The Wonderful Wizard of LoC

Paying Attention to the Man behind the Curtain of Lines-of-Code Metrics

Kalev Alpernas
Tel Aviv University
Tel Aviv, Israel

Yotam M. Y. Feldman
Tel Aviv University
Tel Aviv, Israel

Hila Peleg
University of California, San Diego
La Jolla, CA, USA

Abstract

Lines-of-code metrics (`loc`) are commonly reported in Programming Languages (PL), Software Engineering (SE), and Systems papers. This convention has several different, often contradictory, goals, including demonstrating the 'hardness' of a problem, and demonstrating the 'easiness' of a problem. In many cases, the reporting of `loc` metrics is done not with a clearly communicated intention, but instead in an automatic, checkbox-ticking, manner.

In this paper we investigate the uses of code metrics in PL, SE, and System papers. We consider the different goals that reporting metrics aims to achieve, several various domains wherein metrics are relevant, and various alternative metrics and their pros and cons for the different goals and domains. We argue that communicating claims about research software is usually best achieved not by reporting quantitative metrics, but by reporting the qualitative experience of researchers, and propose guidelines for the cases when quantitative metrics are appropriate.

We end with a case study of the one area in which lines of code are not the default measurement—code produced by papers' solutions—and identify how measurements offered are used to support an explicit claim about the algorithm. Inspired by this positive example, we call for other cogent measures to be developed to support other claims authors wish to make.

CCS Concepts: • **General and reference** → **Metrics**.

Keywords: lines of code, research papers, loc

1 Introduction

Lines of code are everywhere. In Programming Languages, Software Engineering, and Systems (PLSES) research, software often plays a central role as the implementation of a paper's ideas, and the input or output of that implementation. In the academic papers that describe this research, there is a tendency to, among other things, measure that software. A sentence such as "we implemented our algorithm in 2376 lines of Go code" would not be out of place in many PLSES papers, and neither is dividing the benchmark programs on which an implementation is evaluated into difficulty buckets by their sizes.

Many of these measurements are in *lines-of-code* (`loc`), which is habitually used in such cases to measure, more or less, the number of lines in the source code. This reporting has many possible goals in a research paper. It may be there to communicate to the reader that something is hard, to express that something is easy, and, at times, it might be there for no reason at all—authors have seen other papers report implementation sizes, and a cargo cult belief that it is necessary to simply report this fact has developed.

Academic papers are so short and dense that anything that has gone in has ultimately bumped something else out. Therefore, as readers of papers, we must ask: why are these measurements there, and what are they contributing to our understanding?

In this essay we examine the `loc` practice from this perspective: what purposes do lines of code aim to serve (often implicitly)? Does `loc` actually fulfill its intended role in the paper? How can authors better expound on their arguments, in ways that are more meaningful and impactful?

A number of lines of code is not particularly useful to the reader for the most part. The measurement itself is not well defined except in its most naive implementation of counting everything, including non-code lines (including blanks, comments, and includes), which is not a very appealing metric. The fact that every project is measured differently by its own authors means no two measurements in two papers are comparable (with order of magnitude variability, in extreme cases), and these numbers end up being little more than noise taking up space in the paper out of a sense of obligation.

To validate our intuition that this is as ubiquitous as it seems, we surveyed a sample of PLSES papers that are considered good—recent Distinguished Paper Award winners from top tier conferences. And while some papers do not measure their code at all, of those that do, most do so in loc. Additionally, in reporting the measurements of the types of code where loc is more prevalent, the claims are rarely stated explicitly.

We also surveyed existing quantitative measures of code, to try to understand why, with many existing options, loc is such a go-to measure. Most of these measures were developed with other goals in mind than to provide support to the claims papers tend to make, and are, additionally, harder to compute. This means loc becomes a comfortable default, still not providing the best support, but at least commonly used and easy to compute.

This essay attempts to deconstruct the various narrative purposes underlying loc reporting in PLSES papers today, and argues that these narratives are generally better served by means other than loc. It is revealing, perhaps, that locs are often reported while leaving the purpose mysteriously unannounced; as if authors are aware that this measure does not in fact substantiate the underlying, implicit, claim. That claim is often some *qualitative property* of the solution or benchmarks. We contend that papers should choose to do the reverse: make the claim explicit, even without presenting supporting data, rather than presenting not-truly-supporting data for some ghost claim. We argue that in many cases, a *qualitative explanation*, though subjective, is more meaningful, informative, interesting, and compelling than loc metrics.

Some claims and discussions can benefit from loc reports as supporting evidence, but even then the context in which this measurement fits is invaluable for readers looking for remaining challenges or ways to reproduce the results. In such cases authors must not forget to avoid the pitfalls of bad loc reporting.

While the reporting on the size of solutions and benchmarks was, to varying degrees, frequently problematic, we did find one exception to our above complaints: the way output code is reported. Section 7 is dedicated to examining the case of measuring output code as already embodying all our suggestions: making claims about the code explicit, measuring in a way that supports those claims rather than defaulting to loc, and presenting the two together.

Our vision is for research papers to be interesting, instructing, enlightening. A reader who is reading the implementation section of a paper wants to know what was difficult–perhaps this is precisely the problem they are having right now!—and what they should expect to be easy. Pseudo-empirical measures that communicate no information serve no purpose, and in fact do the opposite, they communicate less information to the reader and thereby make the work less reproducible.

