NIST IR 8101

A Rational Foundation for Software Metrology

David Flater
Paul E. Black
Elizabeth Fong
Raghu Kacker
Vadim Okun
Stephen Wood
D. Richard Kuhn

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8101

National Institute of
Standards and Technology
U.S. Department of Commerce

NIST IR 8101

A Rational Foundation for Software Metrology

David Flater
Paul E. Black
Elizabeth Fong
Vadim Okun
Software and Systems Division
Information Technology Laboratory

Raghu Kacker
Applied and Computational Mathematics Division
Information Technology Laboratory

Stephen Wood
Information Access Division
Information Technology Laboratory

D. Richard Kuhn
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8101

January 2016

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Under Secretary of Commerce for Standards and Technology and Director

A Rational Foundation for Software Metrology

David Flater
Paul E. Black
Elizabeth Fong
Raghu Kacker
Vadim Okun
Stephen Wood
D. Richard Kuhn

January 2016

Contents

A Rational Foundation for Software Metrology

Executive summary

This report is the primary deliverable of a 7-month project that was sponsored in part by the Building the Future program of NIST's Information Technology Laboratory.

Much software research and practice involves ostensible measurements of software, yet little progress has been made on a metrological foundation like the International System of Units (SI) for those measurements since the work of Gray, Hogan, et al. in 1996–2001. Revisiting this topic, we found that many software metrics can be described in a manner that is consistent with the International System of Quantities ("metric system"), and we identified opportunities for improving current practices along those lines. On the other hand, we identified structural obstacles to realizing the vision of software metrology that works like physical metrology for all desired measurands.

However, the biggest hindrance to the development of solid software metrology has been that the most popular and sought-after measurands are in fact qualities rather than quantities, and that the critical difference between the two from a measurement perspective has generally not been acknowledged. We have taken the necessary step of drawing that distinction and moderating expectations for what software measurement really can do. Now, on a foundation that is not compromised by confusion, we are prepared to build measurements that will deliver what they promise.

If this project is continued, the next steps will be:

- Build up and refine the system of quantities;
- Develop methods and tools to support the rational use of well-defined metrics in assessments of software; and
- Collaborate on general metrology issues such as dimensional analysis with counted quantities, generalized models of uncertainty, and characterization of complex systems.

Other future work would include:

- Attempt to reproduce published results about software using more rigorous measurement methods than were originally applied;
- Analyze the definitional uncertainty that affects bug-counting and attempt to make the counts more precise, repeatable, and reproducible;
- Use simple examples of software functions as experimental subjects to explore the limits of measurability; and
- Determine whether there is interest in metrology training that is targeted for computer scientists and software engineers.

A Rational Foundation for Software Metrology

Abstract

Much software research and practice involves ostensible measurements of software, yet little progress has been made on a metrological foundation like the International System of Units (SI) for those measurements since the work of Gray, Hogan, et al. in 1996–2001. Given a physical object, one can determine physical properties using measurement principles and express measured values using standard quantities that have concrete realizations. In contrast, most software metrics are simple counts that are used as indicators of complex, abstract qualities.

In this report we revisit software metrology from two directions: first, top down, to establish a theory of software measurement; second, bottom up, to identify specific purposes for which software measurements are needed, quantifiable properties of software, relevant units, and objects of measurement. Although there are structural obstacles to realizing the vision of software metrology that works like physical metrology for all desired measurands, progress is possible if we start with a rational foundation.

1 Introduction

Metrology is the science and technology of measurement. Weights and measures were the earliest, man-made tools that made it possible to describe magnitudes of physical quantities such as the weight of any solid object, the volume of any type of liquid, and the dimension of any type of material, greatly simplifying the construction of buildings and the development of trade and technology.

Originally, all weights and measures were locally defined. They were sufficient for local trade and technology development, but they did not agree with each other. In the 18th century, advancements in chemistry finally enabled the definition of meaningful standards. However, it was still challenging to select units of measurement that were 1) agreeable to all concerned parties, 2) reproducible when and where needed, and 3) unchanging.

Scientific and engineering advances made global standards possible. They, in turn, made global trade and technology possible. But turning that possibility into reality required an international system of units of measurement that were stable, recognized, and used throughout the world. That system is now known as International System of Units (SI) [2]. Any physical quantity known to science can be stated as a mathematical expression—the product of a numerical value and a unit of measurement. The magnitude of a quantity can be expressed in terms of the seven SI base quantities length (m), mass (kg), time (s), electric current (A), thermodynamic temperature (K), amount of substance (mol), and luminous intensity (cd), either individually or in combinations.

The SI system began in the 19th century and has continued to evolve ever since, growing in response to discoveries such as electricity and optics and changing to incorporate more precise measurements. But a new kind of product was invented in the middle of the 20th century to which the SI system has not yet responded: software.

Software has greatly transformed products, services, organizations, and human societies. Although commerce still is dominated by the exchange of physical goods, that exchange and nearly every other commercial process has been revolutionized by, and has become dependent on, software. As a result, although the economic

This report is the primary deliverable of a 7-month project that was sponsored in part by the Building the Future program of NIST's Information Technology Laboratory. The proposal as funded is reproduced in Appendix A.

In addition to the primary authors, text and comments were contributed by William F. Guthrie, Frederick (Tim) Boland, Irena Bojanova, Jack Boudreaux, Albert Jones, Jeffrey Horlick, Frederic de Vaulx, Don Tobin, and W. Eric Wong. Figure 1 and Figure 2 were drawn by Vreda Pieterse for [1].

benefits of information technology are reckoned in trillions of U.S. dollars [3], mistakes that are attributable to the lack of reliable measurements of software can incur costs on a similar scale.

Since the established discipline of metrology has proven effective in so many applications, it may not be immediately obvious why it would not easily expand to apply in the same way to measurement of software. However, the different space that software works in (binary data processed by digital machines) causes its observable nature to be completely different from that of physical objects existing in the continuous ("analog") space of the physical world for which metrology evolved. For example, suppose that one has a 35 mm photographic slide and a digital scan of that slide. The width of the photographic slide is determined by measuring length on a continuous scale, while the width of the digital scan is determined by counting discrete pixels. The physical process involves a standard reference (the meter), calibration, and uncertainty, while the digital process has none of that—yet are the quantities not analogous?

Many *ad hoc* software metrics have been defined and used. But when neither the methods of established metrology nor any comparable alternative are applied, the outcome is metrics and procedures that do not meet expectations for metrological rigor and results whose meaning and significance are unclear. The production of quantitative indicators by any plausible means is no substitute for validated measurement.

In this report we lay the foundation for a rational approach to software metrology. The scope of the current effort is software as the object of measurement. The software *development process* is the object of measurement in a lot of related work that focuses on forecasting the time and effort required to complete a software development project—these metrics are not within our scope of work. We also will not address software being one component of an instrument intended to measure a physical property [4] or software used to model, simulate, or predict physical properties, sometimes called *virtual measurement* (e.g., computational quantum chemistry)—in both cases, the object of measurement is not the software itself but something else.

The current effort is not the first time that NIST has looked at software metrology [5, 6, 7] or software metrics [8, 9, 10, 11, 12, 13]. In addition, an ongoing project, [Mathematical] Foundations of Measurement Science for Information Systems, has applied a measurement perspective primarily for analysis and modelling of networks [14]. Although some individual metrics have met expectations for measurement quality, software metrology in general remains an immature science. This report is the first step in a renewed effort to address that problem.

The remainder of the report proceeds as follows. Section 2 provides more detailed background on metrology in general, software metrology in particular, and previous NIST surveys of the topic. Section 3 discusses the purposes for which software measurements are used and the ways that they are falling short of expectations. Section 4 explains why physical metrology is taken as a role model for software metrology. Section 5 applies the framework of physical metrology to software measurements to the extent that it is applicable. Section 6 explains why the applicability is limited. Section 7 revisits the issue of software qualities (rather than quantities) in more detail. Finally, Section 8 gives conclusions and the path forward.

2 Background

2.1 Existing vocabularies

Two documents are central to the vocabulary of measurement: the International Vocabulary of Metrology (VIM) [15] and the Guide to the expression of Uncertainty in Measurement (GUM) [16].

The first edition of the VIM was developed by four main international organizations concerned with metrology—the International Bureau of Weights and Measures (BIPM), the International Electrotechnical Commission (IEC), the International Organization for Standardization (ISO), and the International Organization of Legal Metrology (OIML)—and published through ISO in 1984. The second edition of the VIM and the first edition of the GUM were later produced by the ISO Technical Advisory Group on Metrology (TAG 4) based on specific recommendations of the International Committee for Weights and Measures (CIPM). Then in 1997 the Joint Committee for Guides in Metrology (JCGM) was formed to maintain and promote the two documents, a role which continues to the present day.

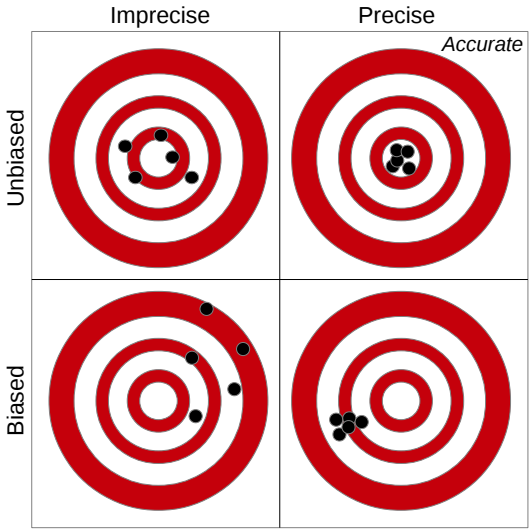

Figure 1: Precision, bias, and accuracy, from [1].

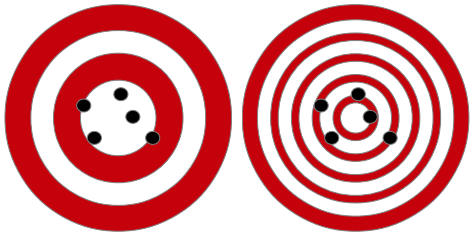

Figure 2: Resolution, from [1].

Although the VIM is the terminology standard, it is complemented by additional definitions and background that are provided in the GUM. The GUM and its supplements codify what could be described as the generally accepted methods for calculating and reporting the uncertainty of measurements. Together, the VIM and GUM provide authoritative definitions for critical metrological concepts such as accuracy, precision, resolution, and bias (see Figure 1 and Figure 2).

Despite this standardization, metrologists may also use vocabulary inherited from longstanding practice in physical science domains such as chemistry and physics, where the historically accepted usage of particular terms differs from the VIM's usage.

The jargon commonly used by computer scientists and software engineers in reference to software measurements differs in some respects from the jargon used in physical metrology. The Wikipedia entry for *software metric* exemplifies the more general use of the word "metric," looser interpretation of what qualifies as a measurement (e.g., not necessarily quantitative), and emphasis on estimation within the development process as the most common measurement goal [17].

The VIM's notion of *quantity* is seldom used by computer scientists and software engineers. The terms *base measure* and *derived measure* may substitute for *base quantity* and *derived quantity* respectively, as in [18]. Also, the meaning of the term *indication* in [18] is quite different from its meaning in the VIM.

ISO/IEC/IEEE 24765 [19] is a very large terminology standard with a broad scope, "applicable to all systems and software engineering work," that includes measurement-related terms. The formal standard is a snapshot of a publicly accessible database known as SEVOCAB (Software and Systems Engineering Vocabulary) [20]. Defined terms include metric (direct and indirect), measure (base, derived, direct, and indirect), measurement, unit of measurement, measurement function, measurement method, measurement

procedure, measurement process, accuracy, repeatability, reproducibility, and measurand. Some definitions were taken directly from the VIM but others were not. The term quantity is used but not defined.

More specialized standards such as ISO/IEC 25010 [21] introduce additional vocabulary.

2.2 Uncertainty

It is the expressed philosophy of the GUM that "all components of uncertainty are of the same nature and are to be treated identically" [16, Appendix E]. The following three subsections describe a contrasting view of uncertainty as something differentiable and categorizable.

2.2.1 Random variation

Sources of uncertainty that are random in nature and which can be modelled as random variables and characterized with statistics are in the category known as *aleatoric uncertainty*. It typically is the main source of variability in repeated measurements of the same quantity under the same conditions and typically is controlled by taking the mean of a sufficiently large number of repeated measurements. The GUM supplies methods for determining what a "sufficiently large number" is for a given error tolerance.

The GUM separates evaluations of uncertainty into "Type A" evaluations based on statistical analyses of series of observations and "Type B" evaluations based on something else, such as consensus or expert judgment. This particular dichotomy has no bearing on the types of uncertainty *per se*, merely on the methods used to assess it (and "it" is almost exclusively aleatoric uncertainty).

2.2.2 Unknowns

Sources of uncertainty attributable to lack of knowledge—to facts that are relevant but not known—are in the category known as *epistemic uncertainty*. Epistemic uncertainty is also said to include the consequences of error in model parameters (such as constants intended to describe physical processes) and of inadequacies of the models themselves, called parameter uncertainty and model uncertainty respectively. The 2008 performance assessment for the Yucca Mountain nuclear waste repository provides a large-scale example of accounting for these kinds of uncertainty as well as an argument that they are not meaningfully distinguishable from one another [22].

