An Exploratory Study on the Evolution of C Programming in the Unix Operating System

Diomidis Spinellis, Panagiotis Louridas, and Maria Kechagia
Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece
Email: {dds,louridas,mkechagia}@aueb.gr

***Abstract*—Context: Numerous factors drive long term progress in programming practices. Goal: We study the evolution of C programming in the Unix operating system. Method: We extract, aggregate, and synthesize metrics from 66 snapshots obtained from an artificial software configuration management repository tracking the evolution of the Unix operating system over four decades. Results: C language programming practices appear to evolve over long term periods; our study identified some continuous trends with highly significant coefficients of determination. Many trends point toward increasing code quality through adherence to numerous programming guidelines, while some others indicate adoption that has reached maturity. In the area of commenting progress appears to have stalled. Conclusions: Studying the long term evolution of programming practices identifies areas where progress has been achieved along an expected path, as well as cases where there is room for improvement.**

I. Introduction

Tracking long-term progress in engineering 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. Specific factors that drive long term progress in programming practices include the affordances and requirements of computer architecture, programming languages, development frameworks, and 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-optimizations, the writing of more descriptive code, the choice of appropriate encapsulation mechanisms, and the convergence toward a common coding style. The objective of this work is to study, in view of these factors, the evolution of programming practice in the context of the Unix operating system.

Many of the practices we outlined can be automatically measured by examining instances of the code over time. By looking at the long term evolution of specific metrics we can determine whether they indeed change over time, as well as the direction and rate of change.

ESEM '15: 9th International Symposium on Empirical Software Engineering and Measurement, Beijing, China.

The results of the study on long term evolution of programming practices can be used to allocate the investment of additional 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 artifact will represent first and foremost the engineering requirements of its construction [1], the results can also indicate areas where developers rationally allocated improvement effort and areas where developers did not see a reason to invest.

II. 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 source code with a custom-developed tool to extract a variety of metrics for each file. We then synthesize these metrics into values that are related to the internal code quality of the whole system, and analyze the results over time using established statistical techniques.

The primary sources of the material 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 the CSRG SCCS repository, the FreeBSD 1 CVS repository, and the Git mirror of modern FreeBSD development. This material 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 [2].

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 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 [3], and will therefore produce reasonably accurate results without requiring access to header files and the like. The *cmcalc* tool calculates size, language feature, code style, and commenting

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

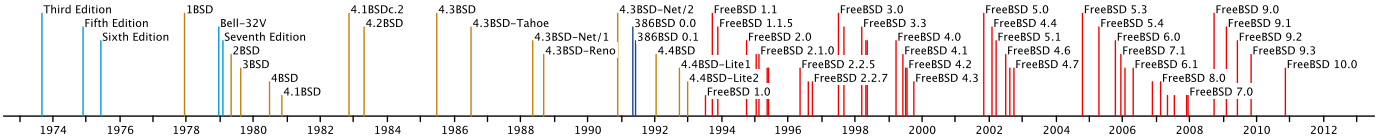

Fig. 1. Mean file date of C source code files in the examined Unix releases

metrics; see the tool's documentation and reference [4] for more details.

We collected the metrics by iterating through 66 Unix releases (see Figure 1), starting from the *Third Research Edition* (1973) and ending with FreeBSD 10.0.0 (2014). For each release we checked out the code and run *cmcalc* on all its C files. Through this process we collected 490 thousand records containing in total 50 million values.

We aggregated the raw metrics at the level of each release and calculated derivative values needed to track the long term evolution of programming practices. The derived aggregate metrics are listed in Table I, roughly clustered into those affected by programming language evolution (L), ergonomics (workstation technology and screen resolution—E), programming guidelines (G), processing capacity (P), conventions (C), and tools (T).

To make metrics comparable across releases, we replaced the absolute numbers in the primary collected metrics with corresponding density figures in the derived ones. The denominator for calculating each density depends on the numerator unit, and can be the corresponding number of files, lines, characters, statements, or identifiers. For example, the *comment character density* is the ratio of comment characters to all characters across all source code files.

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 I.

The length of files (M1) and the corresponding functionality they provide (M2) 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 minimize the impact on build performance on changes. 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 (M3—lines should not exceed the number of characters that can fit in a display row), function length (M4—functions should fit on a screen), identifier length (M5—these should be descriptive rather than cryptically short), and statement nesting (M6—deep nesting should be avoided). On the other hand, all these are also affected by ergonomics. Higher resolution screens can display more characters per row, more rows on a screen, and 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. The level of nesting and available screen real estate can also affect the number of spaces used for indentation. Also, IDEs with code completion can help writing long identifiers, while optimizing compilers can avoid the function call cost through inlining.