This essay calls for:

1. Explicit discussions of qualitative claims about software artifacts and benchmarks, and ways that would increase the value of papers, both for their assessment and their contribution to the community.
2. Conscious, rigorous reports of loc, so as not to render them meaningless.
3. Revisiting the available quantitative measures for software using advances in programming languages and software engineering research.

2 Why We Measure Code

In this section we take a closer look at the role of code in PLSES research, what about it gets reported in academic publications, and, perhaps most importantly, why.

What code. Software has several roles in PLSES research. A paper's *solution* is often implemented in software as a new system, tool, or algorithm. Software can also be part of the *input* to the solution, in which case code is present as benchmarks in the evaluation. Some tools also generate code as the *output* of the solution, as in program repair or synthesis, compilation to an intermediate representation, or language to language translation. Code, in all its different facets, is thus central to many research papers.

In describing all this code, authors often present seemingly-empirical measures. These measurements are usually a part of articulating their claims; sometimes they are present, whereas the claim itself is implicit.

Claims about code. We suggest the following (inevitably partial) list of claims about software that are supported, implied, or appear in correlation to measurements of code. The list is categorized by what is being described: the code itself, the development process, or the problem the software solves.

- **Properties of the development process**
  - The system was hard to implement (because the contribution is large).
  - The system quite easy to implement (because our solution is very concise).
  - The system wasn't very hard to implement, and we could do it again in another context (a different operating system, programming language, problem domain, etc.).
- **Properties of the code (the end-product)**
  - The system has many capabilities.
  - The system interacts with many different components, (e.g. kernel subsystems) and the solution has to address them all.
  - The system interacts with few components (making the solution "surgical").
  - Confidence in the system's correctness is high.
  - The system is extensible.
  - The system is maintainable.

– **Properties of the problem domain**
  ○ The problem we solve is hard: the simplest solution takes a lot of effort.
  ○ The benchmark task the solution is evaluated on is complex to solve.

The rhetoric of code measures. In the next section, we will discuss the ways in which code *can* be measured, but we first pause to reflect on the fact that many of these claims are qualitative. Providing an empirical measurement of the code as "proof" for them is, at best, a *surrogate* measure, or a measure which the authors believe is correlated well enough with their claim that it can serve as proof[1].

For example, a low `loc` count can be a surrogate measure for "the system is maintainable": it is a commonly held belief that a smaller codebase will be easier for a programmer to understand and remember all the details of, and therefore to repair it when necessary. However, this measure does not take into account other things believed to also correlate with maintainability: adhering to best practices like abstraction and encapsulation, comments, and documentation [20].

One may consider the surrogate measure to be sufficient, or decide other empirical measures covering other angles should be reported (for program maintainability, Zhang et al. [20] offer a list of 39 measures in 6 categories, for instance). However, simply adding more measures may not be sufficient for the reader who is seeing the numbers reported and asks, *why?* Or in other words, does the reported measure improve the understanding of the reader, advance the point of the paper, or highlight something about the work? And if not, what and how do we report about software to make sure that is the case?

3 How We Could Measure Code

Estimating and analyzing programming effort and code complexity has many useful applications for software developers and their managers, such as assessing the risk introduced by a software change, or planning time-frames for feature development. The practical benefits of having meaningful estimates have resulted in a rich history of research in this field. In this section, we present a short and necessarily partial overview of the field.

Broadly, we partition the field into two categories – *Code size estimation* and *Programming effort estimation.* Code size estimation deals with estimating the size of the produced code according to various metrics (several of which we describe in Section 3.1), with the underlying assumption being that the larger the code base is, the more effort it required to produce, and the more complex the product.

Programming effort estimation, on the other hand, focuses on estimating the amount of work that was undertaken in

---

[1] As in clinical research, where *surrogate outcomes* are used as a less expensive or invasive way to test true outcomes, but suffer from a slew of validity problems due to the gap between what they test and the actual outcome [7].

the process of creating the software. The estimation methods (Section 3.2) range from subjective effort reporting by developers, to relying on auxiliary development statistics such as those produced or derived from version control systems.

3.1 Code Size Estimation

Code size estimation techniques focus on measuring the size of a software artifact. There have been many metrics proposed over the years for measuring software size, each with its own pros and cons. We list here a few of them.

Lines of code. The simplest metric for measuring code size is measuring the number of lines of code in the source files of a project. The definition of what counts as source code is somewhat arbitrary – for example, do you count header files in C programs or not?

Source lines of code. A more nuanced version of the `loc` metric, *source lines of code* counts the number of source code lines in the source files of a project. This will usually exclude comment lines, empty lines, and syntactic constructs that do not contain program logic, such as e.g., curly-brackets that open or close blocks.

Given the ubiquity of `sloc` measurements, it is no surprise that a lot of practical [1] and theoretical [16, 17, 19] progress has been made over the years, fine-tuning and generalizing the results produced by `sloc` tools.

Some industry projects extend `sloc` to the even more discerning "significant lines of code", which rule out code lines that are necessary but unimportant, such as language-specific delimiters, auto-generated code, and other things deemed unimportant by the project. Most of the tools to count significant lines of code are internal to organizations, but open-source implementations exist [2].

ASTs. `loc` and `sloc` metrics rely heavily on the source code as it appears in the system, which makes them extremely sensitive to the choice of source language and individual programming style. A slightly more general approach is to use the size of *Abstract Syntax Trees* (or similar compilation-time construct) as a unit of code size measurement [3, 11].

ASTs are an abstract representation of the syntactic structure of the program. They have the dual benefit of abstracting away the specific style choices made by developers, as well as the syntactic differences between programming language that share syntactic constructs. For example, a conditional statement in Java or in Python will produce the same AST.

