The Evolution of C Programming Practices: A Study of the Unix Operating System 1973–2015

Diomidis Spinellis
dds@aueb.gr

Panos Louridas
louridas@aueb.gr

Maria Kechagia
mkechagia@aueb.gr

Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece

ABSTRACT

Tracking long-term progress in engineering and applied science allows us to take stock of things we have achieved, appreciate the factors that led to them, and set realistic goals for where we want to go. We formulate seven hypotheses associated with the long term evolution of C programming in the Unix operating system, and examine them by extracting, aggregating, and synthesising metrics from 66 snapshots obtained from a synthetic software configuration management repository covering a period of four decades. We found that over the years developers of the Unix operating system appear to have evolved their coding style in tandem with advancements in hardware technology, promoted modularity to tame rising complexity, adopted valuable new language features, allowed compilers to allocate registers on their behalf, and reached broad agreement regarding code formatting. The progress we have observed appears to be slowing or even reversing prompting the need for new sources of innovation to be discovered and followed.

CCS Concepts

•**Software and its engineering** → **Software evolution;** *Imperative languages; Software creation and management;* Open source model; •**General and reference** → **Empirical studies; Measurement;** •**Social and professional topics** → *Software maintenance; History of software;*

Keywords

C; coding style; coding practices; Unix; BSD; FreeBSD

1. INTRODUCTION

Tracking long-term progress in engineering and applied science allows us to take stock of things we have achieved, appreciate the factors that led to them, and set realistic goals for where we want to go. Progress can be tracked along two orthogonal axes. We can look at the processes (inputs) or at the resulting artefacts (outputs). Furthermore, we can examine both using either qualitative or quantitative means.

The objective of this work is to study the long term evolution of C programming in the context of the Unix operating system development. The practice of programming is affected by tools, languages, ergonomics, guidelines, processing power, conventions, as well as business and societal trends and developments. Specific factors that can drive long term progress in programming practices include the affordances and constraints of computer architecture, programming languages, development frameworks, compiler technology, the ergonomics of interfacing devices, programming guidelines, processing memory and speed, and social conventions. These might allow, among other things, the more liberal use of memory, the improved use of types, the avoidance of micro-optimisations, the writing of more descriptive code, the choice of appropriate encapsulation mechanisms, and the convergence toward a common coding style.

Here are a few examples. The gradual replacement of clunky teletypewriters with addressable-cursor visual display terminals in the 1970s may have promoted the use of longer, more descriptive identifiers and comments. Compilers using sophisticated graph colouring algorithms for register allocation and spilling [12] may have made it unnecessary to allocate registers in the source code by hand. The realisation that the overuse of the *goto* statement can lead to spaghetti code [13] might have discouraged its use. Similarly, one might hope that the recognition of the complexity and problems associated with the (mis)use of the C preprocessor [15,34,48,49] may have led to a reduced and more disciplined application of its facilities. Also, one would expect that the introduction and standardisation of new language features [2,23,45] would lead to their adoption by practitioners. Finally, the formation of strong developer communities, the maturing of the field, and improved communication facilities may lead to a convergence on code style.

In more formal terms, based on a simple-regression exploratory study [54], we established the following hypotheses, which we then proceeded to test with our data.

H1: Programming practices reflect technology affordances

If screen resolutions rise we expect developers to become more liberal with their use of screen space, as they are no longer constrained to use shorter identifiers and shorter lines. Higher communication bandwidth (think of the progress from a 110 bps ASR-33 teletypewriter, to a 9600 bps VT-100 character addressable terminal, to a 10MB Ethernet-connected

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884799

bitmap-screen workstation) makes typing more responsive and the refresh of large code bodies faster. Similarly, if barriers imposed to compilation unit sizes by the lack of memory or processing capacity are removed, we expect developers to abandon artificially-imposed file size limits, and move towards longer files packed with more functionality.

H2: Modularity increases with code size

As the Unix source code evolves and multiplies in size and complexity we expect developers to manage this growth through mechanisms that promote modularity. These include the use of the `static` keyword to limit the visibility of globally visible identifiers, and the pairing of header files with C implementation into discrete modules.

H3: New language features are increasingly used to saturation point

The C language has been changing over time with new features being added, albeit sparingly. We expect them to be increasingly used after being introduced, up to a point — certainly, a language feature can only be used a certain number of times in a program. For example, when the `unsigned` keyword was introduced, it could in theory, be used at most on all integer definitions and declarations of a program, and in practice only on those where the underlying value was indeed non-negative. We call this limit the feature's saturation point.

H4: Programmers trust the compiler for register allocation

In the days of yore, programmers had to deal themselves with the nitty-gritty details of register allocation by declaring variables with the `register` keyword. As compiler technology has improved, compilers have become more adept at the task. We expect that developers have been noticing this and have therefore been more and more trusting of the compilers with handling register allocation optimizations.

H5: Code formatting practices converge to a common standard

We expect developers working on a single project, such as Unix, to adopt a common coding standard, making the code base more homogeneous over time. This can be aided through the increased availability and use of collaboration mechanisms such as version control systems and online communities.

H6: Software complexity evolution follows self correction feedback mechanisms

We expect that as software evolves it becomes more complex, to a degree. A successful project cannot grow in complexity beyond the confines of human comprehension. Therefore we also expect that beyond a certain point self correction feedback mechanisms should kick in, bringing code complexity down.

H7: Code readability increases

As software and its development process and community evolve, growing in size and in complexity (to a degree), we expect its readability to increase, in order to make it manageable by its evolving community of developers. In addition, a long lived project will outlive its original developer cast and accommodations must be made for new cohorts.

The preceding hypotheses can be tested by examining instances of the code over time. By looking at the long term evolution of associated metrics we can determine whether they indeed change over time, as well as the direction and rate of change.

The results of the study on long term evolution of programming practices can be used to allocate the investment of effort in areas where progress has been efficiently achieved, and to look for new ways to tackle problems in areas showing a lack of significant progress. Also, given the hypothesis that the structure and internal quality attributes of a working, non-trivial software artefact will represent first and foremost the engineering requirements of its construction [51], the results can also indicate areas where developers rationally allocated improvement effort and areas where developers did not see a reason to invest.

This paper builds on the earlier short exploratory study [54] by explicitly stating and testing hypotheses, by using a more sophisticated statistical method for the analysis, by a detailed explanation of the employed methods, and by presenting possible causality links between external changes, the metrics, and the hypotheses. To formulate the hypotheses we took into account the general technological trends, such as improved screen technologies and software techniques, combined with metrics, which we could measure from our data and could bear on the trends. In the following sections we describe the methods of our study (Section 2), we present and discuss the results we obtained (Section 3), and outline related work (Section 4). Section 5 summarises our conclusions and provides directions for further work.

2. METHODS

Our study is based on a synthetic software configuration management repository tracking the long term evolution of the Unix operating system. At successive time points of significant releases we process the C source code files with a custom-developed tool to extract a variety of metrics for each file. We then synthesise these metrics into weighted values that are related to our hypotheses, and analyse the results over time using established statistical techniques.

