Noname manuscript No.
(will be inserted by the editor)

High-MCC Functions in the Linux Kernel

Ahmad Jbara · Adam Matan · Dror G. Feitelson

the date of receipt and acceptance should be inserted later

Abstract McCabe's Cyclomatic Complexity (MCC) is a widely used metric for the complexity of control flow. Common usage decrees that functions should not have an MCC above 50, and preferably much less. However, the Linux kernel includes more than 800 functions with MCC values above 50, and over the years 369 functions have had an MCC of 100 or more. Moreover, some of these functions undergo extensive evolution, indicating that developers are successful in coping with the supposed high complexity. Functions with similarly high MCC values also occur in other operating systems and domains, including Windows. For example, the highest MCC value in FreeBSD is 1316, double the highest MCC in Linux.

We attempt to explain all this by analyzing the structure of high-MCC functions in Linux and showing that in many cases they are in fact well-structured (albeit we observe some cases where developers indeed refactor the code in order to reduce complexity). Moreover, human opinions do not correlate with the MCC values of these functions. A survey of perceived complexity shows that there are cases where high MCC functions were ranked as having a low complexity. We characterize these cases and identify specific code attributes such as the diversity of constructs (not only a `switch` but also `if`s) and nesting that correlate with discrete increases in perceived complexity.

These observations indicate that a high MCC is not necessarily an impediment to code comprehension, and support the notion that complexity cannot be fully captured using simple syntactic code metrics. In particular, we show that regularity in the code (meaning repetitions of the same pattern of control structures) correlates with low perceived complexity.

Keywords Software Complexity · McCabe Cyclomatic Complexity · Linux Kernel · Perceived Complexity · Code Regularity

A. Jbara
School of Computer Science and Engineering, Hebrew University, 91904 Jerusalem, Israel.
E-mail: ahmadjbara@cs.huji.ac.il

A. Matan
School of Computer Science and Engineering, Hebrew University, 91904 Jerusalem, Israel.

D. G. Feitelson
School of Computer Science and Engineering, Hebrew University, 91904 Jerusalem, Israel.
E-mail: feit@cs.huji.ac.il

1 Introduction

Mitigating complexity is of pivotal importance in writing computer programs. Complex code is hard to write correctly and hard to maintain, leading to more faults [22,5]. As a result, significant research effort has been expended on defining code complexity metrics and on methods to combine them into effective predictors of code quality [31,9,32]. Industrial testimony indicates that using complexity metrics provides real benefits over simple practices such as just counting lines of code (e.g. [20,8,21,35]).

One early metric that has been used in many studies is McCabe's Cyclomatic Complexity (MCC) [25]. This metric essentially counts the number of linear paths through the code (the precise definition is given below in Section 2). In the original paper, McCabe suggests that procedures with an MCC value higher than 10 should be rewritten or split in order to reduce their complexity, and other somewhat higher thresholds have been suggested by others (e.g. [28,40,41,44,7]). In general, proposed thresholds are typically well below 50, and there appears to be some agreement that procedures with much higher values are extremely undesirable.

Nevertheless, in the context of a study of Linux evolution, we have found functions with MCC values in the hundreds [16]. This chance discovery led to a set of research questions:

1. What are the basic characteristics of high-MCC functions? Specifically,
   1.1 How common are such high-MCC functions? In other words, are they just a fluke or a real phenomenon reflecting the work practices of many developers?
   1.2 What causes the high MCC counts? One may speculate that they are the result of large flat `switch` statements, that do not reflect real complexity. But if other more complex and less regular constructs are found this raises the question of how developers cope with them.
   1.3 Does MCC correlate with other metrics, as has been shown in the past? Or does it provide independent complexity information?
2. Do high-MCC functions evolve with time? If these functions are "write once" functions that serve some fixed need and are never changed, then nobody except the original author really needs to understand them. But if they are modified many times as Linux continues to evolve, it intensifies the question of how do the maintainers cope with the supposedly high complexity.
3. What influences the perception of complexity? Specifically,
   3.1 Does a high MCC correlate with perceived complexity? In other words, does MCC indeed capture the essence of complexity?
   3.2 Can we find discrete elements of complexity? In other words, can we point out specific code attributes that, if present, make a function appear more complex? This is an extremely important question with respect to complexity metrics, as an affirmative answer may indicate that complexity is an additive property of code attributes.
   3.3 Is a visual representation of high-MCC functions better than code listings?
4. What other ingredients of complexity may be missing from MCC? In particular, in our work we found that some high-MCC functions have a very regular structure. This raised the question whether regularity may counteract the supposed complexity reflected by the high MCC.
5. Are all the high-MCC functions we found really required, or can some of this code be replaced or refactored? This issue reflects the tradeoff done by developers, where sometimes allowing additional code with high complexity metrics is nevertheless considered better than trying to minimize it.