Number of functions. In the spirit of abstracting away the details in the source code, an even coarser abstraction than AST is that of functions. Assuming (perhaps perilously) that functions in the code are used to encapsulate some small unit of functionality, the number of functions in the source code can stand in for the complexity of the code [10].

(a) Contribution heatmap

(b) Contributions overview **(c)** Project contributions
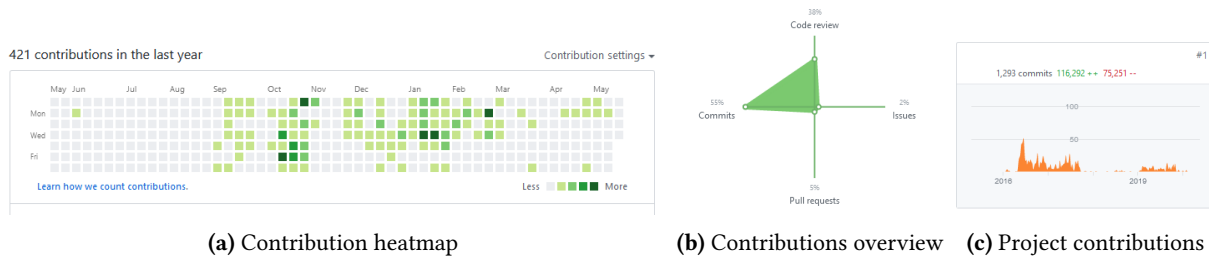
Figure 1. Various ways that GitHub uses to show a developer's history of contribution

Counting the number of functions is even less sensitive to the differences in syntax between languages, as it measures a relatively high-level property of the code, but it is still sensitive to coding style and to the way developers logically partition their code.

McCabe's Cyclomatic Complexity. This metric, as well as Halstead's metric below, derive a single numerical value from the source code. The resulting value is well correlated with complex code, and is ostensibly less sensitive to syntactic nuances (coding style, for example), and more representative of actual complexity of code.

McCabe's cyclomatic complexity [15] is a function over the control flow graph (CFG) of the program, corresponding to the number of loops and branches in the program. Technically, for a control flow graph with v nodes, e edges and p exit points (e.g., return statements), the cyclomatic complexity metric is:

$$C = e - v + 2p$$

The metric grows as edges and exit points are added to the control flow graph—representing more complex relations in the execution and a more complex program flow—and shrinks as nodes are added, representing a program with a more verbose linear execution but fewer loops.

The reliance on control flow graphs in McCabe's metric makes it oblivious to complexity arising from declarative syntactical constructs (e.g., class definitions in object oriented languages), and to complexity in languages where complexity does not necessarily arise from control flow (e.g., purely functional languages such as Haskell).

Halstead's Software Science Metrics. Halstead's software science metrics [9] are an attempt to measure the logical complexity of the code by computing the proportion of redundant operators and operands in the program. The intuition behind these metrics is that algorithms are easier to understand the more the same operands and operators repeat in the code. For example, code that performs the same operation on the same variable (e.g., a loop that advances a counter) is easier to understand than code that performs multiple different operations on a set of different variables making the interactions complex to understand.

Choice of size metric. The various different metrics described herein allude to a variety of options, each producing a different result which further complicates the choice of metric. However, it turns out that (perhaps somewhat counter-intuitively) all these metrics correlate strongly with one another [10]. Consequently, out of these approaches, the best method will usually be the one that is easiest to understand and easiest to measure, explaining, perhaps, the vast popularity of `loc` as a metric. The problems of `loc`/`sloc` measurements on which we expand in Section 5 are thus shared by other size metrics.

3.2 Programming Effort Estimation

Programming effort estimation techniques are focused on estimating how much work is needed to produce the software artifact in question. These techniques are most commonly used in industry in the planning stages of projects, with the goal of creating a workplan or a project schedule.

These techniques focus on the amount of individual effort required by a developer or group of developers, and consequently are inherently more subjective than code size estimation techniques. However, as we show in Section 6.1, what might seem at first glance to be a drawback of the approaches is actually an advantage.

(The mythical) programmer-month. The simplest way to estimate the development effort is to, well, estimate the amount of time it took a person to do the actual work. This metric is usually called *programmer-month* or *person-month*, and is essentially a self-reporting of the amount of time the project took, multiplied by the number of people that worked on the project. If one person worked on the project for one year, then the project is said to have taken one person-year. If two people worked on a project for four months then the project is said to have taken eight person-months.

This approach has it detractors, including the popular essay by Fred Brooks [4] arguing against the seemingly straightforward arithmetic behind the metric. Brooks correctly points out that adding programmers to a project will not necessarily reduce the amount of time a project takes overall, and sometimes, counterintuitively, increase the overall project time. A commonly used refrain against this metric states that while it is true that a human pregnancy typically

requires nine person-months of effort, it does not follow that three people can produce a baby in three months.

A slightly more nuanced variant of this metric also takes into account the experience level of the developers that worked on a project. A month of work by a senior developer does not equal a month of work by a junior developer. Similarly, a grad student-month and a tenured professor-month do not represent the same amount of development effort[2].

Version control statistics. A byproduct of software development is the version control history of a project. Developers use version control to checkpoint and backup their work, to simplify and manage collaborative work with other developers, and to ease the process of reverting faulty changes. A healthy development project includes registering (committing, in git lingo) all but the smallest changes with the version control system multiple times a day while working on a project. As a result, looking at the history of changes in a version control system may give some insight into the work that went into a project.