Most generally, epistemic uncertainty includes the consequences of everything that the scientist may be ignorant of or wrong about, without bound, including the most fearsome "unknown unknowns." However, if nothing at all is known about the theorized source of error, then any accounting for it will be equivalent to an arbitrarily chosen safety margin. Arbitrary safety margins have their uses, but the GUM explicitly takes the position that uncertainty values should be realistic rather than safe or conservative [16, §E.1].

Although epistemic uncertainty is not a new concept, there remains controversy about the best way to account for it.

2.2.3 Future events

Consider the difference between measuring aircraft reliability in terms of mean time between failures in a long period of observation, and predicting aircraft reliability based on measured quantities, simulations, and analysis of designs before any flights occur.

Forecasting differs from measurement in that one is making statements about future events rather than past and present events. This distinction may be immaterial from the uncertainty perspective: if the impact of not having certain information is the same either way, forecast uncertainty is simply a kind of epistemic uncertainty. However, ignorance of present facts can perhaps be remedied while ignorance of the future cannot.

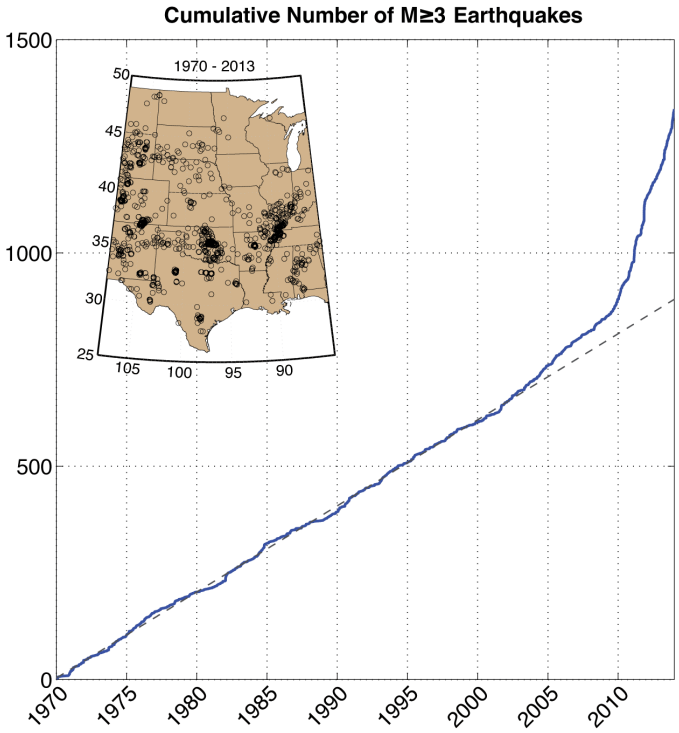

Figure 3: Example for forecast uncertainty taken from [24]: cumulative number of earthquakes with a magnitude (M) of 3.0 or larger in the central and eastern United States, 1970–2013. The linear model (dashed line) fit to 20th century data was accurate and predictive until a certain time, after which error described by structural uncertainty ballooned.

Planning expert Kees van der Heijden distinguishes between uncertainty that can be estimated based on historical data ("risks"), uncertainty that cannot be estimated that way because it results from events that are rare or nonexistent in the historical record ("structural uncertainties"), and uncertainty that results from inconceivable events ("unknowables") [23].

Figure 3 provides an example of structural uncertainty from seismology where a model of U.S. earthquake frequency that was constructed based on the data available in the 20th century could not predict the sudden increase in seismicity that factually occurred in the early 21st century.

Forecast uncertainty is recognized in economic modelling and meteorology.

2.2.4 Recognized limitations of established uncertainty methods

In 1986, Henrion and Fischhoff [25] noted that the best available measured values of c and some other physical constants underwent metrologically incompatible revisions. They suggested that psychological biases cause "persistent overconfidence" in uncertainty assessments. More recently the problem has become known as "dark uncertainty" [26] by analogy with "dark matter"—nobody sees it, but analysis indicates that there is a lot of it around.

As detailed in [27], JCGM guides embody different interpretations of foundational concepts, which lead to different conclusions about the meaning of stated uncertainties.

The current GUM's approach to deriving coverage intervals[1] has been recognized as lacking in generality, but users are apt to make unwarranted assumptions rather than deviate from the given "script" [28, 29].

[1]The features distinguishing "coverage intervals," "credible intervals" and "confidence intervals" are not important within the scope of this report.

2.3 Scales

In 1946, Stanley S. Stevens published an article a mere 4 pages long that defined the following scales for measurement results [30]:

- Nominal scale: supports determination of equivalence but nothing else—a classification scheme with no intrinsic ordering among classes.[2]
- Ordinal scale: additionally supports relative comparisons between values (less than, greater than), but differences of values are not comparable. This list itself forms an ordinal scale: each level supports a superset of the operations of the previous one and is therefore "greater than" by that measure, but differences of scales are not particularly useful.
- Interval scale: additionally supports sums and differences of values, producing well-defined intervals that are comparable to one another, but the ratio of two values is not meaningful (e.g., the Celsius scale of temperature).
- Ratio scale: additionally supports meaningful ratios of values and a zero value that corresponds to a natural or "absolute zero" of the property being quantified (e.g., the Kelvin scale of temperature).

Subsequent writings have elaborated this taxonomy with varying levels of consistency. The two additional scales that we will later reference are

- Dichotomic scale: a nominal scale with only two values.
- Interordinal scale: defines intervals or ranges over an ordinal scale.

Stevens' taxonomy is accurately described as controversial. Velleman and Wilkinson summarized a gamut of previously raised objections [31] and quoted an alternative taxonomy from [32, Ch. 5] as evidence that Stevens' factoring of the problem space is not canonical. Examples that form new scales by adding constraints, translations, transformations, or other functions (e.g., cyclical scales such as angles that are modulus 1 rotation) show that the original, attractively simple taxonomy is merely the top of a slippery slope that leads to more complex theories of data types and mathematical relations (e.g., [33]).

Nevertheless, Stevens' taxonomy or incremental extensions of it are frequently referenced in discussions of metrology. In scoping the definition of measurement, the VIM accepts processes that yield results on an ordinal scale but excludes those that yield results on a nominal scale. Rod White argued that this scope is too conservative because many processes that appear to have the nature of measurement produce nominally-scaled results [34]. However, one could counter that the scope is still too liberal because ordinal quantities do not have well-defined measurement units and therefore cannot be part of a system of quantities with meaningful mathematical relations [15, §1.3]. Results on an ordinal scale do not intrinsically provide any indication of the magnitude of differences in the objective property being quantified, not even in a relative sense. It is impossible to make a rational cost/benefits decision without additional information.

2.4 Existing software metrology

Dozens of books and thousands of articles about software measurement have been published. For this reason we cannot provide an exhaustive review in this report but must instead make generalizations based on a survey.

Similarly, there are many standards that relate to our scope of work. A noteworthy example, ISO/IEC 15939, Systems and Software Engineering—Measurement Process, includes a measurement model and vocabulary but is chiefly concerned with the meta-process (i.e., the process of planning and executing a measurement process, rather than process of measurement or any specific measurements). It relates to our scope of work as an abstract and descriptive outer framework.

The nearest thing to "SI for software" currently existing is in part 13 of the ISO/IEC International System of Quantities (ISQ). IEC 80000-13:2008, Quantities and units—Part 13: Information science and technology,

[2] The term *nominal scale* is oxymoronic: the word *scale* implies a progression, but a nominal scale has no such progression.

focuses on networking and communications (more information technology, less software); however, it does identify some quantities (storage capacity/size, bit rate, clock frequency/rate, information content, and entropy) and reference non-SI units of information (bit/octet/byte and shannon/hartley/nat) that are relevant in the software realm [35].

2.5 Prevalence of counting in software metrics

There is disagreement among metrologists as to whether simple counting is a form of measurement and whether the resulting count, with no SI unit attached, qualifies as a measured quantity value. Counting is unlike any other measurement procedure and usually implies a zero uncertainty (but not always [36]). The VIM makes allowances for counts of "entities" [15, §1.8, Note 4 and §1.10, Note 3]; however, probably this was done only because the SI mole is defined as such a count [2, §2.1.1.6]. In measurements that assign a number of moles to a quantity, counting is not actually involved.

But in software metrology, nearly every quantity that is not actually a physical quantity is determined through counting. In [37, Table A-1], Abran et al. enumerated the "base measures and their measurement units" from parts 2–4 of ISO 9126, a standard for the evaluation of software quality [38]. The units of these base measures are instructive (Table 1): almost 3/4 of the measures are counts of something other than bytes, time, or money.

Table 1: Categorization of ISO 9126 quality metrics by type of units of measurement, derived from [37, Table A-1].

Std part	Total	Time	Money	Bytes	Other count	Weighted score
2 (external)	38	10		2	26	
3 (internal)	34	2		1	31	
4 (in use)	10	4	1		4	1
Total	82	16	1	3	61	1

The weighted score is exemplary of a less obvious type of quantity that is not identical to any specific count but that nevertheless has only counts and constants as input quantities. Although they are more complex, they still are determined through counting.

Non-physical software metrics that are not derived from simple counts are rare, but they do exist; e.g.:

- If there is an oracle for the ideal output of an algorithm, a mathematical function may be used to quantify the distance between the actual output and the ideal output. For example, the performance of a lossy image compression algorithm may be measured by computing the distance between the original image and the result of a compression-decompression cycle.
- The probability of failure for a known fault within a program might depend on the probability distribution of a character string variable over a domain of values that cannot be enumerated.

2.6 Conclusions of previous NIST surveys

Below, we summarize the findings of the previous NIST surveys that are most relevant to the present work.

NIST Interagency Report (NISTIR) 6025, 1997 [5]:

- "[Information Technology (IT)] metrology differs from physical metrology in several ways including; the SI dimensioning system is not as relevant; less analytical methods exist to quantify uncertainty; and the area is relatively new compared to physical metrology." (p. 21)

- "The task group believes that these concepts and the concept of traceability apply to metrology for IT. However, it is important to recognize two aspects which delineate or distinguish IT metrology from physical metrology. First, useful IT quantities are not realizable solely by use of a physical dimensioning system; such as SI. Secondly, existing methods for calculating expressions of uncertainty in physical metrology can not be easily or always applied in IT metrology." (p. 10)
- "Of the seven base units in SI, only the 'second' for time, appears essential for IT metrology. Possibly, the only other base unit necessary for IT metrology is the 'bit' for information." (p. 10)
- Conformance testing was found to be comparable to measurement. Consequently, the vision for uncertainty gravitated toward "level of confidence" and traceability.
- Opportunities were identified in six areas: level of confidence in test results; interoperability testing; automatic generation of test code; IT dimensioning or description system(s); software metrics; and algorithm testing.

Gray 1999 [6]: "There seems to be no identified basis for software measurement which can apply the principals [sic] of physical metrology. ... There is almost nothing relating these measures to other measurement concepts such as uncertainty and traceability."

Hogan et al., 2001 [7]: "After a few decades of research and practice, we appear to have achieved a plateau for the state-of-the-art in software testing.... The physical metrology principles of units, scale, and uncertainty presently have no counterpart in software metrics. ... New methods for measuring and testing IT will be essential to continued growth and development."

NISTIR 7310, 2006 [9] (§2): "The problem of finding a quantitative metric that preserves the ordering of the dimension is a difficult problem in information metrology." "Units and measurements in informational metrology have some characteristics not found in physical metrology. For example: Nonnumeric total orderings... Partial orderings... Relative measures... Boolean measures." "Another factor unique to informational metrology is the relation of information in a system to the entities in the real world to which it refers (*fidelity*). Fidelity is a characteristic not found in physical systems, because physical things are not 'about' anything else, as information is."

NISTIR 7564, 2009 [10] (§3.4–3.5): "Many desired properties such as complexity, usability, and scalability are qualities that can be expressed in general terms, but difficult to define in objective terms that are useful in practice and provide a fully complete picture." "At best, certain properties of software that are assessed are able to capture only some facets of any desired quality." "Quantitative valuations of several security properties may also be weighted and combined to derive a composite value (e.g., rating = .25 * rankingA + .75 * rankingB). Such compositions can, however, yield undesired results." "Security measurements have proven to be much more successful when the target of assessment is small and simple rather than large and complex."

NIST Special Publication (SP) 500-307, 2014 review draft [13]: "Common terminologies (i.e. the definition of measurement, metric, and related concepts) or sets of measurement artifacts (i.e. unit of measurement, metric) often have several definitions, which makes it very difficult for the cloud service customer to compare services or rely on third party tools to monitor the health of the service."

This report adds to the previous work by clarifying the analogies between software and physical metrology, defining the limits of the analogousness and what is measurable, and describing a path forward for rational software metrology.

3 Goals and gaps

In general, measurements can be used to make comparisons, to make decisions, or simply to ascertain facts about an object of measurement (e.g., in hopes of finding an explanation for an anomaly).

Examples of comparisons and decisions that are made with the help of software measurements include:

- Should we ship this software now or test it more?

- Should we harden our existing code base for a new class of vulnerability, or would it be less effort to start from scratch?
- Should we stick with the new process or go back to the old one?
- What quantity of a system resource should we specify as the minimum required for running this software?
- Should we accept the software delivered by the contractor or request more changes?
- Should this software be allowed on the company's systems?
- Which of several competing software packages should we commit to using for a particular purpose for the foreseeable future?
- Is the new version faster or slower than the old version?
- What resolution or bit rate of video can/should we generate?
- Will adding memory, disk space, or CPU/GPU power make this software run better?
- Should we wait on the results, let it run over the weekend, or just give up now?