2.1 Data Collection and Primary Metrics

The analysis of the Unix source code over the long term was made possible by the fact that important Unix material of historical importance has survived and is nowadays openly available. Although Unix was initially distributed with relatively restrictive licenses, significant parts of its early development have been released by one of its rights-holders (Caldera International) under a liberal license. Additional parts were developed or released as open source software by (among others) the University of California, Berkeley and the FreeBSD Project.

The primary sources of the material used in this study include source code snapshots of early released versions, which were obtained from the Unix Heritage Society archive, the CD-ROM images containing the full source archives of Berkeley's Computer Science Research Group (CSRG), the OldLinux site, and the FreeBSD archive. These snapshots were merged with past and current repositories, namely the CSRG SCCS [46] repository, the FreeBSD 1 CVS repository, and the Git mirror of modern FreeBSD development. This material plus results of primary research regarding authorship and genealogy formed the basis for constructing a synthetic Git repository, which allows the efficient retrieval and processing of the Unix source code covering a period of 44 years [52].

Table 1: Analysed Unix Releases

Name	Mean File Date	LoC
Research-V3	1973-08-30	5,963
Research-V5	1974-11-27	27,429
Research-V6	1975-06-03	49,630
BSD-1	1977-12-09	55,378
Bell-32V	1978-12-13	163,643
Research-V7	1979-01-31	157,003
BSD-2	1979-05-06	88,048
BSD-3	1979-08-18	269,646
BSD-4	1980-06-22	373,881
BSD-4.1 snap	1981-04-09	447,540
BSD-4.1c 2	1989-01-03	564,883
BSD-4.2	1990-06-23	683,428
386BSD-0.0	1991-05-13	655,191
386BSD-0.1	1991-06-08	768,469
BSD-4.3	1991-12-03	1,117,677
BSD-4.3 Tahoe	1992-06-21	1,315,451
BSD-4.3 Net 2	1993-07-02	1,445,124
FreeBSD 1.0	1993-07-02	3,515,680
FreeBSD 1.1	1993-09-17	1,519,925
FreeBSD 1.1.5	1993-11-16	1,698,792
BSD-4.3 Net 1	1993-12-25	179,292
BSD-4.4 Lite1	1994-01-15	2,436,446
BSD-4.4 Lite2	1994-01-20	2,487,503
BSD-4.4	1994-01-27	2,828,398
BSD-4.3 Reno	1994-02-26	1,217,391
FreeBSD 2.0	1994-10-09	6,152,630
FreeBSD 2.0.5	1995-01-12	2,154,728
FreeBSD 2.1.0	1995-02-19	2,204,647
FreeBSD 2.1.5	1995-05-20	2,254,479
FreeBSD 2.1.6	1995-05-31	2,266,688
FreeBSD 2.2.5	1996-05-05	2,615,395
FreeBSD 2.2.6	1996-08-07	2,679,569
FreeBSD 2.2.7	1996-09-16	2,710,214
FreeBSD 3.0.0	1997-07-04	3,371,194
FreeBSD 3.2.0	1997-08-26	3,527,404
FreeBSD 3.3.0	1998-03-19	3,575,250
FreeBSD 3.4.0	1998-04-22	3,656,716
FreeBSD 3.5.0	1998-05-17	3,684,268
FreeBSD 4.0.0	1999-03-29	4,422,666
FreeBSD 4.1.0	1999-05-30	4,632,772
FreeBSD 4.1.1	1999-06-29	4,689,328
FreeBSD 4.2.0	1999-07-16	4,717,764
FreeBSD 4.3.0	1999-09-27	4,786,370
FreeBSD 5.0.0	2001-11-02	5,434,960
FreeBSD 4.4.0	2002-02-03	4,881,244
FreeBSD 5.1.0	2002-03-22	5,490,790
FreeBSD 4.6.0	2002-07-03	5,050,687
FreeBSD 5.2.0	2002-08-23	5,706,097
FreeBSD 4.7.0	2002-10-05	5,184,702
FreeBSD 5.3.0	2004-10-16	6,093,719
FreeBSD 5.4.0	2005-04-20	6,126,108
FreeBSD 6.0.0	2005-10-09	6,330,668
FreeBSD 7.1.0	2005-12-09	7,401,793
FreeBSD 7.2.0	2006-01-19	7,501,567
FreeBSD 6.1.0	2006-04-30	6,424,602
FreeBSD 6.2.0	2006-11-14	6,530,096
FreeBSD 8.0.0	2007-02-21	8,016,942
FreeBSD 8.1.0	2007-05-08	8,152,027
FreeBSD 8.2.0	2007-07-22	8,358,507
FreeBSD 6.3.0	2007-11-24	6,614,077
FreeBSD 7.0.0	2007-12-22	7,231,515
FreeBSD 9.0.0	2008-09-22	9,230,038
FreeBSD 9.1.0	2009-02-01	9,497,551
FreeBSD 9.2.0	2009-06-09	9,674,294
FreeBSD 9.3.0	2009-10-29	10,048,538
FreeBSD 10.0.0	2010-11-09	10,767,581

Due to the fact that early releases are only available as snapshots, it was decided to study the code at points of significant software releases, rather than at fixed time intervals. This was done because obtaining code snapshots at fixed time intervals was only possible after 1990, when all software began to be tracked through various revision control systems. The 66 releases examined, the mean date of the files comprising them, and the size of C-proper source code files are listed in Table 1. We did not include header files in our study, to avoid skewing metrics associated with executable code (such as the number of functions per file) with results from files that typically do not include any such code. Interestingly, over the examined period the code body grew by more than three orders of magnitude, from six thousand to ten million lines. A few earlier revisions that existed in the repository were not examined, because they did not contain any C source code files.

We show and use each release's mean file date — based on averaging each file's last modification time — rather than the release date, because this reflects better the age of the corresponding code base. This also avoids the distortion that would be introduced by treating what were sometimes parallel lines of development as a linear sequence. Such parallel development took place during the Berkeley Unix evolution: over the early years with Research Unix and over the late years with 386BSD and FreeBSD.

Obtaining metrics from large code bodies is difficult for technical and operational reasons [20, 39]. On the technical side, code dependencies make it difficult to establish the full context needed in order to parse and semantically analyse the code. This is especially true for C code, where the compilation depends on system header files, compiler-defined macros, search paths, and compile-time flags passed through the build process [18, 33, 49]. The operational reasons are associated with the required throughput, though due to the relatively small number of releases we examined, this was not a major issue in this study.

We addressed the difficulty of parsing C source code without access to the original compilation environment by extending and using our *cmcalc*[1] open source tool, which efficiently calculates through static analysis a variety of C code quality metrics, without requiring full access to the compilation environment's parameters. The tool's operation is based on state machine logic (described in reference [53]), and will therefore produce reasonably accurate results without requiring access to header files and the like. Extensive unit tests (48% of the source code is devoted to them) were used to verify the tool's operation in diverse corner cases.

The *cmcalc* tool calculates size, language feature, code style, and commenting metrics; see references [25], [50, pp. 326–333] for more details. The metrics we used are the following.

