valid numerical integration routine that requires a fourth-order-valid integrator.

[15] Primiero infelicitously defines slips as "material errors" (2014, 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.

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

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

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 Primiero's taxonomy (that correspondence, however, is not one-to-one). Second, there are error types in Primiero's taxonomy for which there is no clear counterpart in the empirical software error types of Table 1. That observation implies that the error types in the existing empirical studies that we reviewed are, relative to Primiero's taxonomy, less than complete and fail to capture some important features of software error. In this context at least, philosophical analysis provides a more fine-grained differentiation of error types and dimensions than we find in the existing empirical software error literature. As we noted above, this difference is especially significant in the case of errors in software systems whose consequences can vary by context of use.

7. Discussion

Primiero's formal characterization of information systems can support a much 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 partitionings of the space of error types (in the generalized sense of 'error in an information system' mentioned in the previous sentence).[16] 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 engineerng practitioners.

Whether any particular taxonomy can be used as the basis for a taxonomy of errors that helps to reduce software errors in practice can be decided experimentally. One can imagine a classroom-scale experiment of the following sort that is designed to provide a pairwise comparison of competing error taxonomies. In this experiment each student is assigned to one of three types of teams.

Team Type 1 develops software to a specification, using only the empirical software error types of a given taxonomy as part of its development processes.

Team Type 2 develops software to the same specification, using a competing taxonomy as part of its development process.

[16] 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, 258). Second, Primiero's classification corresponds reasonably well with at least one classification of errors in science (Primiero 2014, 258).

Team Type 3 analyzes the error statistics of the software developed by Teams of Types 1 and 2.

For a sufficiently large sample of Teams of Types 1 and 2, conventional statistical hypothesis testing can be used to compare the error-rates of Teams of Types 1 and 2.

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. 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 limited 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, the insights this kind of analysis of software error has provided already exceed the insights provided by empirical studies of software error in several important respects. These results show how this kind of analysis can help to clarify how engineering practice can be improved, while at the same time grounding software error characterization in an epistemically and logically transparent idiom.

Acknowledgements: We wish to thank…

References

Aho, A. V., Hopcroft, J. E., and 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.

Chang, C. C., and Keisler, H. J. (2012). *Model Theory.* Third Edition. Dover.

Charette, R. N. (2005). Why software fails. IEEE Spectrum

DeMarco, T. (1982). *Controlling Software Projects.* Yourdon Press.

Horner, 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.

ISO/IEC. (2008). *12207:2008. 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.

Linux authors, TBD. (2018). Version TBD.

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. Accessed 8 June 2018.

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.

Phipps G. (1999). Comparing observed bug and productivity rates for Java and C++. *Journal of Software: Practice and Experience* 29, 345-358.

Primiero G. (2014). A taxonomy of errors for information systems. *Minds and Machines* 24, 249-273.

Royce, W. W. (1970). Managing the development of large software systems: concepts and techniques. *Proceedings, WESCON*, August 29170.

University of Virginia, Department of Computer Science (2018). *Secure Programming Lint (splint),* v3.1.2. http://www.splint.org/. Accessed 2 July 2018.

Stutzke, R. D. (2005). *Estimating Software-Intensive Systems: Projects, Products, and Processes.* Addison Wesley.

Symons, J. F., & Horner, J. K. (2014). Software intensive science. *Philosophy and Technology.* Doi:10.1007/s13347-014-0163.

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.

Wikipedia. (2014). The Linux kernel. http://en.wikipedia.org/wiki/Linux_kernel. Accessed 19 September 2018..wikipedia.org/wiki/Linux_kernel.