6. Are the high-MCC functions unique to Linux, or do they also appear in other operating systems and domains?
7. Altogether, do the high-MCC functions indicate code quality problems with the Linux kernel?

To gain insight into these issues we analyzed the functions in Linux kernel version 2.6.37.5 that have MCC$\geq$100, which turn out to have MCC values ranging up to 587—way above the scale that is considered reasonable. We also analyzed the evolution of all 369 functions that had MCC$\geq$100 in any of the Linux kernel versions released since the initial release of version 1.0 in 1994 (more than a thousand versions). In addition we examined three other operating systems and three systems from different domains — Windows Research Kernel, OpenSolaris, FreeBSD, GCC, Firefox, and OpenSSL — and found that they also contain similar high-MCC functions. The highest MCC values were 246, 506, 1316, 1301, 699, and 371 respectively.

In a nutshell, we found that (in Linux) the most common source of high MCC counts is large trees of if statements, although several cases are indeed attributed to large switchs. 33% of the functions do not change, but the others may change considerably. About 5% of the functions exhibit extreme changes in MCC values that reflect explicit modifications to their design, indicating active work to reduce complexity. We speculate that the ability to work with these functions stems from the fact that switchs and large trees of ifs embody a separation of concerns, where each call to the function only selects a small part of the code for execution. This is especially true if they are nested in each other, rather than coming one after the other, so this explanation is especially relevant for the deeply-nested functions. On the other hand we also observed some cases of spaghetti-style gotos, which are not directly measured by MCC. Such observations motivate studying alternative ways in which code structure may be analyzed when assessing the resulting complexity. In particular, we suggest code regularity as an important attribute that may compensate for complexity.

The remainder of the paper is structured as follows. In the next section we define MCC and review its use. We characterize high-MCC functions in the Linux kernel in Section 3, and their evolution in Section 4. Results of the survey of perceived complexity are presented in Section 5, and the relationship with regularity in Section 6. Section 7 discusses the possibility of reducing high-MCC code. High-MCC functions in other operating systems and domains are examined in Section 8. Discussion, significance of our findings, and further research directions are presented in Section 9. This paper is an extended version of a previous conference paper [19]. The main additions are added experimentation (more subjects and additional experiments), the definition of a metric for code regularity and its effect, an examination of evidence for cloning, and showing that high-MCC functions exist also in other operating systems and domains.

2 McCabe's Cyclomatic Complexity

McCabe's cyclomatic complexity (MCC) is based on the graph theoretic concept of cyclomatic number, applied to a program's control-flow graph. The nodes of such a graph are basic blocks of code, and the edges denote possible control flow. For example, a block with an if statement will have two successors, representing the "then" option and the "else" option. The cyclomatic number of a graph g is

$$V(g) = e - n + 2p$$

where n is the number of nodes, e the number of edges, and p the number of connected components. (In a computer program, each procedure would be a separate connected component, and the end result is the same as adding the cyclomatic numbers of all of them.) McCabe suggested that the cyclomatic number of a control-flow graph represents the complexity of the code [25]. He also showed that it corresponds to the number of linearly independent code paths, and can therefore be used to set the minimal number of tests that should be performed.

Another way to characterize the cyclomatic number of a graph is related to the notions of structured programming, where all constructs have single entry and exit points. The control-flow graph is then planar, and the cyclomatic number is equal to the number of faces of the graph, including the "outside" area. McCabe also demonstrated a straight-forward intuitive meaning of the metric: it is equal to the number of condition statements in the program plus 1 (`if`, `while`, etc.). If conditions are composed of multiple atomic predicates, we could also count them individually; this is sometimes called the "extended" MCC [29]. Note that MCC counts points of divergence, but not joins. It is thus insensitive to unconditional jumps such as those induced by `goto`, `break`, or `return`.