Figure 1 shows a few different ways that the repository service GitHub uses to visually show a developers contributions, all based on the statistics collected from the version control systems used in the projects the developers were contributing to. Figure 1a shows a developer's overall contributions in a specific year, where each square represents a single day's contributions (measured by a number of commits); the darker a day's color is, the more the developer contributed that day. Figure 1b shows a breakdown of the a developer's yearly contributions by kind. Figure 1c shows a developer's contributions *to a specific project* in a period of time, measured by lines-of-code added and deleted.

Figure 1a illustrates the benefit of using the version control history over (or in addition to) the programmer-month to report effort. While the reporting person month might describe the project as a 7 person-month project lasting from September to March, a review of the version-control history shows that the work was not uniform, and there were some periods on inactivity on the project. This does not necessarily mean that the developer were idle in that period, as in the case of a research project some of the work required is theoretical, but it does give one a sense of the technical programming work that was undertaken.

Number of different components modified. The final metric we consider is one that measure the breadth of a project by reporting on the number of unique components (files, modules, functions, etc.) that required modification in the process of implementing the solution. Focusing on breadth rather than quantifying the overall size of the work is, in some cases, a good proxy for the amount of effort required.

---

[2]We leave it to the discerning reader to decide which of these constitutes a larger investment of effort.

Consider for example an algorithm whose implementation requires adding 100 lines of code to a file in the Linux kernel. That project would be, in many cases, much easier to implement than a project that requires adding one line of code to 100 different files in the Linux kernel. Naturally, this metric is only useful when discussing changes to complex existing systems.

The underlying assumption behind this metric is that it elucidates the effort inherent to the process of learning the intricacies of an existing system and making precise modification that do not harm the correct behaviour of the system.

3.3 Why Most of These Are Not Used

The next section will examine recent academic papers to see in greater detail how they utilize code measures, the majority of which are in `loc` and `sloc`. Out of this wealth of measurement techniques, then, why does the community default to `loc`? We believe the answer is twofold. Some of these measures were developed for very different purposes—cyclomatic complexity is intended for use in test case creation, for instance—and are therefore more obviously ill-suited for describing code in this context. Additionally, they are harder to compute or accurately measure. Effort estimations are tainted by the fact that research projects go down wrong paths and must right themselves, and are often stopped and resumed, cyclomatic complexity and AST sizes require compiling the code at least partway, and even `sloc` requires downloading a tool rather than simply using `wc -l`, making `loc` always the easy alternative.

4 How We Actually Measure Code

To examine the ways software is reported on in the current scholarship, we surveyed recent papers from top-tier conferences in Programming Languages, Software Engineering, and Systems: POPL, PLDI, OOPSLA, ICSE, ASE, ASPLOS, OSDI, and SOSP. For each conference we surveyed at least ten distinguished paper award winners, rounding up to all the papers in the year of the 10th paper. E.g., for ICSE we surveyed 11 papers all from 2019, and for PLDI we surveyed 4 papers from 2019, 3 from 2018, and 4 from 2017. For OSDI and SOSP which are biennial conferences, we surveyed ten papers altogether.

4.1 Why Is Code Measured

Of the papers we surveyed, 44% make some sort of claim as to code sizes. This may be to remark on the size or difficulty of the problem domain, e.g., "programs containing millions of lines of code", or to explain the complexity of the solution proposed in the paper, the benchmarks that were tested, and code produced by the tools.

Figure 2a offers a breakdown of the papers per conference that make reference to code metrics, and Figure 2b shows
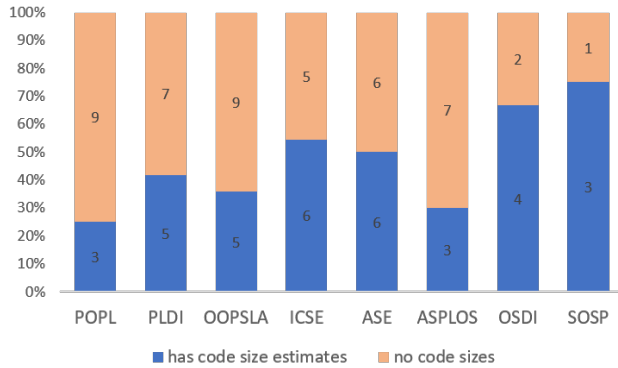
(a) Percentage of surveyed papers that measure software in any way, by conference.
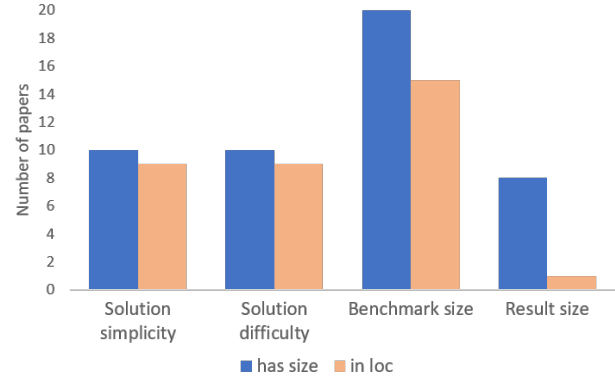
(b) Facts about software that were reported in papers, and how many were in `loc`.