Size metrics: number of characters, lines, statements, functions, identifiers.

Language features: declarations with internal linkage; number of C preprocessor directives; number of C preprocessor include directives; number of C preprocessor conditional directives (`#if`, `#ifdef`, `#elif`); number of `const`, `enum`, `goto`, `inline`, `noalias`, `register`, `restrict`, `signed`, `struct`, `union`, `unsigned`, `void`, and `volatile` keywords.

Code style: mean identifier length; spaces used for in-

[1] https://github.com/dspinellis/cqmetrics

dentation and their standard deviation; mean nesting level; code style infractions.

Commenting: number of comments and comment characters.

In addition, we derived a sloppiness metric (DKLUDGE) by using *fgrep* to count in the code the number of "kludge" words that may indicate problems in the code (FIXME, XXX, TODO, BUGBUG, and swearwords that cannot be reproduced in this paper).

The *cmcalc* tool calculates code style infractions from commonly agreed formatting guidelines. As there are a number of different approaches for formatting C code, *cmcalc* also measures the *consistency* of their application. Specifically, for each way to format a particular construct (for example putting a space after the `while` keyword) *cmcalc* measures the times, a, the rule is applied in the one way (e.g., putting a space) and the times, b, the rule is applied in the other way (omitting the space). Then, the file's style inconsistency for n style rules (19 in our case) as a percentage of possible cases is defined as follows.

$$\mathrm{SI} = \frac{\sum_{i=1}^{n} \min(a_i, b_i)}{\sum_{i=1}^{n} a_i + b_i} \tag{1}$$

We based the checked style rules on the Google,[2] FreeBSD,[3] GNU,[4] and the updated Indian Hill[5] guidelines.

We collected the metrics by iterating through the 66 releases listed in Table 1, checking out the code for each one of them, and running *cmcalc* on all C files of that release. Through this process we collected 490 thousand records containing in total 56 million values. The processing took about five hours on an eight-core server. A considerable speedup was achieved by parallelising the analysis of each file through the use of GNU parallel [57].

2.2 Derived Metrics

We aggregated the raw metrics at the level of each release, and calculated weighted derivative values needed to track the long term evolution of programming practices. The derived aggregate metrics are listed in Table 2.

Column-wise, the metrics are roughly clustered into those that are affected (↵) by programming language evolution, ergonomics (workstation technology and screen resolution), programming guidelines, processing capacity (CPU performance and RAM size), conventions, and tools. Further columns on the right indicate the hypothesis that each derived metric affects (⇀).

To control for size and thus make metrics comparable across releases, absolute numbers in the primary collected metrics were replaced in the derived metrics by corresponding density figures. The denominator for calculating each density is based on the numerator unit, and can be the corresponding number of files, lines, characters, statements, or identifiers. As an example, the comment character density is calculated as the fraction of comment characters over the number of characters across all source code files. We expect that metrics that are not density figures (FILINE, FISTMT, LICHAR, FULINE, IDCHAR, STNEST, and CMCHAR) are implicitly controlled for size through coding practice and convention. For example, when a function or a file increases in too much in size, developers will split it into smaller modules.

The following paragraphs outline how metrics are associated with the factors that may influence them. The analysis follows roughly the order in which the metrics are listed in Table 2.

The length of files and the corresponding functionality they provide can be decided to promote proper encapsulation and modularity. However, on resource constrained computers — think of a 128kB PDP-11 — large files could take overly long to compile, forcing developers to split them in order to minimise the impact of small changes on build performance. As an example, the 7th Research Edition implementation of the *refer* program seems to be arbitrarily split into nine files named `refer0.c` ... `refer8.c`.

Programming guidelines also typically dictate line length (lines should not exceed the number of characters that can fit in a display row), function length (functions should fit on a screen), identifier length (most names should be descriptive but not overly long [8]), and statement nesting (deep nesting should be avoided). On the other hand, all these are also affected by ergonomics. The lengths to which past developers went to avoid long identifiers survives to this day through the Unix `creat` (rather than `create`) system call. Also note that 1970s linkers only recognized the first six characters of identifiers, thus restricting the length of globally-visible identifiers. Higher resolution screens can display more characters per row, more rows on a screen, and thus allow for deeper nesting, while fast workstations make it easy to type and display long identifiers. However, long functions and corresponding deep nesting may be required on slower CPUs in order to avoid the overhead of function calls and the danger of hitting the limits of a shallow stack size. Also, IDEs with code completion can help entering long identifiers, while optimising compilers can avoid the function call cost through inlining.

The declaration of identifiers that are only internally visible within the compilation unit (`static`), depends on the existence of the corresponding language feature. The use of such declarations to limit the identifiers' global visibility is prescribed in guidelines, and can be assisted by tools that identify such problems.

The use of several keywords (`const`, `enum`, `unsigned`, `inline`, `noalias`, `restrict`, `signed`, `register`, `void`, `volatile`) is made possible through their introduction as language features and corresponding compiler support. In addition, the use of some (`const`, `enum`, `signed`, `unsigned`) can clarify the programmer's intent and avoid some errors. Furthermore, when these four keywords are used, static analysis methods can identify common error cases. On the other hand, the `register` and `inline` keywords are there to address deficiencies in the way compilers allocate registers [12] and inline functions, so their use should become less common as technology advances. The `noalias` keyword was a mistaken attempt by the ANSI C standardization committee to handle aliasing through pointers. We expect it to make an appearance for a brief (if any) period of time. The `volatile` and `restrict` keywords control the caching of values and express programmer intent regarding aliasing for cases that

[2] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

[3] http://www.freebsd.org/cgi/man.cgi?query=style&sektion=9

[4] https://www.gnu.org/prep/standards/html_node/Formatting.html

[5] http://www.cs.arizona.edu/%7Emccann/cstyle.html

Table 2: Derived Aggregate Metrics, Factors Affecting them, and Hypotheses they Affect

Name	Description	Language	Ergonomics	Guidelines	CPU & Storage	Conventions	Tools	H1	H2	H3	H4	H5	H6	H7
FILINE	Mean file length (lines)			↵	↵			↗						
FISTMT	Mean file functionality (statements)			↵	↵			↗						
LICHAR	Mean line length (characters)		↵	↵				↗						
FULINE	Mean function length (lines)		↵	↵				↗					↗	
IDCHAR	Mean identifier length		↵	↵			↵	↗						
STNEST	Mean statement nesting		↵	↵	↵		↵						↗	
DSTATIC	Internally visible declaration density	↵		↵			↵		↗					
DCONST	`const` keyword density	↵		↵			↵			↗				
DENUM	`enum` keyword density	↵		↵			↵			↗				
DUNS	`unsigned` keyword density	↵		↵						↗				
DINL	`inline` keyword density	↵					↵			↗				
DNOAL	`noalias` keyword density	↵					↵			↗				
DSIGN	`signed` keyword density	↵					↵			↗				
DREG	`register` keyword density	↵					↵				↗			
DREST	`restrict` keyword density	↵					↵			↗				
DVOID	`void` keyword density	↵					↵			↗				
DVOL	`volatile` keyword density	↵					↵			↗				
DGOTO	`goto` keyword density			↵									↗	
INDEV	Indentation spaces standard deviation			↵		↵	↵					↗		
FMINC	Formatting inconsistency			↵		↵	↵					↗		
INSPC	Mean indentation spaces		↵	↵		↵								↗
DSTMT	Statement density		↵	↵		↵								↗
DCMCHAR	Comment character density		↵	↵										↗
CMCHAR	Mean comment size		↵	↵										↗
DCM	Comment density		↵	↵										↗
DCPIF	C preprocessor conditional statement density			↵									↗	
DCPINC	C preprocessor include statement density			↵	↵		↵		↗					
DCPSTMT	C preprocessor non-include statement density	↵		↵	↵		↵						↗	
DKLUDGE	Kludge word density					↵	↵							↗

the compiler could not detect. We expect them to be used sparingly, with `volatile` used for only a small fixed number of values. The `goto` statement has been with us from the dawn of C and has been "considered harmful" for almost half a century [13].