2.1 Thresholds on MCC

In principle MCC is unbounded, and intuition suggests that high values reflect potentially problematic code. It is therefore natural to try and define a threshold beyond which code should be checked and maybe modified. McCabe himself, in the original paper which introduced MCC, suggests a threshold of 10 [25], and this is also the value used by the code analysis tool sold by his company today [26]. The Eclipse Metrics plugin also uses a threshold of 10 by default, and suggests that the method be split if it is exceeded [34]. VerifySoft Technology suggest a threshold of 15 per function, and 100 per file [44]. Logiscope also uses a threshold of 15 [41]. The STAN static analysis tool gives a warning at 15, and considers values above 20 an error [27]. The complexity metrics module of Microsoft Visual Studio 2008 reports a violation of the cyclomatic complexity metric for values of more than 25 [28]. The Carnegie Mellon Software Engineering Institute defined a four-level scale as part of their (now legacy) Software Technology Roadmap [40]. High risk was associated with values of MCC above 20, and very high risk with values larger than 50. Heitlager et al. used these risk levels and suggested a complexity rating scheme based on the percentage of LOC falling within each risk level [13].

All the above thresholds consider functions in isolation. VerifySoft also suggests a threshold on the sum of all functions in the same file. An alternative approach is to consider the distribution of MCC values. The Gini coefficient, used to measure inequality in economics, was used by Vasa et al. to characterize the distribution of different metrics including MCC [43]; he found that the distribution was highly skewed, as we do too. Stark et al. propose a decision chart that plots the cumulative distribution function (CDF) of MCC values on a logarithmic scale, and if the CDF falls below a certain diagonal line then the project as a whole should be reviewed [42]; in brief, this line requires 20% of the functions to have an MCC of 1, allows about 60% to be above 10, and dictates an upper bound of 90. However, it seems that this was not picked up by others, and using simple thresholds remains the prevailing approach.

2.2 Critique of MCC and Correlation with LOC

It should be noted that MCC is not universally accepted as a good complexity metric, and it has been challenged on both theoretical and experimental grounds.

Perhaps the most common objection to using MCC as a complexity metric is its strong correlation with lines of code (LOC) [36,37,14]. This correlation has been demonstrated many times, and indeed, we find that also in the Linux kernel the correlation coefficient of MCC and LOC is a relatively high 0.88. But if we focus on only the high-MCC functions, the correlation is much lower. We revisit this issue in Section 3.4.

Ball and Larus note that with n predicates there can be between $n+1$ and 2^n paths in the code, so the number of paths is a better measure of complexity than the number of predicates [3]. Others show that MCC only measures control flow complexity but not data flow complexity and has additional deficiencies [36,37]. In particular, MCC is intrinsic to code, so it does not admit the possibility that code fragments interact with each other to either increase or decrease the overall complexity [45]. Finally, Nagappan et al. have shown that while MCC is a good defect predictor for some projects, there is no single metric (including MCC) that is good for all projects [30].

There is, however, no other complexity metric that enjoys wider acceptance and is free of such criticisms, so MCC remains widely used to this day. Oman's ‘maintainability index’ includes MCC as one of its components [33], and Baggen et al. recently used thresholds on MCC in the context of creating a certification mechanism for maintainability [2]. Curtis et al. use a criterion of MCC above 30 to identify ‘highly complex components’, and find that MCC is one of the four most frequent violations of good architectural or coding practice over different languages [7]. The ‘weighted method count’ metric for object-oriented software is usually interpreted as the sum of the MCC over all methods in a class. Recently, Capiluppi et al. used MCC to evaluate the change in complexity of successive revisions of the same file in the Linux kernel [6], and Soetens et al. used it to check the assumption that refactoring reduces complexity (as it turns out, most refactoring does not affect MCC) [38]. Thus, given its wide use and availability in software development and testing environments, MCC merits an effort to understand it better.

2.3 Distribution of MCC in Linux

Our research question 1.1 concerned the prevalence of high-MCC functions. In a previous study of the Linux kernel we found that the distribution of MCC is very skewed, with many thousands of functions with extremely low MCC and few functions with extremely high MCC (the highest value observed was 620) [16]. In addition, we found that the distribution has a heavy tail, namely one that decays according to a power law.

It is especially interesting to observe how this distribution has changed with time. Such a study reveals two seemingly contradictory findings [16]. First, it was found that the absolute number of high-MCC functions is growing with time: in version 1.0 in 1994 there were only 15 functions with MCC of 50 or more, and in 2008 there were more than 400 such functions. At the same time it was also found that the distribution as a whole is shifting towards lower MCC values: In 1994 the median MCC was 4 and the 95th percentile was 20, but by 2008 the median was 2 and the 95th percentile was down to 13. This means that the number of low MCC functions is growing at a higher pace than the number of high-MCC functions.