Figure 2. Code measured in surveyed papers

what is measured. We divided metrics applied to the implementation into one of two opposite goals: indicating its *simplicity*, either indicating that the idea of the paper would be easy to implement, or that it is an elegant solution that does not require extraordinary development effort to work, or indicating its *difficulty*, to indicate that the proposed solution is not trivial. When a large number of lines is reported for a solution using neutral, factual language (e.g. "Our implementation consists of about 3,000 lines of code"), we consider this a (reserved) indication of difficulty. For measurement of benchmarks, if a measurement in `loc` was reported, even if it was not the exclusive form of measurement, we count the paper as measuring in `loc`; in practice only 6 papers combined `loc` with another measurements for benchmarks.

4.2 How Is Code Measured

Overwhelmingly, code is measured in lines of code. Of the papers we surveyed, 80% of the papers who employ code metrics use `loc` or `sloc`. One paper cites a large number of commits in open-source projects as an indication of their practical value. When measuring code benchmarks, they are sometimes measured differently, e.g., by the number of methods that the algorithm needs to process. Papers describing tools whose output is code often used other metrics to describe their output, which we discuss at greater length in Section 7.

Several papers describe the size of a code *change*. Two papers reported the number of lines that were changed in the operating system. Another paper emphasized that the implementation did not necessitate any changes to the underlying framework.

Two papers use effort estimation metrics. One paper reported the time required to develop a particular feature as "less than one day"); another reported an effort of "roughly four person-weeks each" for a certain task. The latter also included size estimation using `loc`.

We also sampled the artifacts of several papers and found that line counts given in the paper are at times taken crudely, including blank lines, comments, syntactic artifacts (e.g., lines containing only a curly brace or a semicolon), and preprocessor directives, which in our very small sample amounted to up to 16% of the reported length. (In other, more extreme examples, which we will show later, this can even be as much as a third of the reported length.)

Concrete examples. In the course of our survey we encountered quite a few examples of measuring that we found objectionable. However, since we are not here to point the finger at particular authors, we make no direct references to them. The hypothetical-like examples we provide throughout the paper are based on things we have seen, but are hopefully paraphrased into anonymization. We do, however, bring several examples of what we consider good practices in Section 6.

5 Why the Way We Actually Measure Code Is Bad

Even in cases where `loc` truly is a good fit and can support the authors' argument, `loc` reports are inherently problematic, unless they are done with care, and with a common pitfall in mind: that `loc` can mean different things for different people and codebases.

So far we have considered `loc` as a single metric used by a multitude of research papers. In reality, there are many different ways to measure `loc`. In C, for example, authors may choose differently in whether or not they count blank lines, comments, declarations, header files, preprocessing directives, etc. We have observed several papers that count such lines, and others that do not. As discussed in Section 3.1, the notion of source lines of code (`sloc`) aims to clarify some of this murkiness by explicitly measuring the number of logical statements, ignoring comments and blanks. However, `loc`

and `sloc` are commonly used interchangeably [16]. Further, `sloc` itself is open for interpretation: for instance, should the line `if A then B else C` be counted as one line or three statements? According to some standardization efforts, this is up to the organization to decide and make explicit [17].

The same example demonstrates another issue with `loc`. The line count depends on coding style and formatting, factors which are usually not of much interest *per se* when assessing software (nor affect quality, simplicity, etc. in a way that correlates with the number of lines). This can be resolved by using clever definitions of what constitutes a "line", but we are again led back to the problem of non-standardization.

Measuring `loc` is thus sensitive to counting policies and formatting. We argue that these points need to take into consideration when reporting or reading `loc`. At first sight, this seems not to be the case, according to research in software measurement, which shows that different ways to count `loc` are reliably correlated [10, 18]. Different formatting policies are perfectly correlated, but cannot be compared to each other directly [18]. This means that the exact way the code was measured can safely be ignored when comparing two measurements *performed the same way*. But when this is not the case, they cannot be ignored:

1. A standalone measurement ("we have implemented the system in 75476 lines of C") can be affected by the specific choice of what is measured, possibly leading to a misguided impression. The `loc` difference might accumulate to an order of magnitude in certain cases, and is thus inadequate even as a rough estimate;

2. A comparison of two measurements that may have been performed differently ("our proof takes 300 lines, whereas the previous paper reports their proof to require 800") might be highly biased. In our survey, we found instances where size measurements were compared to previous work, which could possibly be measured in a different way (unless the authors performed all measurements themselves and did not report this). When comparing two implementations, the choice of measurement policy may not be essential, but it is essential to ensure that both are done the same way.

Both problems can be addressed, at least in part, by always reporting the exact method of counting: this affixes the meaning of standalone measurements, and allows the comparison of code with a `loc` measurement from a different paper. In our survey, only one paper [5] explains how `loc` was measured, and use a standard tool for this purpose.

Interestingly, this already happens in a within-paper capacity: papers commonly list the `loc` size of benchmarks in a table, where their size can be compared, and implementations sizes are reported with a breakdown to individual components, which forms a comparison between parts of the system. Because they were all measured by the same researchers and (presumably) the same way, these numbers can be safely compared to each other. More accurately reporting the way they were measured would also make them comparable to other projects.

6 What We Should Be Doing

So far we have outlined why the current measuring and reporting in lines of code is not ideal. We now suggest a few ways researchers can report their work in a way that helps its audience.

6.1 The Argument for Quality Over Quantity

We discussed the many different ways code can be measured, and the ways these measures are either misleading or meaningless. We must first ask: why measure at all? The points of interest that code measurements are supposed to address are all, essentially, qualitative questions, and the size of the code is only a surrogate measure for them.