The formatting style and the number of spaces used for indentation are a matter of convention. This can be established when code is collaboratively edited through version control systems and shared through online communities and as open source software. Consistency in both areas can also be aided through tools, such as code formatters, editors, and IDEs.

Spacing and the density of comments and statements, are also a matter of guidelines and ergonomics. The level of nesting and available screen real estate can affect the number of spaces used for indentation. Comments should be long and plentiful, while white space among statements should be used to separate code into logical blocks. Fast high-resolution workstations make it easy to type in and display such code.

Research findings regarding spacing and comments present a mixed picture. According to Miara et. al, the amount of spacing used for indentation significantly affects program comprehension: an indentation of six spaces or more can confuse developers [37]. On the other hand, Posnett et al. found that indentation measures did not have a significant impact on their code readability models [42]. Arafat and Riehle, in an empirical study on 5,229 active open source projects, found that commenting the source code is a practice that is consistently followed by successfull projects [3]. However, Buse and Weimer found that even though one would expect that the presence of comments improves readability, the correlation of comments with readability is modest [7]. In addition, Fluri et al. found that code and comments seldom co-evolve and that new code is often sparsely commented [16].

The problems associated with the use of the C preprocessor are well known [15, 34, 48, 49], and most guidelines advocate the avoidance of macro definitions and conditional compilation. Conversely, guidelines also advocate the use of header files (and the `#include` directive) in order to promote code modularity and portability. According to Steve Johnson [43, Ch. 17], early versions of Unix did not share structure definitions through header files.

> "Another major change in V7 was that Unix data structure declarations were now documented on header files, and included. Previous Unixes had actually printed the data structures (e.g., for directories) in the manual, from which people would copy it into their code. Needless to say, this was a major portability problem."

C preprocessor macros can often be replaced by exploiting newer language features, such as enumerations and inlined functions. However, these features and header file inclusion require additional processing power and more sophisticated compiler support.

We end the description of the factors associated with the metrics we tracked by noting that the annotation of code through colourful symbols and swearwords (DKLUDGE), is a matter of convention, which may be fading, being replaced by comments in issue tracking systems, online forums, and configuration management tools.

2.3 Statistical Analysis

We subjected the derived aggregate metrics to statistical analysis in order to discern longitudinal trends. As all the metrics we collected were ordered by date, we performed regression analysis with the days elapsed since the first release as the independent variable and each metric as the dependent variable. We used a General Additive Model (GAM) method [62, 63] with cubic splines as our regression model. We gauged the goodness of fit of each trend by calculating the adjusted R^2 value of each regression.

2.4 Threats to Validity

The internal validity of this study's findings would be threatened by inferring causal relationships without actually demonstrating the mechanism through which the cause drives the effect. In our study we have identified some factors that could affect long term programming practices. However, we have been careful not to draw any conclusions regarding causality, using the factors merely as a starting point for determining and arranging the metrics to examine.

The external validity of any findings is limited by the fact that only a single large system (Unix) has been studied. The developers of Unix are not likely to match the general population of programmers, for they include two winners of the National Medal of Technology Award and the Turing Award (Ken Thompson and Dennis Ritchie), two founders of Fortune 500 companies (Bill Joy and Eric Schmidt), and many highly accomplished scientists, technology authors, and practitioners. However, given the system's continued prevalence, utility, and importance, the lack of generalisability, is not the fatal problem it could otherwise be.

3. RESULTS AND DISCUSSION

Figure 2 contains the plots of all metrics over time, ordered alphabetically. The x axis of all plots is the release date. In every plot we show the regression line with 95% confidence intervals around it and we include the adjusted R^2 value as an indicator of the goodness of fit. In the following paragraphs we detail our findings regarding each hypothesis and discuss their meaning. Figure 1 shows the timeline of analyzed revisions and milestones associated with factors that could influence the evolution of programming practices. Although we have not explicitly performed regression analysis against these factors it may be helpful to refer to them in the discussion of our findings.

H1: Programming practices reflect technology affordances

The metrics FILINE, mean file length ($R^2 = 0.91$); FISTMT, mean file functionality ($R^2 = 0.90$); LICHAR, mean line length ($R^2 = 0.93$); and IDCHAR, mean identifier length ($R^2 = 0.94$), increase steadily over time. This lends strong support to the H1; as bigger, higher resolution, and increased bandwidth terminals or workstations became available, Unix developers have been eager to embrace new capabilities and evolve their programming practices to take advantage of them. The FULINE metric, mean function length, ($R^2 = 0.91$), is also related to this hypothesis, as bigger screens and higher resolutions allow longer functions to fit on screen. This could explain why mean function size increases steadily to about 60 lines. The drop that follows may be explained as a reaction to inreased function complexity, which we discuss in **H6** (Software complexity evolution follows self correction feedback mechanisms).

Our findings here lend support to technology and infrastructure investments that aid developers in their work. Nowadays these can include multiple high-resolution screens, fast computing clusters for compiling and testing the software, and a workplace that allows developers to concentrate.

H2: Modularity increases with code size

The metric DSTATIC, internally visible declaration density ($R^2 = 0.97$), increased steadily until reaching a peak, from which it has receded somewhat. Therefore for most of Unix history there is strong evidence supporting the hypothesis. The recent small decline in the use of the `static` keyword may be counterbalanced through the use of extralingual structuring mechanisms, namely the use of kernel modules and shared libraries.

A different measure of software modularity, the importing of functionality through header files as witnessed by DCPINC, C preprocessor include statement density ($R^2 = 0.37$) has, after an initial bumpy period until the early 1980s, remained remarkably stable over time. A significant rise in this metric (more imports) would indicate a rising number of efferent (outward) couplings. This increased number of dependencies across diverse modules might indicate a lower quality of modular design.