Examples of the information needed are:

- Size of the code base;
- Complexity and intelligibility of the code base;
- Statistics on faults, failures, and changes to the software;
- Identified deviations from requirements, specifications, standards;
- Counts of specific patterns in the source code;
- Performance measures; and
- Resource consumption measures.

In the absence of a rational foundation for software metrology, a number of pathologies have emerged:

- Lack of consensus on the definitions of measurands leads to a proliferation of realized quantities. For example, while the natural language concept "length" has acquired a consensus definition within the SI system, the natural language concept "size" has many different interpretations for software and correspondingly many non-comparable metrics (e.g., bytes used on disk, bytes used in memory, number of lines of source code, number of lines of source code excluding comments and whitespace, number of source and data files, number of classes, number of functions, or any of several different definitions of function points).
- Quantities that are derived as nontrivial functions of disparate counts, which in VIM terms are different *kinds* of quantities [15, §1.2], wind up being numbers for which the reference (in the sense of units) and/or the scale of the results are unclear—so it is not clear what operations and inferences are valid to perform subsequently with those results.
- Most importantly, a failure to understand and confront the differences between qualities and inherent properties and between software and physical objects of measurement has led to overconfidence in and overreliance on results when the methods of physical metrology are copied in a software context.

These pathologies are not unique to software. Comparable problems have arisen in physical disciplines when the putative measurand was an abstract attribute or quality such as health or safety, rather than an inherent physical property such as length or mass:

- For human health generally or healthy weight specifically, there is a proliferation of realized quantities, such as Body-Mass Index (BMI), that each provide a limited and imperfect view. BMI can be measured well enough, being a quantity derived from height and weight, but the metric has no value except as a rough indicator (or worse, an unreliable predictor) of health.
- For vehicle safety, both compliance with safety mandates and crash test results are informative, but they cannot account for design errors and novel collision risks that are unknown until after the fact.
- For financial risk, market prices are not bound by any physical laws, and the market as a whole is not bound to behave rationally or consistently; modelling them as if they were bound by laws of physics or statistics leads to overconfidence in and overreliance on predictions.

Measuring non-summative properties of large, complex systems with many moving parts and interactions is inherently difficult, regardless whether those parts are software components or mechanical components, but abstract properties are furthermore subject to the definitional uncertainty that comes with not having a concrete realization to serve as a common reference.

4 Appeal of physical metrology framework

Quoting from [39]: "Physics has long been regarded as the model of what a science should be. It has been the hope, and the expectation, that if sufficient time, resources, and talent were put into the sciences concerned with other phenomena... [that] the kind of deep, broad and precise knowledge that had been attained in the physical sciences could be attained there too."

Causal relationships that are widely believed to exist between real properties of the objects of measurement and measured quantity values are expressed in the form of physical laws which are strongly supported by empirical evidence. If a set of analogous laws about nontrivial properties of software existed, we too could have "widely accepted measurements traceable to solid principles, rigorously defined quantities, and base and derived units" (quoted from our own proposal in Appendix A). Conversely, the discovery of such laws might just depend on first making the right measurements of the right properties.

Some feel that the primary difference between software measurement and physical measurement is the maturity of the measurement discipline. To the extent that software measurement is analogous to physical measurement, applying the established methods ought to shortcut the centuries-long process of refinement that evolved the capability to perform reliable physical measurements. But if the analogies are not close enough and the established methods are not applicable, then it is just a case of "physics envy" [39] and the best approach to measurement will be found elsewhere.

5 Applying the physical metrology framework to software measurements

5.1 Vocabulary

Repeatability vs. reproducibility: Repeatability is the ability to replicate the measurement result for a given quantity immediately, while reproducibility is the ability for another person, on another occasion and/or with different equipment, to replicate the measurement result for a given quantity under the stated conditions. (This is a paraphrasing and simplification of the VIM definitions, which are longer but more precise in their wording.)

Measurand vs. realized quantity: The measurand is the "quantity intended to be measured" [15, 2.3] while the realized quantity is the quantity that is *actually* measured [16, Annex D].

In Section 5.2 we will introduce related concepts that distinguish measurable quantities from abstractions.

The following concepts are needed in this report but no equivalent definitions were found in existing vocabularies.

Instantaneous vs. cumulative quantity: An instantaneous quantity provides a value for a specific point in time. A cumulative quantity instead provides a value for a specific time *span*. For example, the speed of a moving vehicle is an instantaneous quantity but the distance travelled is cumulative.

As with mathematical derivatives, an instantaneous quantity is practically approximated using the ratio of two continuous quantities for a short time span. Conversely, the maximum, mean, or sum of a number of samples of an instantaneous quantity can be taken to derive a cumulative quantity for a specified time span.

5.2 Abatement of philosophical quagmires

The following controversies are hereby disposed of for the purposes of our work.

1. **Is simple counting a form of measurement?** A well-defined count that is repeatable and reproducible is sufficient for our purposes. It matters not whether it is called measurement. The challenge is in ensuring that the count is well-defined, repeatable, and reproducible. A simple concept like "software bug" can have hidden ambiguity that interferes with counting [40].
2. **Is testing a form of measurement?** A well-defined test that is repeatable and reproducible, that yields a result on a well-defined scale, is sufficient for our purposes. It matters not whether it is called measurement. The challenge is in ensuring that the test is well-defined, repeatable, and reproducible, and that the scale of the result is well-defined.
3. **Is verification a form of measurement?** In simple cases where the property being verified is general and can be regarded as a derived quantity, verification through deductive logic can be considered to be a measurement method for that quantity. However, the verification of complex software properties, such as whether or not a specific algorithm will terminate, is comparable to the verification of complex physical properties, such as whether or not a particular rocket design can reach orbit. The answer depends not just on measured base quantities but on a great deal of context, some of which is specific to each application, and the process becomes mechanical only after experts have disposed of the context. Essentially, this quagmire reduces to a pre-existing quagmire inherited from physical science: where is the boundary between data summarization and analysis?
4. **Can probability be measured?** Given sufficient information and an accurate model, accurate probabilities can be calculated. It matters not whether it is called measurement. The challenge is in obtaining an accurate probabilistic model of software behavior.
5. **What is the best model of uncertainty?** No one is certain. Software measurements are in the same boat as other measurements while the theory and practices of uncertainty characterization continue to evolve. The challenge is to make a reasoned, defensible decision on how to model uncertainty in each case, given the known tradeoffs between different methods.
6. **How can software metrics be made completely general so that they will not be obsoleted by changing programming languages, paradigms, and technology?** It is true that revolutionary new approaches to software are always being invented. However, it is also true that metrics that were applicable in the time of first-generation programming languages remain applicable to the vast majority of software today. If the realm of possibility is not compromised by pragmatism, the quest for complete generality up front leads to analysis paralysis. We choose to be pragmatic, understanding that metrology will need to evolve to remain relevant.
7. **Can qualities and -ilities be measured?** Not generally. The distinction between quantities and qualities is critical. Measurement deals with quantities which are well-defined and that have no connotations beyond a statement of objective fact about the current reality. Quantities exist and can be measured without any judgment as to whether particular values would be good or bad. Qualities, on the other hand, carry open-ended connotations of intangible goodness or fitness for a particular purpose which are not inherent to the object of measurement and which cannot be calibrated to a standard reference without losing generality. If a quality metric is defined and tuned to make it a valid predictor in one context, it may well be invalid or uninformative when replicated unchanged in other contexts. When the definition of the measurand needs to be changed for each use, the process is not measurement.

Figure 4 illustrates the quantity-quality continuum with examples from software measurement. Measurable quantities are the foundation; they either are well-defined expressions of real, measurable properties of objects or are mathematically derived from one or more such quantities. Attributes (as defined here) are abstractions of inherent properties of the object of measurement; they are measurable insofar as a particular quantity or function of quantities is chosen to realize them. But then there are qualities, which are more nebulous, high-level abstractions whose relationships to measurable quantities change based on context. Methods for predicting qualities based on measurable quantities are the subject of conjecture.

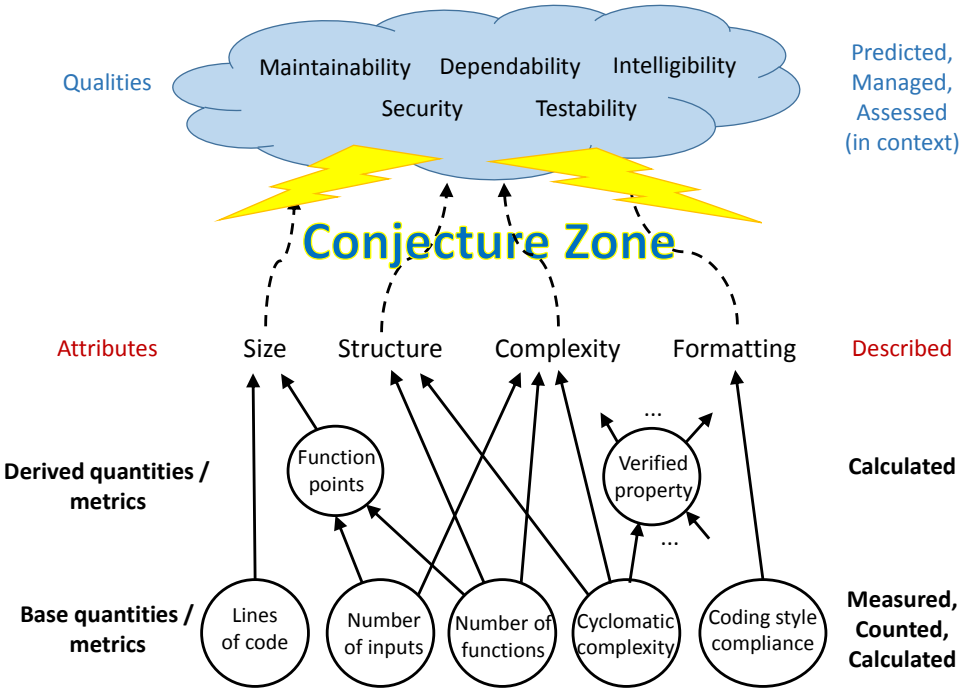

Figure 4: Relating qualities to quantities.

When asked to quantify high-level -ilities with extrinsic dependencies, one has a stark choice to make. Either the -ilities must be defined as nothing more than what can be measured (e.g., the definition of quality as conformance to explicit, testable requirements), or the process of quantifying them must be something other than measurement. With the latter, one acquires the burden of proving that the resulting quantities still deliver the desired qualities of reproducibility, comparability, etc.

The "original sin" of software metrology is to treat qualities as inherent properties of the objects of measurement.

5.3 Objects of measurement

The word "software" covers a lot of territory, either directly or indirectly. A notional software attribute can be different things depending on what exact artifact or process serves as the object of measurement. Therefore, we must identify these objects of measurement to clearly define the quantities being measured.

Table 2 shows an initial breakdown into distinct objects of measurement. Although a measurement of any of these objects can tell us something about software, a reference to *the software* generally means the range from implementation to execution.

We distinguish the *executable*, which is a static artifact, from an *execution* of that executable, which is a process, event, or phenomenon. However, the static artifact in principle specifies the population of possible (or permissible) executions, so statements about that population essentially are statements about the executable.

Executable test cases can be viewed either as features of the applications that they test or as applications in their own right that have specialized metrics. It is also an implementation choice whether to integrate testing with the application, as in the built-in self-test (BIST) concept, or to keep them strictly separate.

The measurements and attributes can be classified as dynamic (measurements or attributes of running software) or static (everything else). While static objects have many easy-to-measure properties, the most useful results are those that quantify how software behaves as a dynamic object, when it runs. Predicting the latter from the former is seldom easy; the task is similar to predicting macroscopic properties of physical

Table 2: Objects of software measurement.

Object	Examples of attributes
Architecture, design	Patterns, conceptual integrity, coupling, cohesion
Requirements	Self-consistency, size
Specification	Self-consistency, consistency with requirements, similarity
Algorithm	Computational complexity, efficiency, correctness, accuracy
Implementation (source code or script)	Size, efficiency, stability, portability, correctness, consistency with specification
Executable (binary or bytecode)	Size, reliability, portability, usability, correctness of compilation
Execution (of binary, bytecode, or script)	Time, energy, power, space (memory, disk, etc.), throughput, correctness of interpretation, correctness of observed behavior, consistency with specification, accuracy of output
Test	Coverage, sensitivity, specificity, size

objects by working upward from the atomic particle level. Contextual differences between in-house and end-user environments further militate against accurate prediction.

5.4 Units

5.4.1 Successes

Software metrics that are merely counts, proportions, or statistics yield dimensionless quantities where the measurement unit is simply 1:

- Lines of code (LOC, a simple count)
- Consortium for IT Software Quality (CISQ) Quality Characteristic scores (counts of non-conformities) [41, 42]
- Events (e.g., number of function calls counted at run time)
- Combinatorial coverage (proportion of combinations tested) [12]
- Cyclomatic complexity (counting linearly independent paths) [43]
- Compression ratio
- CPU utilization

Though it violates [44, §7.5], the type of thing being counted is sometimes used informally as a unit of measure (e.g., LOC).

Physical measurements of software execution that yield quantities of time, frequency, energy, or power should, in principle, use the appropriate SI units. However, the processing unit cycle (CPU, GPU, or other) often appears as both the quantum of time and the base unit for reckoning time within a computer system or subsystem. This occurs because the processor's clock is, in fact, the time reference for the (sub)system, and because the conversion of processor cycles to seconds is not a fixed ratio but can be determined or approximated only in specific circumstances.