For example, instead of reporting that a particular part of the solution was implemented in a certain number of `loc`, authors could instead discuss the trivial or tricky points of the implementation. This would bring with it the added advantage of communicating to anyone who may try to implement the proposed solution what the difficult parts to implement are, and what made them hard. These often will be related to the gap between the algorithms presented in the paper, which are simplified, and their true implementation. For instance, consider replacing the statement "the Foo model of our tool took 1400 lines of C++ to implement" with the far more informative "in our C++ implementation, unlike in Algorithm 2, we must maintain bidirectional links between Foo Bars to be used for garbage collection. Updating these pointers on every update iteration holds a lot of the hidden complexity of the implementation." The same goes for a comparison of components of the code. Instead of "the proof of lemma 1 consists of 700 Coq lines, and lemma 2 of 400 lines", we can report "the proof of lemma 1 was tricky because capture-avoiding substitutions have some subtle points here, while lemma 2 is largely rudimentary inductive proofs about lists".

6.2 A Selection of Claims

We gather here several suggestions of ways to report on software and its development process. Each one is geared toward supporting a different claim about the software, so one must first decide *why* they are planning to report before deciding *what*.

High development effort. If the purpose is to demonstrate the sheer enormity of development effort that went into a system, say so, clearly and unambiguously. As demonstrated in the previous subsection, explicitly pointing to the work-heavy modules or difficult areas will provide the most value to readers. Quantitative metrics like version control

statistics or development time can be good supporting evidence here, since they encompass more of the effort than just a crude size measurement of the final product. The realities of research projects can make effort estimation, when it is frankly reported, enlightening but not without drawbacks, as illustrated by the following passage from a CAV'19 paper [14]: "An old version of the proof is [*sic*] developed on and off for two years. The current version is re-developed, using some ideas from the old version. The development of the new version took about 5 person months."

Simple and easy solution. If the purpose is to demonstrate the conciseness of the solution, loc may be a good metric. For instance, describing the SAT solver Tinisat [12] (which was not part of our survey) as written in under 550 lines of C++ code, does convey an unexpected level of simplicity for something as grandiose as a SAT solver. Huang [12] goes as far as to break down each component of the solver and record the length of its implementation, but since loc are most useful as a relative metric, comparing to the tens of thousands of lines of code implementing other SAT solvers better drives the point home.

However, if ease with which the proposed solution was implemented is the intended message, this may not be the case. Ideally, a short development time would indicate an easy to implement solution very directly. Feng et al. [6], for instance, describe the ease, not of writing their solution but of porting it to a different domain that "took us less than a day". However, since in research projects development time also accounts for the trial-and-error phase of research, this may be impractical in many cases. Researchers can state that the implementation posed "no particular difficulties", or even point out what were expected, likely pitfalls that turned out to not be a problem because of some feature of the solution.

Challenging benchmarks. If the purpose is to indicate that the solution scales to real-world programs as input, that is best indicated by giving it real-world programs as inputs. Otherwise, since the portion of the algorithm that needs to be shown to scale has a technical bottleneck, which can be anything from the number of CFG nodes in the program to the number of type variables in a function, it is best to explain them from that angle; a benchmark can be gargantuan in loc but entirely unchallenging for the algorithm, or vice versa. For example, the benchmark for a program repair algorithm can be a 6kloc program, but if the fault localization algorithm described considers only the function in which a crash occurred, it does not matter if the full program is six thousand or six million lines of code.

For no particular reason. If the purpose is just to have the size of the implementation reported, reconsider.

In the end, it is important to remember that while both a qualitative discussion and a pseudo-quantitative claim are unsubstantiated, the former can actually be informative and enlightening. Explaining why an algorithm was tricky to implement or why a benchmark truly challenges the tool makes the work clearer and more easily reproducible.

6.3 `loc`-ing Right

In the previous subsection, we mentioned cases where loc may be an effective measure to communicate the desired message. However, its effectiveness as a measure is highly eroded by the careless way in which it is often reported.

Let us consider, for example, Tinisat mentioned in the previous subsection. The size of its implementation is reported twice in two different papers [12, 13]. In the first, Tinisat is described as "implemented in under 800 lines of C++", whereas in the second the implementation of the first paper is referred to as "written in less than 550 lines of C++ (excluding comments and blank lines)".

This kind of disparity is common, but it is rarely corrected as it was for Tinisat. As previously mentioned, in the artifacts of papers we surveyed we found that it is not uncommon for the loc measures reported to include blank lines, comments, precondition headers or import statements, or, in short, to be performed by simple line-counting of a text file. The loc measure is already prone to large variation due to differences of formatting and coding style, further encumbering it with non-code reduces it from a problematic measure to downright uninformative.

Considering the disarray of counting methods, and the problems of interpreting the resulting measure, we believe that if loc must be reported:

1. Report size measurements only as part of *comparisons* of code segments measured using the same method, or while indicating the inconsistency when impossible (for example when comparing different programming languages).
2. Perform the count using de-facto standard tools for loc, such as cloc [1].

6.4 On Finding Better Quantitative Arguments

We have argued that loc measures are ill-suited to substantiate many of the interesting claims researchers wish to articulate. Can better measures be devised? We believe that this is not a single question, but one for each class of goals the measurement strives for.

Can properties such as modularity, maintainability etc., be measured in ways that adjust for the program's size, programming language, and technologies used? To describe the effort of implementing the solution, can we identify the specific domain challenges that make such endeavors challenging and quantify them? For example, in the domain of operating systems research, is it possible to identify a set of common challenges—dependence on specifics of the hardware, asynchronicity, availability etc.—and measure them, as a way to demonstrate that the problem being solved is

challenging? For the opposite aim, can we evaluate a streamlined development experience, even if it generates quite a lot of code or time?