In general, the findings regarding this hypothesis support the conjecture that engineering requirements drive an artefact's quality features [51]. In terms of actions, they also support investment in modularity mechanisms. In modern systems the two levels of identifier visibility provided by the C programming language are showing their limits. (This might be another explanation for the recent decline in the `static` keyword density.) Mechanisms that can handle multiple levels of module federations from diverse sources, such as the C++ namespaces and the Java packages, show the way forward. However, these mechanisms cover only identifiers. Other important modularity issues remain open, and will become increasingly important as code size and reuse increase. These include versioning, deployment, (possibly live) updates, and access to shared resources, such as computing threads, power, the cache, and middleware.

H3: New language features are increasingly used

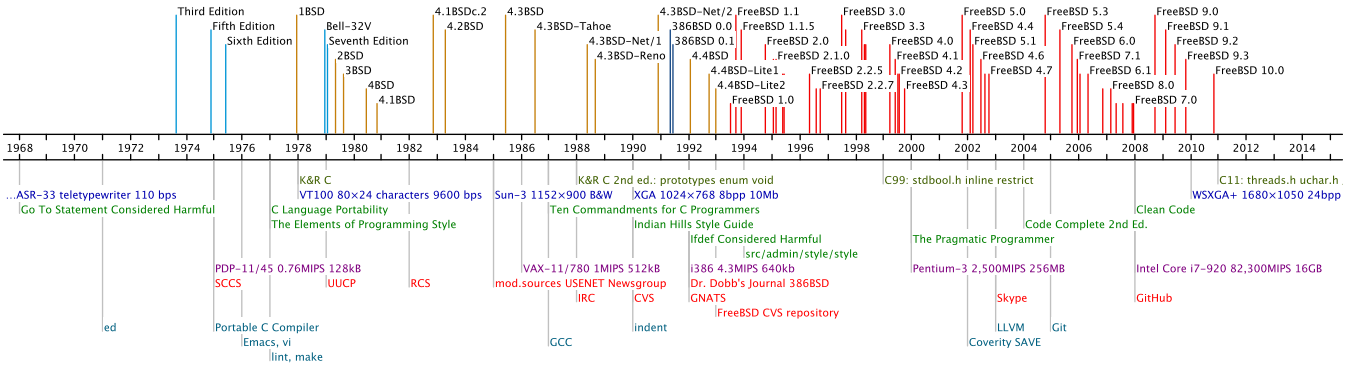

Figure 1: Timeline of indicative analyzed revisions and milestones in (from top to bottom): C language evolution, developer interfaces, programming guidelines, processing capacity, collaboration mechanisms, and tools.

to saturation point

The metrics DCONST, `const` keyword density ($R^2 = 0.97$); DENUM, `enum` keyword density ($R^2 = 0.66$); DINL, `inline` keyword density ($R^2 = 0.64$); DUNS, `unsigned` keyword density ($R^2 = 0.89$); DVOID, `void` keyword density ($R^2 = 0.93$); and DVOL, `volatile` keyword density ($R^2 = 0.86$), have in general climbed up over time.

The overall upward trend of `enum` and `inline` is overlayed on partial downward trends, resulting in a low R^2 figure. One could hypothesize that developers switch between enthusiasm and indifference regarding their use.

The DUNS metric seems to have retreated over the past few years. The reduced use could be attributed to two factors. The first may be the introduction of the `size_t` keyword in the 1999 ISO C standard [23]. This represents more descriptively the size of memory objects. For example, the argument to the `malloc` C library function is `unsigned` in 4.3BSD but is `size_t` in 4.4BSD. The second may have to do with the application of the `unsigned` keyword on integral data types to double the range of (positive) numbers they could hold. The introduction of 64-bit architectures and increases in memory capacity, which provide an integral data type that can hold numbers of up 10^{19} and sufficient memory space for its use, may have obsoleted this practice. Consequently, new code has fewer reasons to use the `unsigned` keyword, resulting in a relative decline of it use.

The metrics DNOAL, `noalias` keyword density, ($R^2 = 0.12$) and DSIGN, `signed` keyword density, ($R^2 = 0.04$), seem impervious to time, consistent with a lacklustre adoption. The `signed` keyword is mostly useful in cases one wants to store a signed value in a `char`-typed variable. Given current memory alignment restrictions and sizes, such a choice is nowadays both expensive and unnecessary. The rise of `noalias` appearances after the year 2000 is due to the use of `noalias` as a plain identifier, after the name had clearly lost its keyword status.

The metric DREST, `restrict` keyword density, ($R^2 = 0.39$), could have shown an upward trend and a higher R^2 were it not for the fact that during the period 1999–2007 it was defined and used as `__restrict` so that the code could easily work with compilers that did not support it.

The hypothesis is therefore validated with the proviso that the newly introduced language feature is widely useful to the programming community; if it is, its use will increase until it plateaus, presumably having reached a saturation point. The DNOAL and DSIGN metrics provide support to an alternative hypothesis: that language features are not adopted, presumably because they do not correspond to an important need. As Dennis Ritchie wrote to the X3J11 C standardization committee, language keywords should carry their weight; it is a mistake having keywords that "what they add to the cost of learning and using the language is not repaid in greater expressiveness" [44].

This finding indicates the importance of investing in sound language evolution. This is something that the C++, Fortran, and Java communities seem to have handled in a competent way, while the Python and Perl communities have mismanaged or neglected. Also languages such as C and Lisp may have reached their limits of significant evolution. It would therefore be interesting to investigate what effect the stewardship of language evolution has on a language's adoption and use.

H4: Programmers trust the compiler for register allocation

The metric DREG, `register` keyword density ($R^2 = 0.96$), has been steadily declining from around 1990. This may have been the point when compiler technology advanced sufficiently and developers really started trusting that the compiler would do a better job at register allocation than they could. At some point compilers began to ignore register declarations, forcibly taking over the task of register allocation. It seems that programmers are eager to slough off responsibility when they can afford to.

Actions associated with this hypothesis would be investment and research in technologies that can assist developers in a similar way. The characteristics we are looking at are minimal (ideally zero) involvement of the programmer, at least modest gains, and a very low downside risk. Obvious areas for such investment include static analysis to locate bugs, resource management, the utilization of multiple computing cores, optimization of cache and memory access patterns, and reduction of energy use.