Non-physical measurements of software may use these additional base units, which are not part of SI but are in use within the software field:

- Bit (symbol b), a unit of information that is commonly used to indicate the capacity to store, transmit, or receive a given quantity of information[3,4]
- Pixel (unofficial symbol px), a unit of length or area for images[5]

[3]The definition of 'b' as the symbol for bit was made in IEEE 100-2000 [45] and thence indirectly by IEEE 1541-2002 [46] but IEC 80000-13:2008 does not use it [35]. Although bytes replace bits in most contexts, network bandwidth is nearly always quoted in terms of bits per second.

[4]The 'b' symbol is also used for the barn, a non-SI unit of area.

[5]Etymologically derived from *picture element*, pixels are the elementary constituents of raster graphics images. One can trivially measure height, width, or area by counting pixels.

The bit is referenced as a unit in IEC 80000-13:2008 [35]; the pixel is not. Pixels arise when specifying or measuring graphical properties of software, such as the screen resolution that it uses and amounts or rates of graphical input or output.

Examples of derived quantities and units:

- Byte (1 B = 8 b)[6,7]
- Data bandwidth (1/s, informally b/s, often written as Mbps or Gbps)[8]
- Entropy (1 Sh = 1 b; 1 Hart = $\log_2 10$ b; 1 nat = $\frac{1}{\ln 2}$ b)
- Display density (1/m, informally px/m)
- Defect density (bugs/LOC—again, this usage is improper according to [44, §7.5])

The first three are used in [35].

When a well-defined metric is applied in conjunction with a transformation of the software or its call graph, such as with module design complexity and essential complexity in [47], it is possible for the definition of the measurand to become obscure. Arbitrary transformations, like arbitrary functions of quantities, must be used carefully in order to preserve an explainable relationship to properties of the object of measurement.

5.4.2 Challenges

Though bug counting is widely practiced and arguably "successful" as a measurement technique, there is no consensus definition of "bug" that is tight enough to ensure reproducible counts [40].

Software measures that are defined as nontrivial functions of disparate counts may yield results whose units and/or scales are not well-defined:

- Halstead measures [48]
- Automated Function Points (AFP) [49][9]

In addition, some existing common practices introduce inconsistencies in the expression of units:

- It is still generally accepted to use SI prefixes that are defined on powers of $10^3 = 1000$ to mean either the original definition or an alternative one based on powers of $2^{10} = 1024$. The set of alternate prefixes for binary multiples defined in [35] and related standards has gained acceptance only in technical contexts.
- Dots per inch (dpi) is a printer's quantity that was misappropriated to mean pixels per inch in displays.
- Similarly, the printer's point, which is a physical unit of length, was misappropriated to indicate the size of a bitmapped font. At this time the rule-of-thumb conversion uses a ratio of 4 pixels to 3 points, implying an assumed linear pixel density of approximately 37.8 cm^{-1} (informally, a display density of 96 px/in).

[6] [35] cautions that the byte was historically comprised of variable numbers of bits but that it now is assumed to mean an octet.

[7] The 'B' symbol is also used for the bel, a non-SI unit of logarithmic ratio quantities. [35] states that its use to mean the byte is English language-specific and not international.

[8] Although recognizing the bit as a unit is seemingly the first step for software metrology, NIST policy requires that quantities be stated in SI units, with any corresponding values expressed in non-SI units following parenthetically. If a number of bits is regarded as a dimensionless quantity then [44, §7.5] also applies.

[9] The function point is notionally a unit of measurement, but its definition in [49] is just that which is (notionally) counted by the complex process that is used to assign a value.

5.5 Quantities and attributes

5.5.1 Architecture or design as object

Cohesion and coupling are very abstract concepts that are believed to be indicative of the quality of software architecture or design. An assortment of realized quantities and qualitative ordinal scales have been defined for cohesion and coupling [50, 51, 52].

Conceptual integrity is an intuitive quality of good architecture and design that was made famous by Frederick P. Brooks Jr. [53]. Although it is widely understood, few attempts have been made to quantify it. A recent example [54] involved making architectural decisions, such as the decision to instantiate a particular design pattern [55], explicit using an ontology, and then computing metrics from the ontology.

Nowadays there is great emphasis on instantiating good design patterns and avoiding instantiating bad ones (known as antipatterns [56]). There is at least one tool that uses counts of antipattern instances as inputs to calculate a "Quality Deficit Index," though it appears to derive the counts from source code (by detecting "code smells") rather than from an architectural or design model [57].

A source to watch is the International Workshop on Software Architecture [and] Metrics (SAM), which was first held in 2014.

5.5.2 Requirements as object

A set of requirements can be analyzed to determine whether it is self-consistent. Inconsistent requirements will contain a logical contradiction.

There exist several standardized processes for functional size measurement that operate on functional user requirements and yield results that are described as a count of function points. Currently popular examples were defined by the International Function Point Users Group (IFPUG) [58] and the Common Software Measurement International Consortium (COSMIC) [59, 60].

5.5.3 Specification as object

A specification can be analyzed to determine whether it is self-consistent or consistent with requirements.

Several measures have been proposed for application to various aspects of formal specifications. Feature models for software product lines (SPLs) are typically specified as feature models, sets of features that are present or absent, out of a large collection of potential features. Thus if an SPL has n features, 2^n possible products can be generated by including various subsets of features with a base product. To provide adequate testing of SPLs it is important to measure the similarity among different products, since the potential product space is enormous for feature sets that may number in the hundreds. Originally developed for botany, the Jaccard distance [61] is being used for SPL similarity measurement [62, 63].

5.5.4 Algorithm as object

If there is an oracle for the ideal output of an algorithm, the correctness or accuracy of the algorithm can be quantified as an average or maximum distance between the algorithm's output and the ideal output using an appropriate distance function (e.g., root-mean-square error).

The computational complexity of CPU-intensive algorithms is expressed in terms of the number of operations performed as a function of the input parameters. Efficiency can be derived as the theoretical minimum number of operations required to complete a task divided by the number of operations actually performed by the algorithm.

The hardness of cryptographic algorithms is reduced to the computational complexity of recovering plaintext without the encryption key(s).

Performance properties of scheduling algorithms and networking protocols (throughput, response time, latency, fairness) can be quantified for average or worst-case scenarios.

5.5.5 Implementation (source code or script) as object

Source code metrics are used for estimating programming effort, complexity of code understanding and maintainability, and other qualities. While some of the following metrics can be computed from lower-level artifacts such as bytecode, the focus is on source code. Metrics can be classified based on the aspect of source code being measured.

Size

Size is commonly used for normalizing other metrics; for example, defect density is defect count divided by code size.

Code size

There are several ways to measure source code size, including:

- Lines of code (LOC) and its variants, such as non-blank, non-comment lines of code;
- Statements or code blocks;
- Lexical tokens; for example, Halstead measures [48] include vocabulary (number of unique operators and operands) and length (total occurrences of operators and operands).

Design size

Design size can be measured by counting design elements in the source code; for example:

- Procedures;
- Size of procedure interfaces in terms of the number of arguments;
- Elements of object-oriented design, such as classes, interfaces, packages, and design patterns.

Functional size

Functional size of software, that is, the amount of functionality in the software product, can be measured from source code. One example is Automated Function Points (AFP) [49] which was based on IFPUG function points [58].

Most of the above measures are counts on a ratio scale. However, as explained in [64], traditional function points are a combination of various quantities with different scales.

Structure

Structure can affect qualities of the software. To measure structural attributes, software is often modelled as graphs. Many metrics for code structure have been proposed [65]. Here we classify the structure metrics based on the execution and data details. There are other ways of classifying structure, for example, based on the level of abstraction.

Syntax elements

Syntax elements can be measured without regard to execution order.

- Counts on call graphs: calling modules (fan-in), modules called (fan-out);
- Loops;
- Switch statements without default case;
- "Sites" (bug "opportunities")—code locations with characteristics relevant to bug classes. For example, a location with array or pointer access may have a buffer overflow.

Control flow elements

Control flow elements are usually counted on the control flow graph.

- Cyclomatic complexity [43];

- Depth of nesting;
- Edges or branches;
- Paths.

Data structure and flow elements

Composition and transfer of data impact various aspects of software quality. This category includes metrics that involve both data and control flow.

- Information flow measures [66];
- Variable definition-use paths;
- Code slices;
- Coupling, the degree of interdependence between modules [67];
- Cohesion, the extent to which the individual components of a module are needed to perform the same task.

Not all metrics can be neatly assigned to a particular category. For example, coupling generally involves data flow; hence it appears in the data structure and flow category above. However, number of calling modules and number of modules called (a particular form of coupling) are computed without regard to data or control flow; hence they appear in the syntax category.

Context-related measures

While size and structure are internal attributes of source code, the following take into account the external purpose.

Satisfaction of requirements and specifications

Requirements and specifications may be thought of as imposing constraints. A verifier attempts to prove that the constraints are or are not satisfied.

As described in Section 5.2, in order to be compatible with a measurement process, the constraints must be well-defined and the verification must be repeatable and reproducible.

Bugs (correctness)

While a software bug (sometimes called a fault or weakness) is manifested in code, it may have been introduced earlier, during specification or design.

Bug counts have several deficiencies. First, they do not consider the differences in consequences or the difficulty of exploitation for malicious purposes. Second, they are often ambiguous [40]. Finally, it is very difficult to enumerate all bugs in any reasonably large piece of software. In particular, static analysis tools, which can be thought of as measuring devices for bugs, have false positives and false negatives.

Despite the above limitations, bug counts are commonly used in practice. Though the impact of bugs depends on context, the number of bugs in the software is considered an indicator of general quality [68].

Portability

The portability of implementations is measured by enumerating the platforms on which they work. Since new platforms appear all of the time, portability can only be measured in the context of some selected set of platforms.

Other measures

Efficiency

Implementing an algorithm usually introduces some platform-specific overhead that is irrelevant from the point of view of the algorithm in the abstract. For example, a step in an algorithm might be to swap the values of two variables, but to implement that step on a given platform might require a minimum of three operations on data. Efficiency can be derived as the theoretical minimum number of operations required to implement an algorithm divided by the number of operations actually used. The theoretical minimum can be scoped within-platform or across-platform depending on which comparisons are desired; e.g., some other

platform might implement a single operation to swap two values, but that only matters if we are willing to change platforms for the sake of efficiency.

Instead of using operation counts, implementation efficiency can be derived in terms of the size of the implementation, the size of the executable produced from it, execution time, resource consumption, etc.—quantities that are obtained by measuring the executable or executions thereof. Theoretical minima for these quantities might be difficult to determine, but relative measures are obtainable without determining the minima.

Stability

Stability (or instability) is the rate of change of the implementation; e.g., the number of lines of code, functions, or source code files that are added, changed, or deleted within a specified time span.

5.5.6 Executable as object

Trivially, the size of executables in storage is measured in bytes, and the portability of executables is measured by enumerating the environments (e.g., which versions of which operating systems on which hardware platforms) in which they will run.

Compilers can either introduce or remove logical faults in the process of compiling source code. Assessment of the security and correctness of software-as-deployed therefore often requires direct examination of the executables.

Expected or observed properties of all possible executions of a given executable may be summarized as properties of the executable itself. For example, while the amount of memory that a program allocated during a given execution is a property of the execution, an upper bound on the amount of memory that a program *ever* allocates when it is executed is a property of the executable. In this usage, the executable represents the infinite population of executions while individual executions are samples from that population. Properties of the infinite population might be determined through static analysis of the executable or they might be estimated from a sample of executions.

General statements about the reliability or usability of an executable similarly characterize the frequency and distribution of failures or usability problems over the population of possible executions. Use cases, scenarios, and scripts for reproducing problems are all ways of identifying a subpopulation of executions that exhibit a problem.

5.5.7 Execution as object

Using the definitions in Section 5.1, metrics for running programs can be instantaneous or cumulative. An instantaneous metric describes a property of a running program at a specific point in time. A cumulative metric describes the program's execution over a specified time span. The practically useful spans include from the start of execution to the current time, from the start of execution to the end of execution, and from the current time minus a specified interval to the current time (e.g. as done on a rolling basis by the Unix top command or the Windows task manager).

Sometimes the executables used for dynamic measurements are specially instrumented to improve observability. These instrumented versions are surrogates for the executables that would be used in practice. While the impact of the measurement process upon the measurand must always be considered (including the famous "heisenbug" uncertainty [69]), substituting one executable for another introduces another risk factor that can alter or invalidate results.

Times

Many different kinds of time are measurable for a software execution, including:

- Elapsed time, a physical measurement;
- CPU/GPU time, the amount of time during which the program was actually running on a processing unit;

- Self and total time, which are function-level metrics produced by application profiling tools;
- Latency, the delay between a stimulus and a response (also known as response time).

All of these are cumulative except for latency. Latency is measurable only when a stimulus-response event pair actually occurs. However, as with other instantaneous metrics, the maximum, mean, or sum of a number of latency samples can be taken to create a cumulative metric for a specified time span.

Units: seconds or CPU/GPU cycles.

Discrete resource consumption metrics

Any quantifiable unit of system resources, be it a hardware resource or a software resource, has a corresponding instantaneous metric which indicates how many of the resource units the program is using at a given time. That instantaneous metric can be converted to a cumulative metric by taking the maximum, mean, or sum of some number of instantaneous samples taken during the specified time span.