Answers to such questions could provide methods to ascribe quantitative meaning to support qualitative claims, making them less subjective and more reproducible. Whether this would be worthwhile depends, we believe, on the level of insight such research directions could provide us about software and software development in general.

7 Case Study: Evaluating Projects That Produce Code as Output

As we have seen, when papers quantify code, the choice of metric is usually `loc`, and the reasons behind that choice are usually vague or nonexistent. One category of code quantification stands out from the rest, though, and that is the *output quantification* in projects that *produce code as output.* These include, among others, tools for *program synthesis*, *compilers*, and tools that perform *code repair.* In our survey, all but one of the papers that measured code output used meaningful metrics, with only one paper resorting to reporting lines of code.

In this section we investigate this discrepancy, and get a glimpse at a roadmap towards the adoption of meaningful metrics in the rest of the PLSES community.

Output code. Software projects that produce code as an output usually have two distinct axes that merit reporting. The first axis they share with other software projects, and that is the effort that the researchers exerted in creating the software project. The second axis is an analysis of the size of the outputs that the project or tool creates when operating on a given input. This section focuses on how papers report their findings on the second axis. We consider three types of projects that produce output code—synthesis tools, compilers, and code repair tools—though this list is by no means exhaustive. Many other kinds of projects fall into the category of projects that produce code as output, and we expect that the findings described here occur in those other cases as well.

Program synthesis tools take a program specification— some description of the desired program behaviour—as an input, and produce a program that matches the specification as an output. Specifications range in robustness and rigour from detailed formulae in temporal logic, through input-output pairs, to descriptions in natural language. In general, the problem of program synthesis is undecidable [8], and its more limited formulations are simply very hard, but recent years have seen significant advances in both theory and application, and synthesis tools are now able to produce more complex results. Demonstrating and communicating these advances requires quantifying the size and complexity of the programs produced by synthesis tools. As we explain in example 7.1, the problem domain of program synthesis

produces natural, and useful, metrics that have been adopted in practice by the research community.

Compilers accept as input programs in a given programming language, and produce a (hopefully) equivalent program in an assembly or intermediate language. Program repair tools accept a program as input, locate an incorrect behavior within it, and produce a different program that fixes this incorrect behavior.

In all of these cases, when evaluating a project, the size and complexity of the produced output are important both for objectively evaluating the project, and for comparison purposes. For example, a compiler that produces an otherwise equivalent but smaller output program may be preferable when considering a compilation target with limited resources, such as Internet-of-Things (IoT) devices. When reporting on compilers optimized for IoT applications, we should expect said reporting to describe the output code size for the representative programs, as well as comparisons to other similar compilers.

Measuring output code. The products of tools and algorithms presented in these papers are usually described in terms of the internal representation the algorithm uses. This may be an *abstract syntax tree* (AST), a *control flow graph* (CFG), or a target langauge for compilation. Describing the size of the result is most often in those terms: code generation is often measured in the size of the AST generated (height or number of nodes), repair is measured by the number of edits made to the AST, and the effectiveness of compilation and optimization are measured in the size in bytes or CFG nodes of the resulting executable. Indeed, in our survey we encountered only one paper that measured code output in lines of code.

The discrepancy between measuring output code and other measurements. Unlike the other code metrics, it is relatively easy to see why the size of the output is being measured, and in what ways the measurement will be meaningful. Often, this will be measuring whether or not the tool is solving the problem it has set out to solve, be it reducing the size of a program in optimization or finding nontrivial programs to meet a specification. Measurements of the output are crucial to show that the tool works as intended to solve its problem, and that the result is not problematic by being too small (e.g., a trivial program) or too large (e.g., a bloated executable).

Because of this tight-knit relationship with the operation of the algorithm, it is also more likely that measures of the output will be reported in some other metric than lines of code. Even if the result *is* code, it must be formatted to be measured in lines of code, but it already exists within the tool in a representation that can be measured more precisely,

removed from formatting conventions, splitting into statements, or other tricks which may make the `loc` measure bigger/smaller (as desired), but less precise and more removed from what success means for the algorithm.

Example 7.1 (AST Nodes as a measure of synthesis output).
Let us assume an enumerating synthesis algorithm returns the target program `foo(bar(2,"string"[1])) + 1`, where `foo`, `bar`, `+`, `[]` string literals and integer literals are all atomic components provided to the search algorithm.

Internally, an algorithm that constructs expressions via grammar rules on programs, explores possible programs and comes up with this one. This means that internally this program is represented as follows:
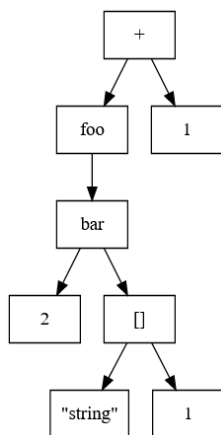
The researchers wish to report that their tool found a large output program. Since larger programs are harder to synthesize by virtue of being constructed from their individual components and picked from an astronomically large space of possible programs, this is no small feat. They therefore want to report its size to show their algorithm is good.

The program is a single expression. However, if they were to report their output sizes in `loc`, they may be tempted to format it like so:

```
let x0 = 1;
let x1 = 2;
let x2 = "string";
let x3 = x2[x0];
let x4 = bar(x1,x3);
let x5 = foo(x4);
let x6 = x5 + x0;
```

and report on a program that is 7 lines long[3].