In this paper we focus on the tail of the distribution, namely the functions with the highest MCC values. This is the interesting part of the distribution, because functions with such high MCC values are thought to be too complex and should not exist.

3 Analysis of High-MCC Functions in Linux

When studying the evolution of the Linux kernel, and in particular how various code metrics change with time, we found that some Linux kernel functions have MCC values in the hundreds [16]. Here we focus on high-MCC functions in version 2.6.37.5, released on 23 March 2011, as well as on the evolution of high-MCC functions across more than a thousand versions released from 1994 to 2011.

3.1 Data Collection

To calculate the MCC we use the pmccabe tool [4]. This tool also calculates the extended MCC, i.e. it also counts instances of logical operators in predicates (&& and ||). We use the extended version, in order to avoid the confounding effect of coding style (where a programmer uses either nested conditionals or a logical operator to achieve the same effect).

Our scripts parse all the implementation files of each Linux kernel, and collect various code metrics for functions with MCC above 100. However, in some cases the parsing is problematic. In particular, the Linux kernel is littered with #ifdef preprocessor directives, that allow for alternative compilations based on various configuration options [24]. As we want to analyze the full code base and not just a specific configuration, we ignore such directives and attempt to analyze all the code. As the resulting code may not be syntactically valid, the pmccabe tool may not always handle such cases correctly. Consequently a small part (around 1%) of the source code is not included in the analysis. (As a side note, the conditional compilation itself may also add to the complexity of the code, but we discuss this issue in another paper [17]).

3.2 Description of High-MCC Functions

The functions with MCC values of 100 or more in Linux kernel 2.6.37.5 have values ranging up to 587. 104 of these functions come from the drivers subdirectory, with others coming from arch (12 functions), fs (12 functions), sound (5 functions), net (3 functions), lib (1 function) and crypto (1 function). The sources of all 369 functions with MCC$\geq$100 that ever appeared in Linux are tabulated in Table 1. We manually examined a few of the top functions in the drivers subdirectory and found them dominated by switch statements of symbolic constants. These constants essentially represent *ioctl* codes for devices, different modes for emulations, and usage tables of different human interface devices.

Our research question 1.2 concerns the origin of high MCC counts. A high MCC can be the result of any type of branching statements: cases in a switch, if statements, or the loop constructs while, for, and do. But in the high-MCC functions of Linux the origin is usually multiple if statements or cases in a switch statement, as shown in Fig. 1. These can be nested in various ways. Somewhat common structures are a large switch with small trees of ifs in many of its cases, or large trees of ifs and elses. Logical operators, which can also

Directory	*Subdirectory*	*# high-MCC functions*	*Comments*
drivers	staging	65	new drivers being staged into the system
	media	35	
	video	25	
	sound	25	
	scsi	24	
	isdn	21	
	net	15	
	usb	14	
	char	14	character device drivers e.g. ttys and mice
	gpu	11	
	block	9	block device drivers like IDE disks
	others	27	
	total	**285**	
arch	m68k	6	
	sparc64	5	
	sparc	4	
	powerpc	4	
	parisc	4	
	x86	3	
	ia64	2	
	cris	1	
	mn10300	1	
	total	**30**	
sound	oss	18	cross platform Open Sound System
	pci	2	
	isa	1	
	total	**21**	
fs	xfs	4	
	ext4	2	
	ncpfs	2	
	others	9	
	total	**17**	
net	ipv6	2	
	ipv4	2	
	core	2	
	802	1	
	atm	1	
	ieee80211	1	
	inet	1	
	total	**10**	
others	–	6	

Table 1 *Classification of the 369 high-MCC functions according to the directories that contain them.*

be considered as branch points due to short-circuit evaluation, also make some contribution. Loops are quite rare.

Apart from the highest-MCC function, which is an obvious outlier, the rest of the distribution shown in Fig. 1 is seen to decline rather slowly. Indeed, in this version of Linux there were 138 functions with MCC$\geq$100, and 802 with MCC$\geq$50. Thus high-MCC functions are not uncommon (albeit they are a very small fraction of the total functions in Linux — those with MCC of 50 or more constitute just 0.3%).