If resource units are non-shareable then the metric is a simple count. If resource units are shareable then there are different ways of accounting or pro-rating the usage which result in non-integral values.

Units: 1 (either a simple count or a percentage of the resource units available) or bits (in the special case of memory or storage).

Examples: memory utilization, number of CPU or GPU cores utilized, number of file descriptors used.

Event count metrics

Any detectable event pertaining to a program execution, be it a periodic event such as a timer interrupt or a non-periodic event such as entering a blocked state, has a corresponding cumulative metric which indicates how many times the event occurred during the specified time span.

The perf application profiling framework can report many such metrics that are derived from hardware counters in the CPU, software counters in the kernel, or arbitrarily defined tracepoints [70].

Units: 1 (simple count).

Examples: instructions executed, interrupts, invocations of a specified function or system call, cache misses.

Bandwidth/throughput metrics

Any resource whose availability is parameterized by time can be measured instantaneously (as a rate) or cumulatively (the sum total for a specified time span).

Units: 1 or bits for cumulative, 1/s (informally b/s) for instantaneous.

Examples: CPU or GPU utilization, network transmission rate or throughput, bus data rate or throughput, storage I/O rate or volume.

Energy and power

These are physical measurements. Power is the instantaneous metric; energy is the cumulative metric. The energy or power consumption attributable to a particular program execution is related to elapsed time, CPU/GPU time, and CPU/GPU utilization, but the relationship is indirect and complicated.

Units: for energy, J (joule) or the non-SI kWh (kilowatt hour); for power, W (watt).

Correctness and accuracy

If there is an oracle for the ideal output of an execution, a mathematical function may be used to quantify the distance between the observed output and the ideal output. However, if such an oracle does not exist or the output is not simple enough to apply a distance function to, then we have the more general problem of determining whether the observed behavior is consistent with requirements and specifications, which reduces to conformance testing.

Correctness of interpretation (was the software executed correctly by the machine?) can be distinguished from correctness of the observed behavior (was the output correct for the given input?). For example, a faulty implementation of floating-point division in a particular version of a CPU will cause incorrect interpretation

of certain mathematical operations, which may or may not lead to detectable changes in the output of any given program [71].

Units: 1 (e.g., mean squared error), bits, pixels, or a dichotomic scale (i.e., either correct or incorrect).

5.5.8 Test as object

Test case coverage is expressed in terms of branch coverage, path coverage, combinatorial coverage, etc. [12, 72, 73]

The notions of sensitivity and specificity can be applied to software tests, but only if there is a clear statement of what a given test is attempting to demonstrate. The problem is that it is seldom possible to test only one requirement at a time. It is neither unusual nor undesirable for a test to trip on a different kind of bug than what it was designed to find.

If we just say that the test suite as a whole is attempting to detect bugs in general, then the sensitivity of the test suite as a whole is estimated using test case coverage, and specificity less than 1 (false positives) is usually treated as a fault in the test suite.

Various measures of the similarity or distance between different test cases have been used in attempts to select small test suites that nevertheless have much diversity of tests [74].

Individual test cases can be sized generically like any other software, or they can be sized in terms of the number of assertions, requirements, or input combinations that they test. Test suites in turn can be sized by the number of test cases that they contain or by the number of assertions, requirements, or input combinations that they test. One can then derive coverage metrics such as the number of requirements tested divided by the total number of requirements in the specification of the software.

5.6 Opportunities for improvement

5.6.1 Methodology

Criticism of methodological inadequacies has been going on for decades with no obvious improvement in the median level of scientific rigor that is applied in practice [75, 76, 77, 78, 79]. The cited examples are all within the scope of performance measurement, which is one of the easier cases to reconcile with established metrology, and thus all the more troubling.

A great deal of improvement could be made if practitioners in the software industry consulted with experts in established metrology and experimental design. Unfortunately, this requires a cultural change and a dialog between separate specialties that barely speak the same language, in an industry that is notoriously fixated on short-term deliverables.

Improved methods may gain traction if NIST or a NIST grantee or collaborator can present industry with ready-made solutions that are perceived as pure profit. This traction then may incrementally provide the platform through which the underlying science can finally get an audience.

Nevertheless, where there is misunderstanding and equivocation about what quantities are being measured or what measurement is, it will be necessary to overcome existing misunderstandings. An outreach and education program modelled on the Statistical Engineering Division's training courses and workshops might be successful once its relevance has been proven to the intended audience.

5.6.2 Clarifying units, scales, and kinds of quantities

In the case of software metrics that yield results whose units and/or scales are not well-defined, it would obviously be good to refine or replace them with equally useful ones that do not have that problem. For example, the IFPUG and AFP function point standards do not provide a realization of the function point as a unit of measure in the way that SI quantities can be realized. Their function points do not exist as distinct features to be simply counted. The reported quantities are calculated as nontrivial functions of counts of various different features of the requirements or the source code, but no particular feature suffices as a canonical reference for what a function point *is*. This metrological issue is one of the motivations given for the competing COSMIC standard [59, 60]. The COSMIC function point is currently defined as the "size" of one data movement.

If it is construed as a direct measure of complexity, cyclomatic complexity can be criticized for having neither a defined unit nor an obvious scale: is a function with cyclomatic complexity of 20 exactly twice as complex as a function with cyclomatic complexity of 10, or what? But if cyclomatic complexity is regarded simply as the count of linearly independent paths, there is no problem. The problem is introduced by confusing the attribute of complexity with one particular measurable quantity that only partially describes it.

The VIM concept of *kinds* of quantities, though acknowledged to be "to some extent arbitrary" [15, §1.2], should be applied as part of a principled approach to derived quantities. With most of the quantities having dimension 1, differentiation of kinds is necessary to provide a rational basis for rejecting comparisons and combinations that yield ill-defined results. Alternatively or in conjunction, a different approach to units for counted quantities could be applied [80].

5.6.3 Distinguishing realized quantities from measurands

The dissatisfaction and confusion surrounding many software quantities is illuminated by the following two quotations found via [81]:

> Many of the attributes we wish to study do not have generally agreed methods of measurement. To overcome the lack of a measure for an attribute, some factor which can be measured is used instead. This alternate measure is presumed to be related to the actual attribute with which the study is concerned. These alternate measures are called surrogate measures. [82]

> It is usual that once a scale of measurement is established for a quality, the concept of the quality is altered to coincide with the scale of measurement. The danger is that the adoption in science of a well defined and restricted meaning for a quality... may deprive us of useful insight which the common natural language use of the word gives us. [83]

When this process goes awry, we are left with an instance of the streetlight effect, also known as the lamppost problem: we are measuring a given realized quantity not because it is what we were looking for, but because it is what we know how to quantify (it is where the light is).

Being mathematical in nature, both the input quantities and the result (output quantity) of a measurement function [16, §4.1] have the qualities of precision, resolution, dispersion, kurtosis,[10] and drift, and the references that specify their units and scales. The measurand is objective in nature, yet going down the previous list of mathematical qualities elicits corresponding objective qualities (Table 3).

The relationship between the measurand and the realized quantity can also be qualified. The following ordinal scale expands upon the established concepts of direct and indirect measurements to provide more information on the closeness of the relationship:

1. The two quantities are identical;
2. The difference between the two is bounded or otherwise predictable;
3. The difference between the two is unbounded and unpredictable but has low kurtosis;

[10] Kurtosis is a statistical indicator of the shape of a distribution. For a distribution with a given variance, low kurtosis indicates that extreme values are relatively unlikely to appear. The variance therefore is attributable primarily to a well-behaved cluster around the mean rather than to outliers.

Table 3: Objective qualities corresponding to mathematical qualities.

Numerical quality	Corresponding quality of a measurand
Precision	Physical limit of measurability, such as quantum uncertainty
Resolution	Continuous or quantized in nature
Dispersion	Variability
Kurtosis	Instability, bad behavior
Drift	Rate of change
Reference	Definition of the quantity

4. The two correlate or covary in the average case but can diverge wildly;
5. There is no demonstrable relationship;
6. The two are not even comparable.

In the first four cases, the difference between the measurand and realized quantity can be refined to a ratio scale and included in the uncertainty budget to enable more realistic assessments of the validity and uncertainty of software measurements. In contrast, the last two cases include measurements that cannot be validated, where the condition that the measurement process must yield values that can "reasonably be attributed to the measurand" [15, Introduction] is not satisfied. Any software measurements that fall under these latter two cases represent problems to be solved.

5.6.4 Validation

Like most software, software measuring instruments such as application profiling tools are both complex and fragile. The level of validation that is required to achieve trustworthy results is on par with the most fault-prone of physical measurements. Validation in practice usually falls short of that. Although software instruments have the advantage of being easily reproducible by a large user population, this is not necessarily enough to mitigate the risk of faults impacting the measurement at hand. Relatively simple validations performed in preparation for software performance measurements at NIST have discovered previously unknown faults as well as unexpectedly large impacts from known faults that would have ruined the measurements [84][85, §3.7].

Software metrics are also subject to validation. [38, §A.2.1] lists the desirable properties of software quality metrics as reliability, repeatability, reproducibility, availability, indicativeness, correctness (subdivided into objectivity, impartiality, and sufficient precision), and meaningfulness. [38, §A.2.2] then provides criteria for demonstrating the validity of metrics. Below, we have reinterpreted those criteria in a more general context.

- Correlation: in repeated use on different objects, the measured quantity value should exhibit a consistent correlational relationship to the measurand.
- Tracking: a change in the measurand should always result in the measured quantity value moving in the same direction.
- Consistency: measured quantity values should always preserve a relative ordering by the measurand.
- Predictability: for metrics that are used in forecasting, prediction error should always be within the specified range or tolerance limit.
- Discriminative: high and low values of the true quantity should be distinguishable from the measured quantity values.

Of course, there is a catch-22 in statements involving the measurand when no oracle exists for its true value. These criteria nevertheless can be used to describe validity problems when they come to light; e.g., if a software application that complies with a set of requirements is demonstrably inferior to a noncompliant example with regards to the quality that the requirements were intended to ensure, then compliance as a metric for that quality fails the consistency criterion.

5.6.5 Uncertainty

Software measurements usually do not state uncertainties. This presents an opportunity for improvement.

If ordinary aleatoric uncertainty has been ignored for a software metric that has a random component (e.g., execution time or reliability in the face of arbitrary scheduling of concurrent operations), this is a methodological failing that must be remedied. If it is mathematically derived from simple counts then there may be no aleatoric uncertainty to address. However, even making that assumption, there are several other categories of uncertainty that may apply. Epistemic uncertainty attributable to ignorance of present facts about the software, such as the presence of undiscovered faults, incorporates the "code uncertainty" that was described by Kennedy and O'Hagan [86]:

> The output of the computer code given any particular configuration of inputs is in practice not known until we actually run it with those inputs. *In principle*, we could say that it is not really unknown because the output is a known function of the inputs. After all, there is a mathematical model which, at least implicitly, defines that function. Nevertheless, *in practice* the relationship is so complex that it needed to be implemented in a computer code (which may even take hours to run), and it is not realistic to say that the output is known for given inputs before we actually run the code and see that output. It may not be practical to run the code to observe the output for every input configuration of interest, in which case uncertainty about code output needs to be acknowledged.

The other categories include:

- Uncertainty characterizing a definitional difference between the realized quantity and the measurand, as discussed in Section 5.6.3;
- Definitional uncertainty affecting the measurand itself, such as uncertainty whether a software fault should be counted as one bug or two;
- Uncertainty of numerical and logical origin; e.g., the consequences of using data types with limited ranges and precisions;
- Uncertainty attributable to prediction of future events, which may diverge from the best current models (also known as forecast uncertainty or model uncertainty).

As Bock et al. noted in [9], software measurements utilize a wide variety of scales. The use of a particular scale for a measurement value not only restricts the operations that can meaningfully be applied to the estimate, it also entails a particular scale for expressing the uncertainty. Table 4 shows some corresponding scales. It is not a typo that both interval and ratio scales have uncertainty on a ratio scale: the uncertainty of results on an interval scale has a natural zero, unlike the interval scale itself, so it is on a ratio scale.

It is important to note that a probability distribution (either discrete or continuous, as applicable) would subsume the indicated expression of uncertainty in every case.

Table 4: Scales for uncertainty corresponding to recognized scales of measurement values. y, y_{low}, y_{high}, and u are as used in the GUM.

Scale of estimate	Scale of uncertainty
Dichotomic	Level of confidence
Nominal	Nominal coverage set {A, B, C}
Ordinal	Interordinal ($y_{\text{low}} \leq y \leq y_{\text{high}}$)
Interval	Ratio ($y \pm u$)
Ratio	Ratio ($y \pm u$)

5.6.6 Reproducibility

When measurement results are published, it is not enough to provide a measured quantity value, and it still is not enough to provide a value with uncertainty. One must provide information that is sufficient to enable someone else to reproduce that result, sometimes called the measurement context [87]. Reproducibility is especially problematic for measurements characterizing the performance or behavior of software ("works for me"), but even static source code analysis results can be affected by environmental factors such as variations in system-level header files.

5.6.7 Generality

Software metrics that are defined in terms of source code structures are vulnerable to obsolescence as programming languages and even programming paradigms change; e.g., from procedural to non-procedural languages, or from text-based source to a more interactive representation. Even the venerable lines of code measure becomes meaningless when programming is accomplished by assembling software components like toy blocks. But as we noted earlier, attempting to plan for every conceivable paradigm change up front leads to analysis paralysis. The need to evolve is a certainty; the need to evolve faster than is practical, merely a risk.