The declaration of identifiers that are only internally visible within the compilation unit (`static`—M8), 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`—M9, `enum`—M10, `unsigned`—M11, `signed`—M12, `register`—M13, `void`—M14, `volatile`—M15) is made possible through their introduction as language features. In addition, the use of some (`const`, `enum`, `unsigned`) can clarify the programmer's intent and avoid some errors. Furthermore, when these three keywords are used, compilers can use static analysis to identify common error cases. On the other hand, the `register` and `volatile` keywords are there to address deficiencies in the way compilers allocate registers and detect aliasing, so their use should become less common as technology advances.

The formatting style (M16) and the number of spaces used for indentation (M7, M17) are a matter of convention. Consistency in both areas can also be aided through tools, such as code formatters, editors, and IDEs.

The size and density of comments (M20, M19, M21) and the density of statements (M18) are also a matter of guidelines and ergonomics. 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.

The problems associated with the use of the C preprocessor are well known [5], [6], and most guidelines advocate the avoidance of macro definitions (M24) and conditional compilation (M22). Conversely, guidelines also advocate the use of header files (and the `#include` directive—M23) in order to promote code modularity and portability.

C preprocessor macros can be 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 `goto` statement (M25) has been considered harmful for almost half a century [7].

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 linear

TABLE I
DERIVED AGGREGATE METRICS, FACTORS AFFECTING THEM, COEFFICIENT OF DETERMINATION, AND OBSERVED TREND

Metric identifier and description		L	E	G	P	C	T	R^2	Scatterplot	Trend
M1	Mean file length (lines)			✓	✓			0.954		
M2	Mean file functionality (statements)			✓	✓			0.930		
M3	Mean line length (characters)		✓	✓				0.853		
M4	Mean function length (lines)		✓	✓				0.396		
M5	Mean identifier length		✓	✓			✓	0.964		
M6	Mean statement nesting		✓	✓	✓		✓	0.186		
M7	Mean indentation spaces		✓					0.511		
M8	internally visible declaration density	✓		✓			✓	0.942		
M9	`const` keyword density	✓		✓			✓	0.874		
M10	`enum` keyword density	✓		✓			✓	0.740		
M11	`unsigned` keyword density	✓		✓				0.873		
M12	`signed` keyword density	✓						0.029		
M13	`register` keyword density	✓					✓	0.850		
M14	`void` keyword density	✓						0.833		
M15	`volatile` keyword density	✓					✓	0.754		
M16	Formatting inconsistency					✓	✓	0.689		
M17	Indentation spaces standard deviation					✓	✓	0.615		
M18	Statement density		✓	✓		✓		0.517		
M19	Comment character density		✓	✓				0.404		
M20	Mean comment size		✓	✓				0.618		
M21	Comment density		✓	✓				0.122		
M22	C preprocessor conditional statement density			✓				0.369		
M23	C preprocessor include statement density			✓	✓		✓	0.252		
M24	C preprocessor non-include statement density	✓		✓	✓		✓	0.000		
M25	`goto` keyword density			✓				0.408		

regression analysis with the days elapsed since the first release as the independent variable and each metric as the dependent variable. We chose to use the Ordinary Least Squares method as we are treating each variable independently of the others. Our statistical model is a linear one $y = a + bx$, trying to capture straightforward, upwards or downwards trends in the evolution of the metrics.

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. Given the system's continued prevalence and importance, the lack of generalizability is not a showstopper.

III. RESULTS AND DISCUSSION

The results of the statistical analysis are summarized in the rightmost columns of Table I.

In sum, exploratory data analysis showed some continuous trends with highly significant coefficients of determination, as well as trends that after some period of time level off, and others that have a segmented structure. The last two categories obviously have considerably lower linear regression coefficients of determination, but are nevertheless interesting. Analysing them with segmented regression methods could provide additional weight to the significance of their shape.

The evolution of many programming practices seems to be correspondingly aligned over time with the factors we outlined in Section II. Specifically, increasing identifier (M5), line (M3), function (M4), and file lengths (M1, M2) match improved programming facilities, the increasing density of most examined keywords (M9, M10, M11, M12, M14, M15) indicates that language facilities are adopted, and the fall in the use of the `register` keyword (M13) matches improved compiler technology.