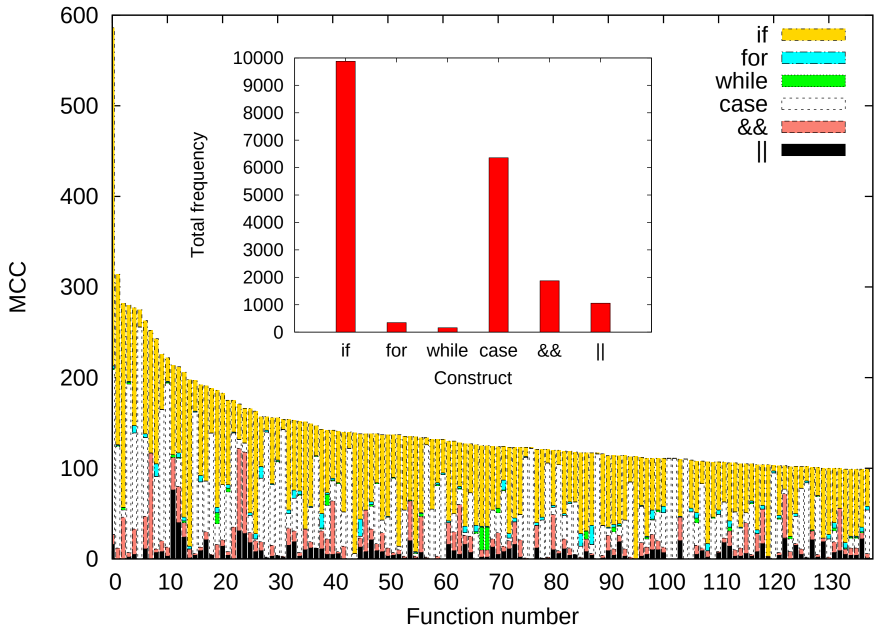


Fig. 1 *Distribution of constructs in high-MCC functions.*

3.3 Visualization of Constructs and Nesting Structure

High-MCC functions are naturally quite long, and include very many programming constructs. As a result, it is hard to grasp their structural properties. To overcome this problem and provide better insights into research question 1.2, we introduce control structure diagrams (CSD) to visualize the control structure and nesting. These are somewhat similar to the diagrams used by Adams et al. [1] to visualize patterns of using the C preprocessor.

In these diagrams (for example Figure 2) the bar across the top represents the length of the function, which starts at the left and ends at the right. Below this the nesting of different constructs is shown, with deeper nesting indicated by a lower level. Each control type is represented by a different shape and color. Each construct (except large loops) is scaled so as to span the correct range of lines in the function. This helps to easily identify the dominant control structures, which are possible candidates for refactoring.

Using the CSDs we easily observe each function's nesting structure and regularity, which may affect the perceived complexity of the code[1]. Some of the high-MCC functions are relatively flat and regular. An example is shown in Fig. 2. This function starts with many small ifs in sequence, and then has 9 large ifs with nested small ifs, two of which have large else blocks with yet another level of nested small ifs. Despite the large number of ifs this function is shallow and regular and does not appear complicated. Other functions, like that shown in Fig. 3, include deep nesting and appear to be more complicated. Regularity and its effect on perceived complexity are discussed in Section 6.

Recall that the high MCCs observed are predominantly due to if statements and cases in switch statements. This means that the flow is largely linear, with branching used to select

[1] Graphs for all functions analyzed are available at www.cs.huji.ac.il/~ahmadjbara/hiMCC.htm