5.7 Identifying failures

In principle, something like a failure mode, effects, and criticality analysis (FMECA) [88, Ch. 13] could be used to catalog and evaluate the ways in which a measurement can go wrong. The following quick list is not necessarily complete.

1. Basic assumptions of metrology unsatisfied.
   a. The property is not observable.
   b. The quantity is not well-defined or what you want to measure is not a quantity at all.
   c. The quantity is not objective in nature or extrinsic dependencies (measurement context) are uncontrolled.
   d. There is no reference (in the VIM sense that the magnitude of a quantity must be expressible as a number and a reference).
   e. There is no calibration (e.g., when a quantity is derived from many inputs and the weighting of the inputs is an unknown variable).
   f. Not demonstrated that the process produces "values that can reasonably be attributed to the measurand" (a phrase that appears repeatedly in the VIM and GUM).
   g. Repeatability and reproducibility not achieved or not achievable.
   h. A significant source of uncertainty is unaccounted for.
2. Quantitatively unfit for purpose.
   a. Inadequate accuracy (due to bias, imprecision, or both).
   b. Inadequate resolution.
   c. Scale of results or uncertainty is insufficient to support conclusions.
   d. Results from different applications are not comparable.
3. Invalid application.
   a. Invalid surrogacy (no demonstrated relationship between the realized quantity and the measurand).
   b. Experimental design issues (e.g., sample too small or biased).
   c. Abuse of scale (e.g., treating ordinal as interval or ratio).

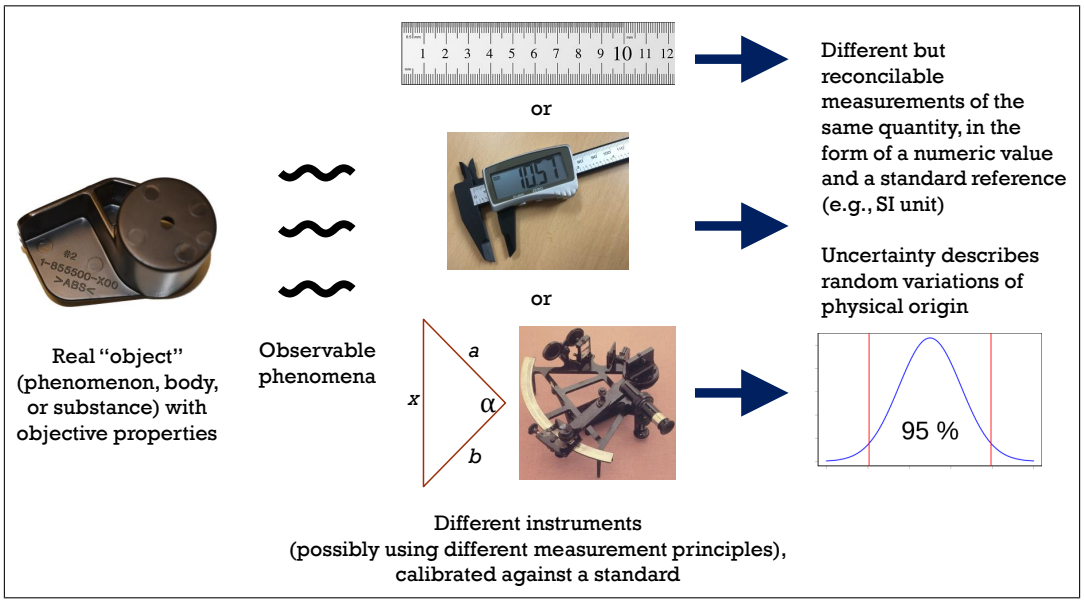

Figure 5: The VIM embodies a realist world view.

6 Limits of analogousness

6.1 Nature of the objects

The scientific tradition that begat modern metrology is focused on the discovery of facts about the physical world. Established metrology consequently is built on a foundation of philosophical realism: we take it as given that there exists an objective reality separate and apart from our perceptions and our instruments, and metrology is all about optimizing our capability to discover objective facts about that reality. The validity of measurements hinges upon the establishment of a chain of causation or correlation that relates the objective properties that are ostensibly being measured to the final outputs of the measurements.

A process that is consistent with that world view (illustrated in Figure 5) could be said to have the nature of a measurement, from the realist point of view. A process that lacks some aspect of that world view—lacking a clear object, say, or an identifiable phenomenon within the chain of causation—might be deemed a non-measurement on that basis. Any metrological problems that arose with such a process could then be dismissed with "Of course it does not behave like a measurement; it is *not a measurement*."

The dissonance between established metrology and software metrology arises in part simply because software is not a physical object and software metrology cannot conform to the same world view without surrendering all claim to the word measurement. The question going forward is how much of the nature of measurement can be preserved or recovered when the objects of measurement exist in cyberspace or conceptual space instead of physical space. All measurements of objects in cyberspace are necessarily indirect.

Boris Beizer, known authority on software testing, rejected the notion that quality assurance for software could involve a measurement process comparable to what is used for quality assurance of manufactured goods. He wrote, "Software products are not physical—they're mental. We don't have mental micrometers for mental constructs. There is no such thing as an 'ideal' program or 'the' correct program." [89, §2.1]

6.2 Nature of the measurands

Although attributes such as size, structure, complexity, and formatting are abstractions of inherent software properties that do not depend on context, they still are conceptual attributes rather than physical ones, and there is no unique right answer of which quantity to use to realize them in a measurement of software.

For attributes in the physical world, consensus has been convergent; but for attributes of software, there has been a continuous churn of divergent definitions. In the physical world, the size of any object can be expressed using the meter without generating controversy; for software, not even international standards can end the debate over how to quantify size.

Some JCGM documents do not preclude the possibility of a measurand having multiple true values[11] due to different precisifications of the measurand's definition. However, those JCGM documents nevertheless require that the true values must be so close together that the difference is below the noise [27]. This may be the case when deciding between an optical or a mechanical definition of physical length [90], but the competing definitions of software attributes do not satisfy that assumption!

Rationality may compel us to divide the multiply-defined attributes; e.g., to eliminate the general notion of software size and to instead discuss functional size and source code size as different attributes—and then to subdivide again according to the incompatible metrics that have been proposed for each of these. However, this takes us ever farther away from the ideal of an SI-like system of quantities that everyone can use.

6.3 Nature of the uncertainty

In a typical physical measurement, the physical system is believed to be well-understood, and the dominant component of uncertainty is the random (aleatoric) uncertainty that arises naturally in physical systems. In contrast, the complexity of software and computer systems guarantees our ignorance of important facts about them—any non-physical measurements of software that are more ambitious than counting (e.g., lines of code), proportions (e.g., combinatorial coverage), or statistics have significant epistemic uncertainty—but the aleatoric uncertainty can be zero.

Although it is expected that the confidence intervals for physical measurements will not always capture the true or ideal values, physical metrology is trusted to produce results that, if not strictly correct, are close enough to the true or ideal values to avoid catastrophes in the real world. Software, on the other hand, is notorious for having essentially infinite sensitivity to obscure and unknown factors. If a function that had reliably returned values between 0 and 255 in literally millions of previous trials suddenly returned a value of 2 147 483 647 for no obvious reason, no software developer would be astonished. Numerical overflow, which occurs in software but not in mathematics generally, is one mechanism that permits large errors to be more likely than errors that are relatively close to zero.

6.4 Nature of the space

Reference [5] observed, "One advantage in IT metrology appears to be that, whatever base and derived units are used, the technological challenge posed in realizing SI units does not exist. In other words anyone can define and establish a 'bit' of information without use of a measurement device."

The relative ease with which the bit is realized is a consequence of it being the unit of quantization for cyberspace. Cyberspace as we know it today was deliberately engineered to quantize into bits, so to "realize" the bit in this space is tautological. Experimental physics has confirmed no analogous property of physical space thus far [91].

We purposely build digital devices to have discrete, instead of continuous, states, to have undiminishing effect at a distance, instead of inverse square law, and to have positive feedback or gain, where energy is waiting to amplify small effects. These three effects lead the digital world, and the software that often drives it, to behave very differently from typical physical phenomena.

To push a strained analogy even farther, the absence of useful quantities expressed in b^2 or b^3 suggests that cyberspace is 1-dimensional, like the tape on a Turing machine, but this may be a matter of perspective. For digital images, both lengths and areas are stated in pixels, but that does not mean that there is no difference between length and area. Quantization is what makes it possible for a 2-dimensional image to be

[11] The concept of "true value" is alternately deprecated and defended in metrology literature. As of yet we have not found a compelling need to avoid using the concept.

represented by a string of bits; achieving a 2-dimensional perspective requires only cutting the string of bits into separate rows.

Quantized space of a specific size has a finite number of possible states, so in principle, it should be possible to cover the input space of a software function. In practice, exhaustive testing of software is feasible only when the input space is very small (somewhere between 32 and 64 bits).

6.5 Relationship to SI

The definition of the mole refers to "elementary entities" [2, §2.1.1.6]. The bit is satisfactory as an elementary entity used to indicate the cyberspace analog of "amount of substance," which is data. Simultaneously, the bit is the reference for binary "length" or "size;" however, if the object of measurement is contiguous, there is no meaningful distinction between these concepts—they all are simply the count of bits.

Some have argued that the mole should be removed from SI on the grounds that simple counts of arbitrary entities have no need of a designated unit [92]. If, on the other hand, a count of molecules or other non-atomic entities has merit as a base unit, then perhaps a count of enumerable features of a piece of software does as well. Such potential unit definitions would proliferate.

The CPU cycle is a quantum of time that competes with the SI unit. The conversion between CPU cycles and seconds is not a fixed ratio, but can be determined or approximated in specific circumstances. Similarly, the storage of bits takes up varying amounts of physical space, but the latter quantity has become irrelevant for most applications as bit density has escalated.

The pixel is an interfacing concept that connects graphical constructs manipulated by software to a physical manifestation that has height, width, area, 2-dimensional location, and color. An expression of the latter in terms of existing SI units would be obscure. On the software side, although both lengths and areas in digital images are expressed as counts of pixels, the most consistent approach is to consider the pixel to be an elementary entity which, like the bit, is used as the reference for more than one kind of quantity (including both length and area).

Addition of the bit and pixel as SI units would mean that bandwidths expressed in terms of b/s or px/s would comply with [44, §7.5]. However, this still excludes many other bandwidths of the form x-per-second, such as FLOPS (floating-point operations per second) or vertices per second (a legacy metric for GPU performance). The quantity "12 gigaflops" would need to be rewritten as "a floating point operation rate of 12 GHz." Relevant discussion of the treatment of dimensionless units in SI is ongoing [80, 93, 94].

7 On qualities

7.1 Distinguishing complex quantities from qualities

A complex quantity or composite metric is one where the composition is known. Given a finite set of input quantities, one can define a measurement equation that takes n inputs and yields a value for the composite metric. It might be difficult to calibrate and it might not be useful, but it is feasible to define. A physical example would be a weighted score based on different crash test results for passenger cars; a software example would be a weighted score based on static source code attributes and dynamic test results.

But qualities are abstract properties where the composition is unknowable. Vehicle safety is one: no matter how much information is known, there could be something unexpected and untested that completely changes the outlook. General software "quality" is another: no matter how many cases are tested, a user could do something unexpected and untested with the software that ends badly. Security is another: it is no more possible to design a software system that is immune to attack than it is to design a fortress that cannot be breached or an airplane that cannot be shot down.

Using a composite metric as a putative measurement of a quality leads to trouble when there is no recognition of the open-ended uncertainty that distinguishes the two. Quality is an open universe, and one cannot cover

a space by sampling at points. The assumption that the sampled points are representative of all points within some region of the space (i.e., the use case) needs some justification, and the more complex the system is and the broader the use case is the more difficult it is to justify.

7.2 General "quality"

Much work has gone into developing metrics for software quality, but it is erroneous to think of software quality as a single, intrinsic property. Software quality as generally understood by users is an amalgam of disparate concerns such as reliability, usability, performance, stability, longevity, lack of onerous dependencies, doing what the user expects, and not creating work or problems for the user. Software quality as generally understood by developers also includes concerns such as source code complexity, maintainability, fault density, and compliance with requirements. These concerns present separate measurement problems of widely varying types, from trivial counting to complex questions that are as difficult as security. (Many people in fact consider security to be an aspect of software quality, while some believe the reverse.)

Reference [41] realizes a quantity based on automated source code measures. This approach can provide only a partial view of the complex just described above. Fitness-for-use nevertheless arises in the debate over the correct weighting of different input quantities [41, §5.1]. There is no weighting that is fit for all purposes.

Identifying quality as an unsolved problem brings up discussion of the methods of Philip Crosby [95], W. Edwards Deming [96], and Joseph Juran [97], and their applicability is worth examining.