Happily, many trends point toward increasing code quality through adherence to the following guidelines: longer identifiers (M5), more declarations with internal visibility (M8), increased inclusion of header files (M23), falling statement nesting (M6—from the mid 1990s onward), the reduction of non-include preprocessor directives (M24—in the past couple of decades), reduced formatting inconsistency (M16), and increased use of enumerations and constants (M10, M9). These trends may also indicate that there could be underlying pressure forces, such as increased code complexity and more strictly enforced coding standards, that drive the changes.

On the other hand, some plateaus and reverse trends are worrisome. These include the stagnant size of comments (M20), the falling comment density (M21, M19), and the

slightly rising use of the `goto` statement (M25). Reverse trends may be examples of over-enthusiasm regressing to the mean, while plateaus could indicate entering a phase of diminishing returns. Similarly problematic is the lack of a clear downward trend in the (ab)use of the C preprocessor (M22, M24). It seems that this particular tool is simply too valuable to let go. On the other hand some plateaus are simply a sign that the corresponding area has reached a steady state associated with maturity. These include: the use of some C keywords (M14, M15), formatting inconsistencies (M16), the standard deviation of indentation spaces (M17), and statement density (M18).

IV. Related Work

The evolution of software has been the subject of decades of research [8]. A central theoretical underpinning regarding software's evolution are Lehman's eponymous laws [9]. We will not expand on the topic, because a recent extensive survey of it [10] 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 [11], as well as software aging [12], or decay [13].

One study particularly relevant to our work [14] 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 [15] 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. A third study [16] looked at commenting practices in ProsgreSQL from 1996 to 2005 and found that the percentage of functions with comments remained roughly constant over time. Compared to these studies, our work examines a wider variety of metrics over a significantly longer time scale.

V. Conclusions and Further Work

Programming practices appear to evolve over long term periods: our study identified some continuous trends with highly significant coefficients of determination. The evolution of many programming practices seems to be aligned over time with language features, ergonomics, programming guidelines, processing capacity, conventions, and tools. Many trends point toward increasing code quality through adherence to numerous guidelines, while some others indicate that adoption has reached maturity. On the other hand in the area of commenting progress appears to have stalled or reversed.

In general, the results show that language-related features are easily adopted and quick to pay off. They also demonstrate that it is difficult to affect change through preaching on, say, the value of comments, or the risks of C preprocessor abuse. These two findings taken together suggest that gradually deprecating language features, as is done for example by the Java community, may be a powerful way to drive progress.

Work in the area of long term programming practice evolution can be expanded on a number of fronts. An important task 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. Moreover, the accuracy of the analysis can be improved by studying a period's specific code changes, rather than complete snapshots, which also incorporate legacy code. Also, the segmented trends can be formally examined using segmented analysis. 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 issue lifecycle, and team collaborations. Finally, many more systems should be studied in order to validate the generalizability of the results.

Acknowledgements

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.

References

[1] D. Spinellis, "A tale of four kernels," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. New York: Association for Computing Machinery, May 2008, pp. 381–390.

[2] ——, "A repository with 44 years of Unix evolution," in *MSR '15: Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 13–16.

[3] ——, "Tools and techniques for analyzing product and process data," in *The Art and Science of Analyzing Software Data*, T. Menzies, C. Bird, and T. Zimmermann, Eds. Morgan-Kaufmann, 2015, to appear.

[4] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Boston, MA: Addison-Wesley, 2002.

[5] H. Spencer and G. Collyer, "#ifdef considered harmful or portability experience with C news," in *Proceedings of the Summer 1992 USENIX Conference*, R. Adams, Ed. Berkeley, CA: USENIX Association, Jun. 1992, pp. 185–198.

[6] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, Dec. 2002.

[7] E. W. Dijkstra, "Go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968.

[8] A. Capiluppi, "Models for the evolution of OS projects," in *ICSM '03: International Conference on Software Maintenance*, 2003, pp. 65–74.

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

[10] 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*, vol. 46, no. 2, Nov. 2013.

[11] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 73–87.

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

[13] 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 Trans. Softw. Eng.*, vol. 27, no. 1, pp. 1–12, Jan. 2001.

[14] 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*, March 2009, pp. 189–198.

[15] 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*, Sept 2011, pp. 504–507.

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