However, the truly impressive feat of finding this result expression (as a single expression) is not that it can be broken up into 7 lines, but that the AST of the expression is of height 4, a depth where the search space is so large the

synthesis algorithm must be highly optimized to find the result. A different target program, `baz(1,2,3,4,5,6)` would also require 7 lines of code, but is only of height 1, and can be found very easily by even the most naive of algorithms.

More generally, measuring the results of code-manipulating and code-generating algorithms tends to have a clearly-apparent purpose—to demonstrate that the algorithm *works*. This means the measurement is far more likely to be tied to how the algorithm was described, and to be measured in a way that accentuates its core. Because describing the outputs of the algorithm in the experimental results is such a key part in showing the efficacy of a tool, one that often cannot be omitted without raising questions, the choice of metric is often far more deliberate—and far less likely to be `loc`.

8 Conclusion

The use of `loc` as a measure in PLSES papers is the commonplace standard for measuring the software research projects implement, consume, and produce. This near-excessive use is sometimes there to express that something is hard, other times to express that it is easy, and more often than should be the case, it expresses nothing at all. The standardization of a metric that expresses nothing but is believed to be important is problematic, but even more problematic when there is a point the authors are trying to make, but out of adherence to the standard they are trying to express it with `loc`.

This essay has attempted to analyze the real claims that papers are behind reported `loc` counts in recent PLSES papers, and found them to be qualitative claims, what the authors then try to support with quantitative proof, perhaps under the assumption they are better supported when empirical. We explained that `loc` is no better support for those claims about the work than qualitative explanations, and suggested ways those claims could be delivered, and what quantitative claims better support them, if those exist.

Using `loc` measurements in papers takes away from both readers' understanding and the space available to the authors to make their claims, and implicit claims may simply not register with readers at all. We showed examples of subjective and qualitative arguments that are more informative, compelling, and interesting than a non-indicative number. Even though we did show cases where `loc` was good supporting evidence to the claims authors would be making, we also pointed out `loc` should not be measured carelessly, and the way it was measured should be reported accurately so that the result is comparable to other measurements rather than a meaningless number. And even when a quantitative measure is reported, authors should still state along with it the claim it is there to support—qualitative measure should be made explicit and stated clearly. And finally, new quantitative measures that are better suited to support the kinds of claims PLSES research makes about its software make for interesting future PL and SE work.

---

[3]In a true feat of results padding, the two instances of the integer literal 1 could be separated into two different `let`s thereby earning an extra line of code.

Our hope is that `loc` reporting for the sake of `loc` reporting can stop, and that a new best practice takes hold, one where communicating to readers useful information that helps make the work reproducible is the norm.

Acknowledgments

References

[1] [n. d.]. CLOC: Count Lines of Code. ([n. d.]). https://github.com/AlDanial/cloc

[2] [n. d.]. jakevn/sloc: Performant significant-lines-of-code counter in 250 lines of Go. ([n. d.]). https://github.com/jakevn/sloc

[3] Bas Basten, Mark Hills, Paul Klint, Davy Landman, Ashim Shahi, Michael J Steindorfer, and Jurgen J Vinju. 2015. M3: A general model for code analytics in rascal. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. IEEE, 25–28.

[4] Frederick P. Brooks, Jr. 1975. The mythical man-month: Essays on software engineering. *Reading, Mass: Addison-Weley* (1975).

[5] Mikaela Cashman, Myra B. Cohen, Priya Ranjan, and Robert W. Cottingham. 2018. Navigating the maze: the impact of configurability in bioinformatics software. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 757–767. https://doi.org/10.1145/3238147.3240466

[6] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435.

[7] B. Goldacre. 2008. *Bad Science* (1 ed.). Fourth Estate. 338 pages.

[8] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[9] Maurice Howard Halstead et al. 1977. *Elements of software science*. Vol. 7. Elsevier New York.

[10] Israel Herraiz and Ahmed E Hassan. 2010. Beyond lines of code: Do we need more complexity metrics? *Making software: what really works, and why we believe it* (2010), 125–141.

[11] Yoshiki Higo, Akira Saitoh, Goro Yamada, Tatsuya Miyake, Shinji Kusumoto, and Katsuro Inoue. 2011. A pluggable tool for measuring software metrics from source code. In *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*. IEEE, 3–12.

[12] Jinbo Huang. 2007. A case for simple SAT solvers. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 839–846.

[13] Jinbo Huang. 2007. The Effect of Restarts on the Efficiency of Clause Learning.. In *IJCAI*, Vol. 7. 2318–2323.

[14] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal Verification of Quantum Algorithms Using Quantum Hoare Logic. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11562. Springer, 187–207. https://doi.org/10.1007/978-3-030-25543-5_12

[15] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[16] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. *A SLOC Counting Standard*. Technical Report. University of Southern California: Center for Systems and Software Engineering.

[17] Robert E Park. 1992. *Software size measurement: A framework for counting source statements*. Technical Report. Carnegie-Mellon University Pittsburgh PA Software Engineering Institute.

[18] Jarrett Rosenberg. 1997. Some misconceptions about lines of code. In *Proceedings fourth international software metrics symposium*. IEEE, 137–142.

[19] Martin Shepperd. 1995. *Foundations of Software Measurement*. Prentice Hall International (UK) Ltd., GBR.

[20] Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E Hassan. 2013. How does context affect the distribution of software maintainability metrics?. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 350–359.