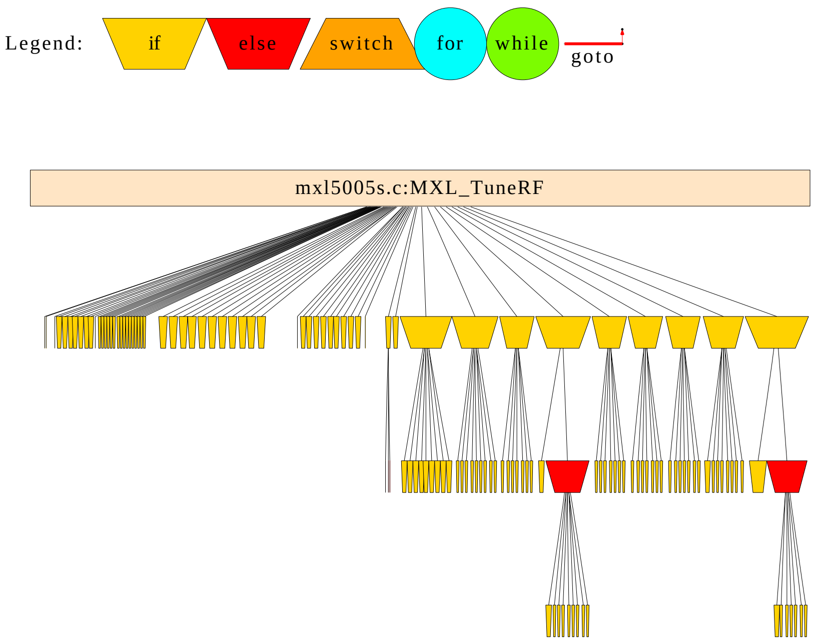


Fig. 2 *A function that is a largely flat sequence of* ifs.

the few pieces of code that should actually be executed in each invocation of the function. Only a relatively small fraction of the functions include loops, and in most cases these are small loops. Fig. 4 shows an example of a function that had relatively many loops, and even in this case they can be seen to be greatly outnumbered by ifs and cases.

While most practitioners typically limit themselves to using nested structured programming constructs, some also use goto. The goto instruction is one that breaks the function's structure and decreases code readability, in particular when backwards jumps occur between successive constructs [10]. The CSD visualizes the source and destination points of each goto and their relative locations within the code. Fig. 5 shows examples of two functions that use goto. In the first gotos are used only to break out of nested constructs in case of error, and go directly to cleanup code at the end of the function. This is usually considered acceptable. But the second uses gotos to create a very complicated flow of control, which is much more problematic.

3.4 Correlation of MCC with Other Metrics

Research question 1.3 deals with the correlation of MCC with other metrics. Indeed, one of the criticisms of MCC is that it does not provide any significant information beyond that provided by other code metrics, notably LOC (lines of code). The claim is that longer code naturally has more branch points, and thus LOC and MCC are correlated. Indeed, when comparing the MCC and LOC of all the functions in the Linux kernel, a significant correlation is observed (Fig. 6). The correlation coefficient is 0.88, and the regression line indicates that on average there are 3.8 lines of code for every branch (unit of MCC). However, there