H5: Code formatting practices converge to a common standard

The metrics FMINC, formatting inconsistency ($R^2 = 0.81$), and INDEV, indentation spaces standard deviation

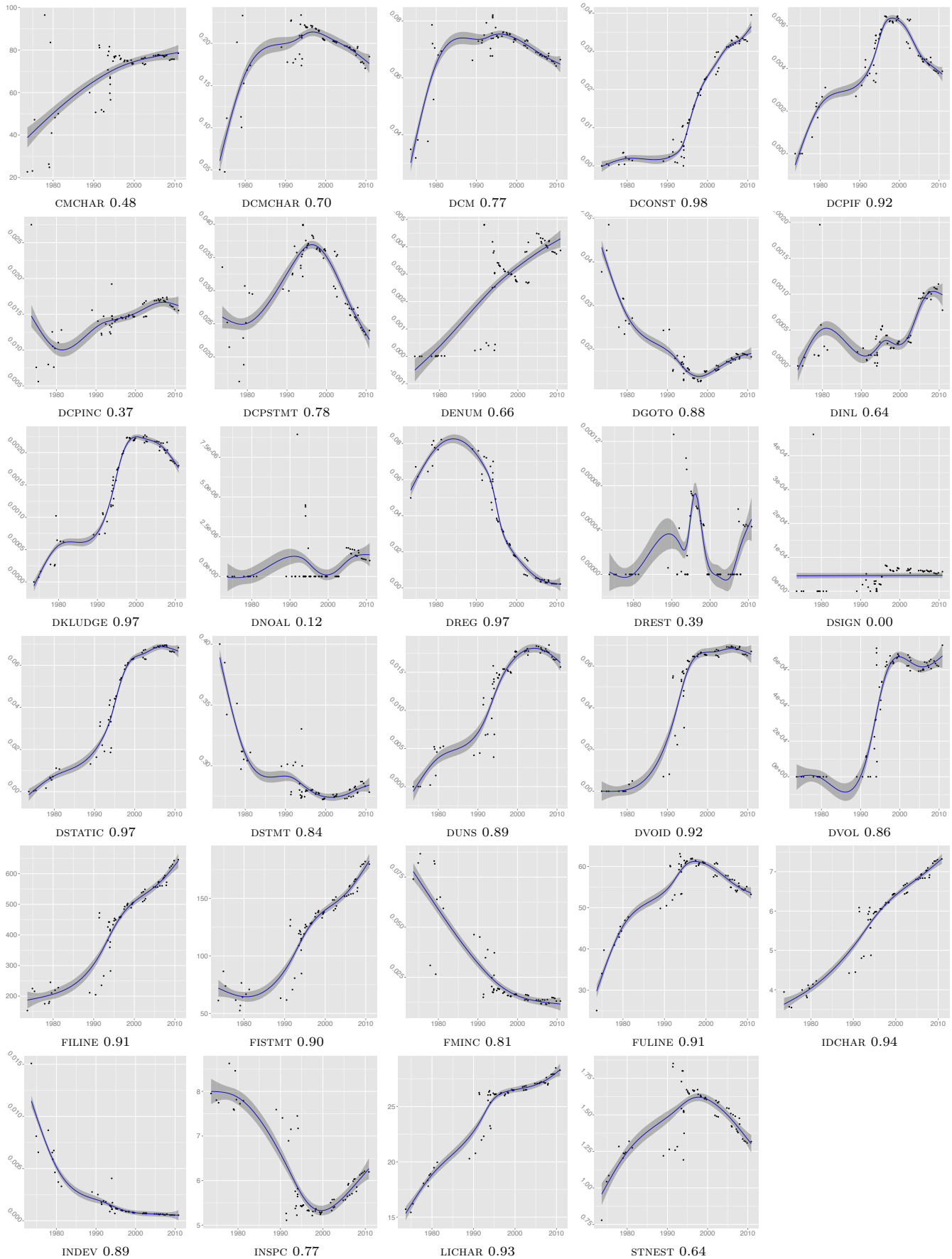

Figure 2: Long term evolution of programming practices. (Metrics appear in alphabetical order; the number following the metric name is adjusted R^2.)

($R^2 = 0.89$), have both been falling over time, providing strong support to the hypothesis that code formatting practices do converge, reducing inconsistencies and evolving towards a more homogeneous style.

Our findings seem to show an instance where an online community has successfully driven consensus and adoption of common practices. Research could examine drivers for other desirable trends, such as software security, unit testing, and operations support.

H6: Software complexity evolution follows self correction feedback mechanisms

The metrics FULINE, mean function length,  ($R^2 = 0.91$); STNEST, mean statement nesting,  ($R^2 = 0.64$); DCPIF, C preprocessor conditional statement density  ($R^2 = 0.92$); DCPSTMT, C preprocessor non-include statement density ($R^2 = 0.87$), all seem to follow a pattern where the metric grows indicating greater complexity, until a reverse trend kicks in and complexity starts falling. For example, FULINE and STNEST may have increased due to new ergonomic affordances (more lines can fit on a display), but their increase may have overreached regarding the programmers' capacity to comprehend the corresponding code. That is consistent with a view of software complexity evolution being guided through self correcting feedback. Perhaps controversially, the same can be argued for the evolution of DGOTO, `goto` keyword density  ($R^2 = 0.88$), whose value has started increasing after a long declining trend. It seems that programmers are loath to give up a maligned language construct, despite half a century of drilling against it. Perhaps `goto` is too valuable to let go; or letting it go results in code that is more complex than simply using it judiciously.

Our findings here are in agreement with several of Lehman's laws of software evolution [29], namely: *Increasing Complexity*, *Conservation of Familiarity*, *Declining Quality*, and *Feedback System*. The corresponding consequences of these laws have been investigated for many years [21], so we will not attempt to expand on this topic here.

H7: Code readability increases

We would expect code readability to increase until reaching a plateau, consistently with all programming advice. As proxies for code readability we have INSPC, mean indentation spaces  ($R^2 = 0.77$); DSTMT, statement density  ($R^2 = 0.84$); DCMCHAR, comment character density  ($R^2 = 0.70$); CMCHAR, mean comment size ($R^2 = 0.48$); DCM, comment density  ($R^2 = 0.77$); DKLUDGE, kludge word density  ($R^2 = 0.97$). Taken together, the metrics present a mixed, though mostly positive, picture. Comment density and comment character density follow a pattern where important improvements have started losing some ground. This could be related to the adoption of longer identifiers, which may make code more self-documenting. Statement density followed a consistent downwards trend, which is good, until well into the 2000s when it started edging upwards, which is worrying. The density of words indicating substandard code in comments has only recently started declining. The amount of whitespace used for indentation seems to be converging at or below the six spaces considered to be the point above which readability suffers. These metrics show that Unix code readability has improved significantly over the early years. However, they do not provide evidence that readability is still increasing. The future will tell us whether we are currently witnessing a regression toward the mean that corrects excessibe adherance to readability guidelines, or whether developers have lost interest in these mechanical aspects of readability.

Our failure to support this hypothesis may well indicate the limits on how far internal code quality can be improved through commenting, and, in general, the diminishing returns of investing in the code's documentary structure [60]. Consequently, research has to be directed into other areas that can drive internal code quality. Examples include more powerful programming structures, refactoring, specialized libraries, model-driven development, meta-programming, domain-specific languages, static analysis, and online collaboration platforms.

4. RELATED WORK

Work related to the study of the long term evolution of programming practices can be divided into that associated with software evolution, that looking at the actual programming practices, and that associated with our specific hypotheses.

The evolution of software has been the subject of decades of research [10]. A central theoretical underpinning regarding software's evolution are Lehman's eponymous laws as initially stated [27] and subsequently revised and extended [28–30]. We will not expand on the topic, because a recent extensive survey of it [21] provides an excellent historical overview and discusses the current state of the art. Along similar lines, a number of important studies focus on the stages of software growth [6, 31, 59], as well as software ageing [40] or decay [14].