- Can a Zero Defects program succeed in removing faults and vulnerabilities from software by managing programmers toward not adding them in the first place? Although due diligence by programmers is necessary and requires supporting management practices, many programming mistakes follow from cognitive limitations. If the programmer fails to see the possibility that triggers a failure, staring at the source code for longer will not make the fault more visible. "Doing it right the first time" in software is rarely witnessed; serious attempts to achieve it are the demesne of formal methods and regimented processes that few software-developing organizations have found to be cost-effective except in regulated industries where they are mandatory.
- Can quality and security be reduced to sets of verifiable requirements that are necessary and sufficient? When an unforeseen attack vector compromises the security of a software application, it is easy to say in hindsight that the requirements were incomplete, but this trivializes the difficulty of writing a set of verifiable requirements that not only produces the desired behaviors but adequately inhibits undesired, unforeseen ones. A recent example was the successful attack against a System Management Mode firmware template that was described as "verifiably correct and secure from any normal attack vector" [98] (although, we have not located an actual verification of it).
- The economics of software quality are different from those of durable goods. Firstly, it is difficult for customers to sue companies for selling them faulty software, so the Price of Nonconformance (PONC) is often borne by the customers rather than the company. Secondly, if the company does decide to correct a fault, no cumbersome and expensive product recall needs to occur; software patching is routine, inexpensive, and heavily automated. Despite the vast amount of research that has been done into how to produce high-quality software, it is far from obvious whether production quality is at all relevant to the success of software products.

All of the above may be considered old-fashioned in the context of software development, which has its own set of continually evolving management methods, such as the many variants of Agile [99]. Given the values expressed in the Agile manifesto, acceptance test-driven development [100] is a logical approach to quality; the standard is "working software," not zero defects. This still carries the challenge of reducing quality to a set of testable requirements, but with a highly iterative process in which those requirements are constantly revised.

7.3 Security

We begin with the good news:

- If the object of measurement is a cryptographic algorithm, the "hardness" of the algorithm can be quantified in terms of CPU/GPU time required to overcome it using the most efficient *known* attacks. (This is not the same as the security of an implementation, which may introduce flaws or provide ways of circumventing the encryption.)
- More generally, the "hardness" of any defense can, in principle, be quantified in terms of the resources required to overcome it. The resources could be anything from time to a number of unknown zero-day vulnerabilities (as in [101]).
- One can also quantify the maximum or expected loss that would result from the total compromise of a given software application, in terms of time, money, and information.
- A lower limit on the vulnerabilities of software applications can be established in retrospect as vulnerabilities are discovered.

Now, the bad news. The concept of software security as practically applied, where the object of measurement is an actual software application, follows not from the theoretical hardness of the software's design in the absence of unknown attacks but from whether there will be attacks that are practical and worthwhile to carry out within the lifetime of the software. Of the existence of such attacks one can have knowledge, but of their non-existence one can have only degrees of belief. The model that suggests itself in this context is not measurement of an observable property, but rather a quantification of defensible belief using Bayesian, Dempster-Shafer [102], or other similar theories. The relationship between this kind of metric and an *assurance case*, which also focuses on evidential reasoning, was described in [103].

A quantification of defensible belief in the security of a software application would be derived from all *measurable quantities* that we find to be relevant; e.g.:

- Existence of known vulnerabilities;
- Security track record of previous versions of the application and of the people who develop and maintain it;[12]
- Reputation of the application and the people who develop and maintain it;
- Traceability or strength of the chain of custody;
- Roles and privileges required to install or operate the software;
- Theoretical hardness of implemented defenses;
- Number of attack vectors;
- Static code quality metrics and adherence to fixed rules, such as the CISQ Security Measure [41, §4.5][42]; and
- Results of testing.

Other relevant quantities are unknowable except when the news is bad:

- The adversaries' knowledge of existing vulnerabilities (unless they are seen to exploit them);
- The probability of being vulnerable or of being attacked (unless it is demonstrated to be 1); and
- The actual hardness of implemented defenses (unless demonstrated to be not hard enough).

The uncertainty of security metrics constructed using this approach would be entirely epistemic. They would need to be validated with respect to their predictive value *for a particular context*, as different factors will be more or less important in different uses.

Having described that approach, however, it must be acknowledged that security is conventionally considered to have at least three primary aspects—confidentiality, integrity, and availability—that have tradeoffs among them. Reducing security to a single score may provide desired simplicity for a go-no go decision, but a truly

[12] As with reliability growth models, a "trivial security prediction model" that simply assumes that future performance will be the same as past performance has some predictive accuracy [104].

informative description of security requires multiple quantities—the inputs to the decision function, not its output.

To be meaningful, security for any system, software or physical, must be defined according to some specification of the protection that it will provide in a particular threat environment. Assumptions must be made regarding the capabilities and ingenuity of the adversary and the pace of future technological progress, all of which are unknown.

8 Conclusion

Although there are structural obstacles to realizing the vision of software metrology that works like physical metrology for all desired measurands, this project did succeed in establishing a rational foundation for improving software measurement:

- We identified analogies between physical and software metrology and opportunities for improving the latter;
- We identified the limits of the analogousness and concrete reasons why software metrology is different;
- We clarified the relationship between software quantities and SI; and
- We distinguished qualities and attributes from quantities.

The biggest hindrance to the development of solid software metrology has been that the most popular and sought-after measurands are in fact qualities rather than quantities, and that the critical difference between the two from a measurement perspective has generally not been acknowledged. We have taken the necessary step of drawing that distinction and moderating expectations for what software measurement really can do. Now, on a foundation that is not compromised by confusion, we are prepared to build measurements that will deliver what they promise.

If this project is continued, the next steps will be:

- Build up and refine the system of quantities;
- Develop methods and tools to support the rational use of well-defined metrics in assessments of software; and
- Collaborate on general metrology issues such as dimensional analysis with counted quantities, generalized models of uncertainty, and characterization of complex systems.

Other future work would include:

- Attempt to reproduce published results about software using more rigorous measurement methods than were originally applied;
- Analyze the definitional uncertainty that affects bug-counting and attempt to make the counts more precise, repeatable, and reproducible;
- Use simple examples of software functions as experimental subjects to explore the limits of measurability; and
- Determine whether there is interest in metrology training that is targeted for computer scientists and software engineers.

Acknowledgments

This work was sponsored in part by the Building the Future program of NIST's Information Technology Laboratory. The proposal as funded is reproduced in Appendix A.

In addition to the primary authors, text and comments were contributed by William F. Guthrie, Frederick (Tim) Boland, Irena Bojanova, Jack Boudreaux, Albert Jones, Jeffrey Horlick, Frederic de Vaulx, Don Tobin, and W. Eric Wong. Figure 1 and Figure 2 were drawn by Vreda Pieterse for [1].

References

[1] Vreda Pieterse and David Flater. The ghost in the machine: don't let it haunt your software performance measurements. NIST Technical Note 1830, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, April 2014. https://doi.org/10.6028/NIST.TN.1830.

[2] BIPM. *The International System of Units (SI)*, 8th edition, 2006. http://www.bipm.org/en/publications/si-brochure/.

[3] Robert D. Atkinson and Andrew S. McKay. Digital prosperity: Understanding the economic benefits of the information technology revolution. Report, Information Technology & Innovation Foundation (ITIF), March 2007. http://www.itif.org/files/digital_prosperity.pdf.

[4] Theodore H. Hopp. Computational metrology. *Manufacturing Review*, 6(4):295–304, December 1993.

[5] MEL/ITL Task Group on Metrology for Information Technology (IT). Metrology for Information Technology [IT]. NISTIR 6025, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, 1997. http://www.nist.gov/manuscript-publication-search.cfm?pub_id=900816.

[6] Martha M. Gray. Applicability of metrology to Information Technology. *Journal of Research of the National Institute of Standards and Technology*, 104(6):567–578, November 1999. https://doi.org/10.6028/jres.104.035.

[7] Michael D. Hogan et al. Information Technology measurement and testing activities at NIST. *Journal of Research of the National Institute of Standards and Technology*, 106(1):341–370, January 2001. https://doi.org/10.6028/jres.106.013.

[8] W. J. Salamon and D. R. Wallace. Quality characteristics and metrics for reusable software (preliminary report). NISTIR 5459, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, May 1994.

[9] Conrad Bock, Michael Gruninger, Don Libes, Joshua Lubell, and Eswaran Subrahmanian. Evaluating reasoning systems. NISTIR 7310, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, May 2006.

[10] Wayne Jansen. Directions in security metrics research. NISTIR 7564, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, April 2009.

[11] Tim Boland, Charline Cleraux, and Elizabeth Fong. Toward a preliminary framework for assessing the trustworthiness of software. NISTIR 7755, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, September 2010.

[12] D. Richard Kuhn, Itzel Dominguez Mendoza, Raghu N. Kacker, and Yu Lei. Combinatorial coverage measurement concepts and applications. In *Second International Workshop on Combinatorial Testing, Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361, March 2013. https://doi.org/10.1109/ICSTW.2013.77.

[13] NIST Cloud Computing Reference Architecture and Taxonomy Working Group. Cloud computing service metrics description. NIST Special Publication 500-307, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, 2014. To appear; draft available at http://www.nist.gov/itl/cloud/upload/RATAX-CloudServiceMetricsDescription-DRAFT-20141111.pdf.

[14] Foundations of Measurement Science for Information Systems, in Applied and Computational Mathematics Division yearly reports, 2007–2013. http://www.nist.gov/itl/math/.

[15] Joint Committee for Guides in Metrology. *International vocabulary of metrology—Basic and general concepts and associated terms (VIM)*, 3rd edition. JCGM 200:2012, http://www.bipm.org/en/publications/guides/vim.html.

[16] Joint Committee for Guides in Metrology. *Evaluation of measurement data—Guide to the expression of uncertainty in measurement*. JCGM 100:2008, http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf.

[17] Wikipedia. Software metric, May 2015. https://en.wikipedia.org/w/index.php?title=Software_metric&oldid=659015498.

[18] ISO. *Systems and software engineering—Measurement process*. ISO/IEC 15939:2007, http://www.iso.org/.

[19] ISO. *Systems and software engineering—Vocabulary*. ISO/IEC/IEEE 24765:2010, http://www.iso.org/.

[20] ISO/IEC, IEEE, and PMI. SEVOCAB: Software and systems engineering vocabulary. http://www.computer.org/sevocab.

[21] ISO. *Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*. ISO/IEC 25010:2011, http://www.iso.org/.

[22] John C. Helton, Clifford W. Hansen, and Cédric J. Sallaberry. Model uncertainty in performance assessment for the proposed Yucca Mountain repository for high-level radioactive waste, December 2009. http://energy.sandia.gov/wp/wp-content/gallery/uploads/Model-Uncertainty_12-2-09.pdf.

[23] Kees van der Heijden. *Scenarios: The Art of Strategic Conversation*. John Wiley & Sons, 2nd edition, 2005.

[24] U.S. Geological Survey. Induced earthquakes, 2015. http://earthquake.usgs.gov/research/induced/.

[25] Max Henrion and Baruch Fischhoff. Assessing uncertainty in physical constants. *American Journal of Physics*, 54(9):791–798, September 1986. https://doi.org/10.1119/1.14447.

[26] Michael Thompson and Stephen L. R. Ellison. Dark uncertainty. *Accreditation and Quality Assurance*, 16(10):483–487, October 2011. https://doi.org/10.1007/s00769-011-0803-0.

[27] Raghu N. Kacker. Probability distributions and coverage probability in GUM, JCGM documents, and statistical inference. *Measurement*, 65:61–70, April 2015. https://doi.org/10.1016/j.measurement.2014.12.056.

[28] Walter Bich, Maurice G. Cox, René Dybkaer, Clemens Elster, W. Tyler Estler, Brynn Hibbert, Hidetaka Imai, Willem Kool, Carine Michotte, Lars Nielsen, Leslie Pendrill, Steve Sidney, Adriaan M. H. van der Veen, and Wolfgang Wöger. Revision of the 'Guide to the expression of Uncertainty in Measurement'. *Metrologia*, 49(6):702–705, 2012. https://doi.org/10.1088/0026-1394/49/6/702.

[29] Walter Bich. Revision of the 'Guide to the expression of Uncertainty in Measurement'. Why and how. *Metrologia*, 51(4):S155–S158, 2014. https://doi.org/10.1088/0026-1394/51/4/S155.

[30] Stanley S. Stevens. On the theory of scales of measurement. *Science*, 103(2684):677–680, June 1946. https://doi.org/10.1126/science.103.2684.677.

[31] Paul F. Velleman and Leland Wilkinson. Nominal, ordinal, interval, and ratio typologies are misleading. *The American Statistician*, 47(1):65–72, February 1993.

[32] Frederick Mosteller and John W. Tukey. *Data Analysis and Regression*. Addison-Wesley, 1977.

[33] R. Duncan Luce, David H. Krantz, Patrick Suppes, and Amos Tversky. *Foundations of Measurement*, volume 3: Representation, Axiomatization, and Invariance, chapter 20: Scale Types. Dover Publications, 2007.

[34] Rod White. The meaning of measurement in metrology. *Accreditation and Quality Assurance*, 16(1):31–41, August 2010. https://doi.org/10.1007/s00769-010-0698-1.

[35] IEC. *Quantities and units—Part 13: Information science and technology*, 1.0 edition. IEC 80000-13:2008, http://www.iec.ch/.

[36] S. Pommé, R. Fitzgerald, and J. Keightley. Uncertainty of nuclear counting. *Metrologia*, 52(3):S3–S17, 2015. https://doi.org/10.1088/0026-1394/52/3/S3.

[37] Alain Abran, Rafa E. Al-Qutaish, Jean-Marc Desharnais, and Naji Habra. ISO-based models to measure software product quality. In B. Ravi Kumar Jain, editor, *Software quality measurement: Concepts and approaches*, chapter 5. ICFAI Books, 2008. http://publicationslist.org/data/a.abran/ref-2273/1096.pdf.