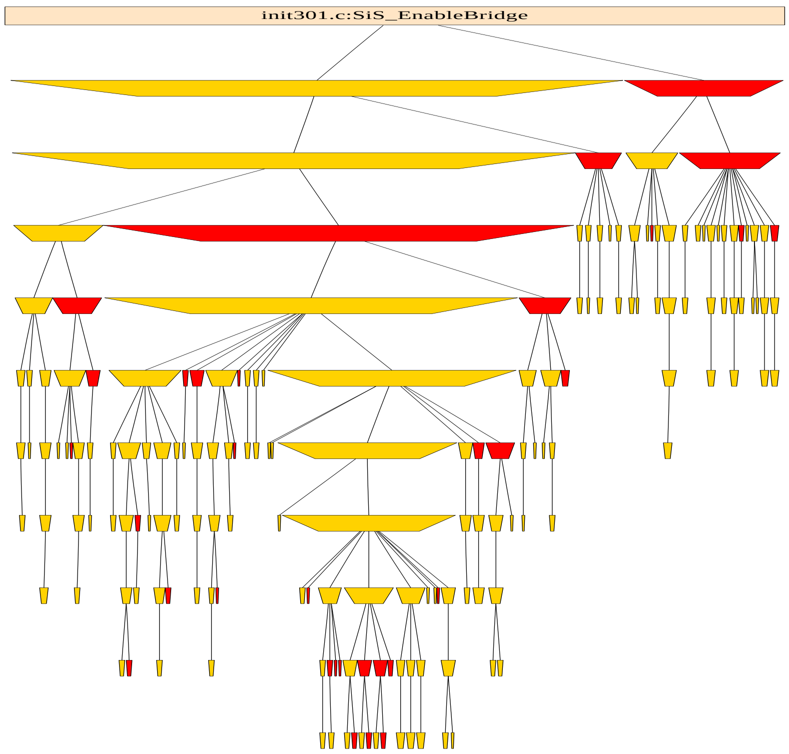


Fig. 3 *A function with irregular ifs and relatively deep nesting.*

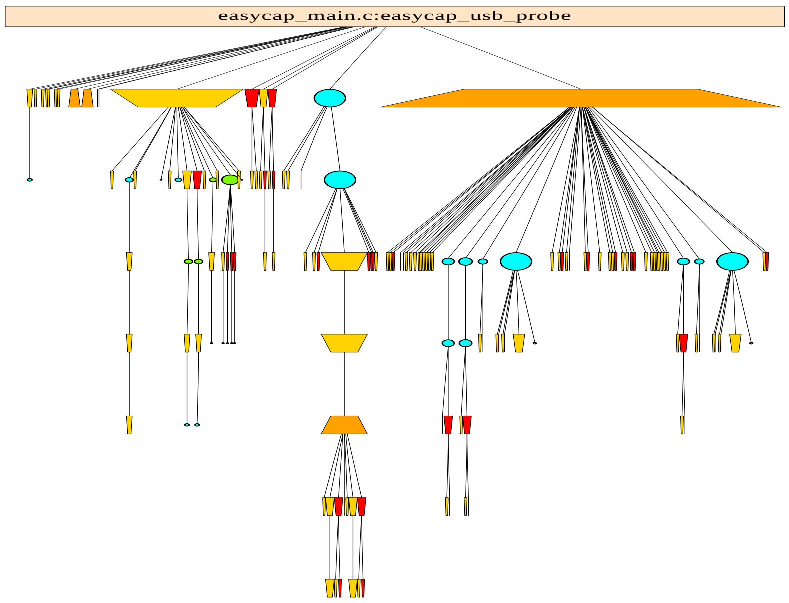


Fig. 4 *A function with relatively many loops.*

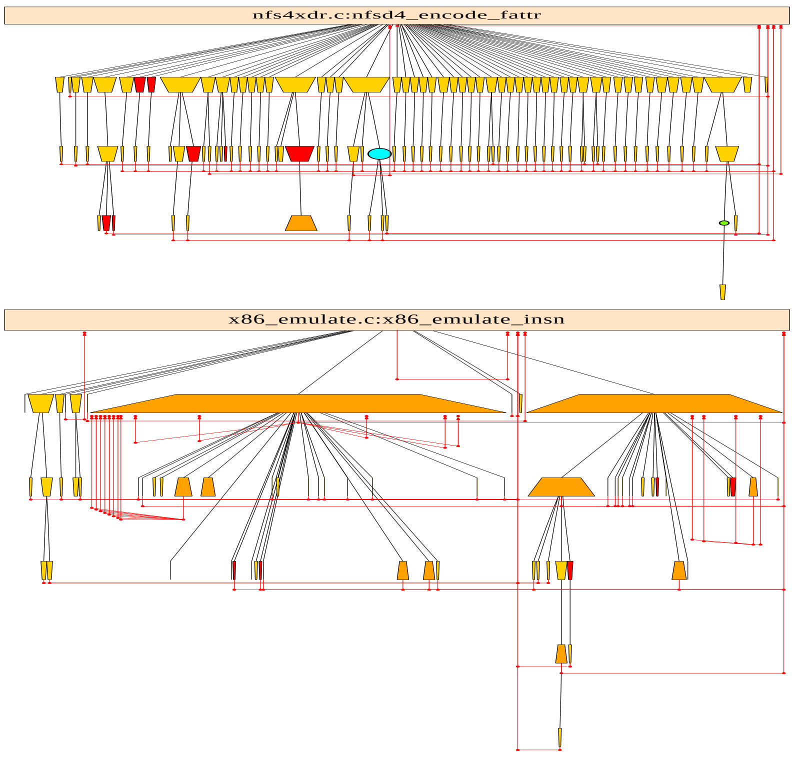


Fig. 5 *Examples of functions using goto.*

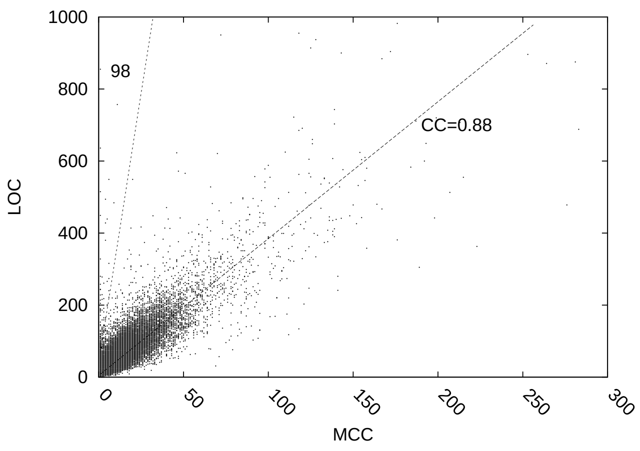
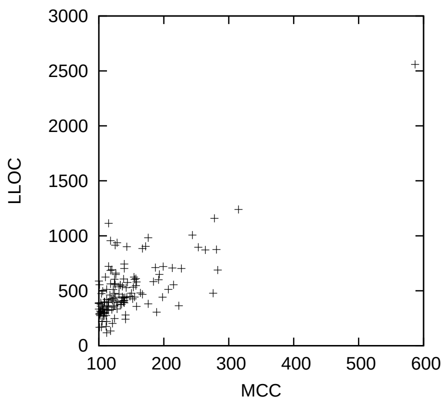