One study particularly relevant to our work [1] examined how the vocabulary used in two software projects evolved over five and eight software versions respectively. The researchers found that identifiers faced the same evolution pressures as code. A related study [47] examined the changes in code convention violations in four open source projects. The authors report that code size and associated violations appear to grow together in a 100-commit window. Two studies have looked at the evolution of commenting practices over time: one [24] examined ProsgreSQL from 1996 to 2005, and one [19] Linux from 1994 to 1999. Both studies found comments to have remained roughly constant over time. The second study [19] also observed an increase in the size of the files. Compared to the preceding studies, our work examines a wider variety of metrics over a significantly longer time scale.

On the subject of programming practices McConnel's *Code Complete* [36] contains a full chapter (11) discussing the naming of variable names, and one (chapter 31) discussing code layout and style. Style guidelines can also be found in references [9, 17, 22, 26, 38, 55, 56, 61]. An interesting theoretical contribution on this front has been made by Van De Vanter [60], who has argued that all extralingual information added by programmers for aiding the human reader forms the code's *documentary* structure, which is orthogonal to the linguistic elements used by the compiler.

A substantial body of work has been performed on the subject of our hypotheses, though not on the long timescale we examined. Here we summarize the most important papers. Regarding **H1** (Programming practices reflect technology affordances) Leonardi [32] argues that the imbrication

metaphor (between human and material agencies) can help us to explain how the social and the material become interwoven and continue to produce improved infrastructures that people use to get their work done. On the subject of **H2** (Modularity increases with code size) Capiluppi et al. [11] found that — as one could expect — modularity is related to code size. They state that as a project grows, developers reorganize code adding new modules. Concerning **H3** (New language features are increasingly used to saturation point) Parnin et al. [41] explored how Java developers use new language features regarding generics. Interestingly, they found that developers may not replace old code with new language features, meaning that only the introduction of a new language feature is not enough to ensure adoption. On the topic of **H4** (Programmers trust the compiler for register allocation) Arnold [4] also found that the register keyword quickly became advisory, and that now all C/C++ compilers just ignore it. Regarding **H5** (Code formatting practices converge to a common standard) according to an empirical study conducted by Bacchelli and Bird at Microsoft [5], one of the most important tasks of code reviewers is to check if the code follows the team standards and conventions in terms of code formatting and in terms of function and variable naming. On the topic of **H6** (Software complexity evolution follows self correction feedback mechanisms) Terceiro et al. [58] also agree that system growth is not necessarily associated with structural complexity increases. Finally, as far as **H7** (Code readability increases) is concerned Buse and Weimer [7] propose a learning metric for code readability, which correlates strongly with three measures of software quality: code changes, automated defect reports, and defect log messages. Their study suggests that comments, in and of themselves, are less important than simple blank lines to local judgments of readability.

5. CONCLUSIONS AND FURTHER WORK

We started this paper writing that tracking long-term progress in a discipline allows us to take stock of things we have achieved, appreciate the factors that led to them, and set realistic goals for where we want to go. The results we obtained by examining the evolution of the Unix source code over a period of more than four decades paint a corresponding broad brush picture.

Over the years the developers of the Unix operating system appear to have: evolved their code style in tandem with advancements in hardware technology, promoted modularity to tame rising complexity, adopted valuable new language features, allowed compilers to allocate registers on their behalf, and reached broad agreement regarding code formatting. Our findings have also shown two other processes at play, namely self correcting feedback mechanisms associated with rising software complexity and documentary code structure improvements that have reached a ceiling.

The interplay between increased complexity, on the one hand, and technological affordances, on the other, may explain some of the inflection points we have witnessed. For example, technology has allowed developers to work with longer functions with ease, but longer functions are also more complex than shorter ones. We can speculate that when complexity trumped technology developers backed down and reverted to writing shorter, simpler functions. Time will show whether there will be another reversal in the future, with developers opting once more for more complexity. If so, a mechanism similar to that of other dynamical systems describing opposing and interactive forces may lurk behind the phenomenon.

Although we do not claim to have proven a causal relationship, it is plausible that the trends we observed are related to advances in tools, languages, ergonomics, guidelines, processing power, and conventions. So, even though we have not proposed an underlying model by which such advances translate to the changes in metrics we have observed, we believe it is very unlikely that they have not had an impact on software development.

Further investments in tools, languages, and ergonomics are likely to be progress drivers. On the other hand, further returns on investment in programming guidelines and conventions are unlikely to match those of the past. The structure and forces of the IT market guarantee that additional processing power is likely to reach developers without requiring significant interventions. The fact that some metrics have retreated or reached a plateau indicates that this is no longer enough to increase quality further. In addition, the looming end of progress in semiconductor manufacturing driven by Moore's Law [35] means that corresponding advancement of software development practices based on its fruits will also stall; we will need to come up with other, more frugal or different, sources of innovation.

Social coding, as performed on GitHub, and cloud computing come to mind, but other new ways to drive progress surely remain to be discovered.

The study of long term programming practice evolution can be expanded on a number of fronts. An important task, to which we alluded above, would be to examine and establish causality regarding the factors affecting the trends. This could be helped by performing regression with the underlying factor (e.g., screen resolution) rather than time as the independent variable. Based on these results one could then perform meaningful cost-benefit analysis regarding investments in software development infrastructure. Moreover, the accuracy of the analysis can be improved by studying a period's specific code changes, rather than complete snapshots, which mostly incorporate legacy code. This could be done by employing qualitative rather than quantitative methods. Such a study could also provide insights regarding the various inflection points we observed in the 1990s and early 2000s. Are these inflections the result of reaching (after several decades) a limit regarding complexity and therefore regressing to the mean, or did the change involve a catalyst? Furthermore, in addition to code, it would also be valuable to perform a quantitative examination of other process-related inputs, such as configuration management entries, the issues' lifecycle, and team collaborations. Finally, many more systems should be studied in order to validate the generalisability of our results.

Acknowledgements

John Pagonis provided a number of insightful comments on earlier versions of this document.

This research has been co-financed by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thalis — Athens University of Economics and Business — Software Engineering Research Platform.

6. REFERENCES

[1] S. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol. Analyzing the evolution of the source code vocabulary. In *CSMR '09: 13th European Conference on Software Maintenance and Reengineering*, pages 189–198, March 2009.

[2] American National Standard for Information Systems — programming language — C: ANSI X3.159–1989, Dec. 1989. (Also ISO/IEC 9899:1990).

[3] O. Arafat and D. Riehle. The commenting practice of open source. In *OOPSLA '09: 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 857–864, New York, NY, USA, 2009. ACM.

[4] K. Arnold. Programmers are people, too. *Queue*, 3(5):54–59, June 2005.

[5] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *ICSE '13: 35th International Conference on Software Engineering*, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[6] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM.

[7] R. Buse and W. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010.

[8] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *CSMR '10: 14th European Conference on Software Maintenance and Reengineering*, pages 156–165, March 2010.

[9] L. W. Cannon, R. A. Elliott, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, H. Spencer, D. Keppel, and M. Brader. Recommended C style and coding standards. Available online http://sunland.gsfc.nasa.gov/info/cstyle.html (January 2006). Updated version of the Indian Hill C Style and Coding Standards paper.