[38] ISO. *Software engineering—Product quality*. ISO/IEC TR 9126-2:2003, -3:2003, and -4:2004, http://www.iso.org/.

[39] Richard R. Nelson. Physics envy: Get over it. *Issues in Science and Technology*, 31(3):71–78, spring 2015.

[40] Paul E. Black. Counting bugs is harder than you think. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 1–9, 2011. https://doi.org/10.1109/SCAM.2011.24.

[41] CISQ Technical Work Groups. CISQ specifications for automated quality characteristic measures. Technical Report CISQ-TR-2012-01, Consortium for IT Software Quality, 2012. https://goo.gl/wezWdM.

[42] Consortium for IT Software Quality. Automated quality characteristic measures, 2016. http://it-cisq.org/standards/automated-quality-characteristic-measures/.

[43] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. https://doi.org/10.1109/TSE.1976.233837.

[44] Ambler Thompson and Barry N. Taylor. Guide for the use of the Interational System of Units (SI). NIST Special Publication 811, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, March 2008. http://www.nist.gov/pml/pubs/sp811/.

[45] IEEE. *The Authoritative Dictionary of IEEE Standards Terms, 7th edition (Section B)*. IEEE 100-2000 B, https://doi.org/10.1109/IEEESTD.2000.322233.

[46] IEEE. *IEEE Standard for Prefixes for Binary Multiples*. IEEE 1541-2002, https://doi.org/10.1109/IEEESTD.2009.5254933.

[47] Arthur H. Watson and Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, September 1996. http://www.mccabe.com/pdf/mccabe-nist235r.pdf.

[48] Maurice H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[49] Automated Function Points (AFP), version 1.0. Document formal/2014-01-03, Object Management Group, January 2014. http://www.omg.org/cgi-bin/doc?formal/2014-01-03.

[50] James M. Bieman and Byung-Koo Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the Symposium on Software Reusability*, pages 259–262, 1995. https://doi.org/10.1145/223427.211856.

[51] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, 1995. http://www.isys.uni-klu.ac.at/PDF/1995-0043-MHBM.pdf.

[52] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FCE)*, pages 354–364, 2011. https://doi.org/10.1145/2025113.2025162.

[53] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 2nd edition, 1995.

[54] Sayed Mehran Sharafi and Zahra Sabet. A metric-based approach for measuring the conceptual integrity of software architectures. *International Journal of Software Engineering & Applications*, 6(1), January 2015. https://doi.org/10.5121/ijsea.2015.6109.

[55] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[56] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley Computer Publishing, 1998.

[57] Radu Marinescu. Measuring software: From data to actionable knowledge (keynote presentation). In *Second International Workshop on Software Architecture and Metrics*, 2015. http://www.slideshare.net/radumarinescu/measuring-software-from-data-to-actionable-knowledge.

[58] ISO. *Software and systems engineering—Software measurement—IFPUG functional size measurement method 2009*. ISO/IEC 20926:2009, http://www.iso.org/.

[59] ISO. *Software engineering—COSMIC: a functional size measurement method*. ISO/IEC 19761:2011, http://www.iso.org/.

[60] COSMIC. *The COSMIC Functional Size Measurement Method Version 4.0.1 Measurement Manual*, April 2015. http://cosmic-sizing.org/publications/measurement-manual-401/.

[61] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et du jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:547–579, January 1901. https://doi.org/10.5169/seals-266450.

[62] Emelie Engström and Per Runeson. Software product line testing—a systematic mapping study. *Information and Software Technology*, 53(1):2–13, January 2011. https://doi.org/10.1016/j.infsof.2010.05.011.

[63] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines, November 2012. http://arxiv.org/abs/1211.5451.

[64] Alain Abran. *Software Metrics and Software Metrology*. Wiley, 2010.

[65] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 3rd edition, 2014.

[66] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, September 1981. https://doi.org/10.1109/TSE.1981.231113.

[67] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979.

[68] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, 2nd edition, 2002.

[69] Eric S. Raymond. Heisenbug. In *Jargon File,* version 4.4.7. http://catb.org/jargon/html/H/heisenbug.html.

[70] Events, in perf tutorial, 2015. https://perf.wiki.kernel.org/index.php/Tutorial#Events.

[71] Wikipedia. Pentium FDIV bug, 2015. https://en.wikipedia.org/wiki/Pentium_FDIV_bug.

[72] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997. https://doi.org/10.1145/267580.267590.

[73] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[74] Hadi Hemmati and Lionel Briand. An industrial investigation of similarity measures for model-based test case selection. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 141–150, November 2010. https://doi.org/10.1109/ISSRE.2010.9.

[75] Ulf Grenander and R. F. Tsao. Quantitative methods for evaluating computer system performance: a review and proposals. In Walter Freiberger, editor, *Statistical Computer Performance Evaluation*, pages 3–24. Academic Press, 1972. Proceedings of a conference held at Brown University, Nov. 22–23, 1971.

[76] Robert Berry. Computer benchmark evaluation and design of experiments: A case study. *IEEE Transactions on Computers*, 41(10):1279–1289, October 1992. https://doi.org/10.1109/12.166605.

[77] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGARCH Computer Architecture News*, 37(1):265–276, March 2009. https://doi.org/10.1145/2528521.1508275.

[78] Jan Vitek and Tomas Kalibera. R3: repeatability, reproducibility and rigor. *ACM SIGPLAN Notices*, 47(4a):30–36, April 2012. https://doi.org/10.1145/2442776.2442781.

[79] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, pages 63–74, 2013. https://dl.acm.org/citation.cfm?id=2464160.

[80] Peter J. Mohr and William D. Phillips. Dimensionless units in the SI. *Metrologia*, 52(1):40–47, 2015. https://doi.org/10.1088/0026-1394/52/1/40.

[81] Cem Kaner and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *Proceedings of the 10th International Software Metrics Symposium*, 2004.

[82] M. A. Johnson. Effective and appropriate use of controlled experimentation in software development research. Master's thesis, Portland State University, 1996.

[83] L. Finkelstein. Theory and philosophy of measurement. In P. H. Sydenham, editor, *Handbook of Measurement Science*, volume 1. Wiley, 1982.

[84] David Flater. Configuration of profiling tools for C/C++ applications under 64-bit Linux. NIST Technical Note 1790, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, March 2013. https://doi.org/10.6028/NIST.TN.1790.

[85] David Flater. Screening for factors affecting application performance in profiling measurements. NIST Technical Note 1855, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, October 2014. https://doi.org/10.6028/NIST.TN.1855.

[86] Marc C. Kennedy and Anthony O'Hagan. Bayesian calibration of computer models. *Journal of the Royal Statistical Society, Series B*, 63(3):425–464, 2001. https://doi.org/10.1111/1467-9868.00294.

[87] Evaluate Collaboratory. Measurement contexts. http://evaluate.inf.usi.ch/anti-patterns/measurement-contexts.

[88] Clifton A. Ericson, II. *Hazard Analysis Techniques for System Safety*. Wiley, 2005.

[89] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.

[90] John A. Simpson. Foundations of metrology. *Journal of Research of the National Bureau of Standards*, 86(3):281–292, May 1981. https://doi.org/10.6028/jres.086.010.

[91] Robert J. Nemiroff, Ryan Connolly, Justin Holmes, and Alexander B. Kostinski. Bounds on spectral dispersion from Fermi-detected gamma ray bursts. *Physical Review Letters*, 108(23), June 2012. https://doi.org/10.1103/PhysRevLett.108.231103.

[92] P. de Bièvre and H. S. Peiser. 'Atomic weight'—The name, its history, definition, and units. *Pure and Applied Chemistry*, 64(10):1535–1543, 1992. https://doi.org/10.1351/pac199264101535.

[93] B. P. Leonard. Comment on 'Dimensionless units in the SI'. *Metrologia*, 52(4):613–616, 2015. https://doi.org/10.1088/0026-1394/52/4/613.

[94] Peter J. Mohr and William D. Phillips. Reply to comments on 'Dimensionless units in the SI'. *Metrologia*, 52(4):617–618, 2015. https://doi.org/10.1088/0026-1394/52/4/617.

[95] Philip B. Crosby. *Quality is Free: The Art of Making Quality Certain*. McGraw-Hill, 1979.

[96] W. Edwards Deming. *Out of the Crisis*. MIT Press, 1982. Republished in 2000.

[97] Joseph M. Juran. *Quality Control Handbook*. McGraw-Hill, 1951. 6th edition was published in 2010 as *Juran's Quality Handbook*.

[98] Christopher Domas. The memory sinkhole. In *Black Hat*, 2015. https://goo.gl/4f1FfP.

[99] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001. http://www.agilemanifesto.org/.

[100] Ken Pugh. *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*. Addison-Wesley Professional, 2011.

[101] Lingyu Wang, Sushil Jajodia, Anoop Singhal, and Steven Noel. k-zero day safety: A network security metric for measuring the risk of unknown vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 11(1):30–44, January 2014. https://doi.org/10.1109/TDSC.2013.24.

[102] Glenn Shafer. *A mathematical theory of evidence*. Princeton University Press, 1976.

[103] John Knight, Ben Rodes, and Kim Wasson. A security metric based on security arguments. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics (WeTSOM)*, pages 66–72, 2014. https://doi.org/10.1145/2593868.2593880.

[104] Timm Grams. Reliability growth models criticized, August 1999. http://www2.hs-fulda.de/~grams/RGM/RGMcriticized.html.

A Building the Future proposal (as-funded)

TITLE: A Rational Foundation for Software Metrology

CONTACTS: David Flater, Paul E. Black, Don Tobin, Stephen Wood, Will Guthrie

OVERVIEW: Many items in the call for proposals include metrology for information technology or software, yet little progress has been made on an SI-like metrological foundation since the work of Gray, Hogan, etc. a decade ago or more. Given a physical object, one can determine physical properties using measurement principles. In contrast, most software metrics are merely counting schemes that, at best, correlate with or are estimates of properties or quantities. We propose to establish a rational foundation for software metrics through two aspects: first, top down, to establish a theory of software measurement; second, bottom up, to identify specific purposes for which software measurements are needed, possible software quantities, base units, and objects of metrology.

RESULTS:

- A white paper delineating a theory of software metrology, including principles and philosophy.
- An annotated bibliography of software measurements.
- Publications or presentations whose goal is to increase the level of rigor expected in software measurements.
- An initial set of possible software quantities with their base and derived units, which represent progress toward a coherent set meaningfully integrated with SI.

WHY NIST/ITL: ITL was the home of foundational work in software metrology, for instance, funding McCabe's cyclomatic complexity work in the 1970s, Gray in the 1990s, Hogan et al. around 2001, and ACMD's 2007 Foundations of Measurement Science for Information Systems. NIST has the extensive background in metrology along with expertise in practical applications needing software measurements, such as security, manufacturing, and cyber-physical systems. As a leader in metrology, NIST can raise the expectation for software metrology from the current ad-hoc "street light" computations to widely accepted measurements traceable to solid principles, rigorously defined quantities, and base and derived units.

ACTIVITIES:

- Collect existing software measures and annotate them with purposes they serve or needs they satisfy, quantities or properties being measured, units of those quantities, and objects being measured, e.g. algorithm, architecture, source code, binary, or an execution run.
- Cross-connect, compare, and refine these needs, quantities, units, and objects toward a coherent set. For instance, there may be two measures of "complexity," but are they the same or is one the complexity of human understanding (cognitive load) and the other computer analysis complexity? Different measures may be based on the same units, such as code size, speed of execution, or number of data paths.
- Understand the principles of physical metrology and map them to software metrology as far as possible. Interpret International Vocabulary of Metrology (VIM) terms in the context of software to establish fundamental concepts. Identify failures (and successes) or limitations of software measurements and understand root causes.

EXPECTED IMPACT:

- Establish a common vocabulary of software quantities and units with accepted, objective definitions.
- Show which software properties are *measurable* in a strict sense and which can only be assessed or characterized less rigorously because the conditions for measurement are not present.
- Allow deeper understanding of what constitutes or produces software quality, security, etc.

RESOURCES:

- $100K

B Links

The following web sites containing related reading were valid as of January 2016. This list is provided as a convenience; the inclusion of particular sites does not imply that they are the best available for the purpose. Conferences, journals, and related societies are not repeated here (see References instead).

NIST projects:

- Software Assurance Metrics and Tool Evaluation: http://samate.nist.gov/
- Automated Combinatorial Testing for Software: http://csrc.nist.gov/groups/SNS/acts/
- Software Performance Project: https://goo.gl/yJdBbx

Other:

- International Bureau of Weights and Measures: http://www.bipm.org/
- Joint Committee for Guides in Metrology: http://www.bipm.org/en/committees/jc/jcgm/
- Scientific Test and Analysis Techniques in Test and Evaluation Center of Excellence, Graduate School of Engineering and Management, Air Force Institute of Technology: https://www.afit.edu/STAT/; esp. best practices and case studies archive: https://www.afit.edu/STAT/page.cfm?page=713
- Measurement and analysis page of Software Engineering Institute: https://www.sei.cmu.edu/measurement/
- Software quality page of American Society for Quality: https://asq.org/learn-about-quality/software-quality/overview/overview.html
- Advanced Center on Software Testing and Quality Assurance: http://paris.utdallas.edu/stqa/
- Evaluate Collaboratory (experimental evaluation of software and systems in computer science): http://evaluate.inf.usi.ch/