[10] A. Capiluppi. Models for the evolution of OS projects. In *ICSM '03: International Conference on Software Maintenance*, pages 65–74, 2003.

[11] A. Capiluppi, P. Lago, and M. Morisio. Characteristics of open source projects. In *7th European Conference on Software Maintenance and Reengineering*, pages 317–327, March 2003.

[12] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–101, June 1982.

[13] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, Mar. 1968.

[14] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan. 2001.

[15] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, Dec. 2002.

[16] B. Fluri, M. Wursch, and H. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *WCRE 2007: 14th Working Conference on Reverse Engineering*, pages 70–79, Oct 2007.

[17] The FreeBSD Project. *Style—Kernel Source File Style Guide*, Dec. 1995. FreeBSD Kernel Developer's Manual: style(9). Available online http://www.freebsd.org/docs.html (January 2006).

[18] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *PLDI '12: Programming Language Design and Implementation*, pages 323–334, 2012.

[19] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *CSMR '00: International Conference on Software Maintenance*, pages 131–142, 2000.

[20] G. Gousios and D. Spinellis. Conducting quantitative software engineering studies with Alitheia Core. *Empirical Software Engineering*, 19(4):885–925, Aug. 2014.

[21] I. Herraiz, D. Rodriguez, G. Robles, and J. M. González-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys*, 46(2), Nov. 2013.

[22] G. Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, June 2006.

[23] International Organization for Standardization. *Programming Languages — C*. ISO, Geneva, Switzerland, 1999. ISO/IEC 9899:1999.

[24] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in PostgreSQL. In *MSR '06: 2006 International Workshop on Mining Software Repositories*, pages 179–180, New York, NY, USA, 2006. ACM.

[25] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Boston, MA, second edition, 2002.

[26] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, second edition, 1978.

[27] M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1979.

[28] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, 1985.

[29] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[30] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution — the nineties view. In *METRICS '97: 4th International Symposium on Software Metrics*, pages 20–32, Washington, DC, USA, 1997. IEEE Computer Society.

[31] F. Lehner. Software life cycle management based on a phase distinction method. *Microprocessing and Microprogramming*, 32(1OCo5):603 – 608, 1991. Euromicro symposium on microprocessing and microprogramming.

[32] P. M. Leonardi. When flexible routines meet flexible technologies: Affordance, constraint, and the

imbrication of human and material agencies. *MIS Quarterly*, 35(1):147–168, Mar. 2011.

[33] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *AOSD '11: 10th International Conference on Aspect-Oriented Software Development*, pages 191–202, 2011.

[34] P. E. Livadas and D. T. Small. Understanding code containing preprocessor constructs. In *IEEE Third Workshop on Program Comprehension*, pages 89–97, Nov. 1994.

[35] C. Mack. Fifty years of Moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, May 2011.

[36] S. C. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, second edition, 2004.

[37] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, 26(11):861–867, Nov. 1983.

[38] T. Misfeldt, G. Bumgardner, and A. Gray. *The Elements of C++ Style*. Cambridge University Press, Cambridge, 2004.

[39] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *MSR '09: 6th IEEE International Working Conference on Mining Software Repositories*, pages 11–20, Washington, DC, USA, 2009. IEEE Computer Society.

[40] D. L. Parnas. Software aging. In *ICSE '94: 16th International Conference on Software Engineering*, pages 279–287, Washington, DC, May 1994. IEEE Computer Society.

[41] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: How new features are introduced, championed, or ignored. In *MSR '11: 8th Working Conference on Mining Software Repositories*, pages 3–12, New York, NY, USA, 2011. ACM.

[42] D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *MSR '11: 8th Working Conference on Mining Software Repositories*, pages 73–82, New York, NY, USA, 2011. ACM.

[43] E. S. Raymond. *The Art of Unix Programming*. Addison-Wesley, 2003.

[44] D. M. Ritchie. noalias comments to X3J11. Usenet Newsgroup comp.lang.c, Mar. 1988. Message-ID: 7753@alice.UUCP. Available online https://groups.google.com/forum/message/raw?msg=comp.lang.c/K0Cz2s9il3E/YDyo_xaRG5kJ.

[45] D. M. Ritchie. The development of the C language. *ACM SIGPLAN Notices*, 28(3):201–208, Mar. 1993. Preprints of the History of Programming Languages Conference (HOPL-II).

[46] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):255–265, 1975.

[47] M. Smit, B. Gergel, H. Hoover, and E. Stroulia. Code convention adherence in evolving software. In *ICSM '11: 27th IEEE International Conference on Software Maintenance*, pages 504–507, Sept 2011.

[48] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. In R. Adams, editor, *Proceedings of the Summer 1992 USENIX Conference*, pages 185–198, Berkeley, CA, June 1992. USENIX Association.

[49] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, Nov. 2003.

[50] D. Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley, Boston, MA, 2006.

[51] D. Spinellis. A tale of four kernels. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE '08: 30th International Conference on Software Engineering*, pages 381–390, New York, May 2008. Association for Computing Machinery.

[52] D. Spinellis. A repository with 44 years of Unix evolution. In *MSR '15: 12th Working Conference on Mining Software Repositories*, pages 13–16. IEEE, 2015. Best Data Showcase Award.

[53] D. Spinellis. Tools and techniques for analyzing product and process data. In T. Menzies, C. Bird, and T. Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 161–212. Morgan-Kaufmann, 2015.

[54] D. Spinellis, P. Louridas, and M. Kechagia. An exploratory study on the evolution of C programming in the Unix operating system. In Q. Wang and G. Ruhe, editors, *ESEM '15: 9th International Symposium on Empirical Software Engineering and Measurement*, pages 54–57. IEEE, Oct. 2015.

[55] R. Stallman et al. GNU coding standards. Available online http://www.gnu.org/prep/standards/ (January 2006), Dec. 2005.

[56] Sun Microsystems, Inc. Java code conventions. Available online http://java.sun.com/docs/codeconv/ (January 2006), Sept. 1997.

[57] O. Tange. GNU parallel: The command-line power tool. *;login:*, 36(1):42–47, Feb. 2011.

[58] A. Terceiro, M. Mendonça, C. Chavez, and D. Cruzes. Understanding structural complexity evolution: A quantitative analysis. In *CSMR '12: 16th European Conference on Software Maintenance and Reengineering*, pages 85–94, March 2012.

[59] M. van Genuchten and L. Hatton. Quantifying software's impact. *Computer*, 46(10):66–72, 2013.

[60] M. L. V. D. Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, 2002. Special Issue on Source Code Analysis and Manipulation (SCAM).

[61] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur, and P. Thompson. *The Elements of Java Style*. Cambridge University Press, Cambridge, 2000.

[62] S. Wood. *Generalized Additive Models: An Introduction with R*. Chapman and Hall/CRC, 2006.

[63] S. N. Wood. Fast stable restricted maximum likelihood and marginal likelihood estimation of semiparametric generalized linear models. *Journal of the Royal Statistical Society (B)*, 73(1):3–36, 2011.