Fig. 6 *Correlation of MCC with LOC for all functions in Linux kernel 2.6.37.5.*



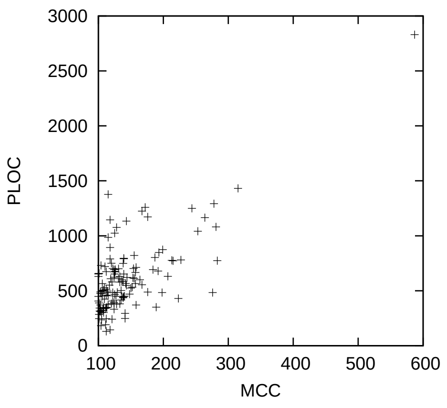


Fig. 7 *Correlation of MCC with LLOC and PLOC.*

is some variability, with a few functions where the LOC outstrips the MCC by a factor of 30 or more (to the left of the top line in the figure).

But if we focus on the high-MCC functions, the picture is somewhat different. The results are shown in Fig. 7, with a distinction between LLOC, the non-comment non-blank lines of code, and PLOC, the total number of lines. The Spearman's rank correlation coefficients are 0.586 and 0.507, respectively, indicating a moderate degree of correlation; and indeed some functions have a relatively low MCC but high LOC, or vice versa. We used Spearman's coefficient rather than Pearson's because it is more sensitive to correlations when the relationships are not linear.

Another question is whether MCC is correlated with other complexity metrics. As an example, we checked the correlation of MCC with levels of indentation and nesting, based on the premise that indentation reflects levels of nesting and higher complexity [15]. Note that this has to be done carefully so as to avoid artifacts resulting from continuation lines where indentation does not reflect the structure of the code.

The results are shown in Fig. 8. Obviously there is almost no correlation of MCC with the average level of indentation or nesting in each function (verified by calculating the correlation coefficient). This reflects our findings that high-MCC functions could be either flat `switch`s and sequences of `if`s, or else deep trees of nested `if`s, so a high MCC can come with either high or low nesting.

4 Maintenance and Evolution of High-MCC Functions

Linux is an evolving system [16]. It has shown phenomenal growth during the 17 years till the time the kernel we studied was released in 2011: version 1.0 had 122,442 lines of actual code, and version 2.6.37.5 had 9,185,179 lines, an average annual growth rate of 29%. This testifies to Lehman's law of "continuing growth" of evolving software systems [23]. Obviously, most of the functions in the current release didn't exist in the first release—

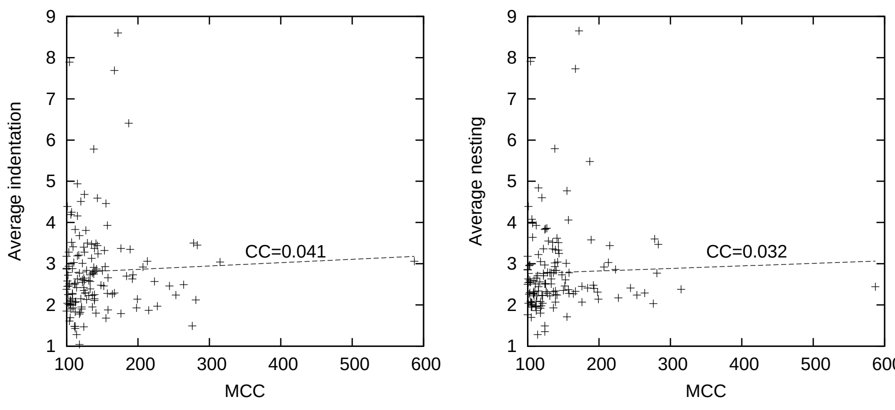


Fig. 8 *Correlation of MCC with indentation and nesting.*

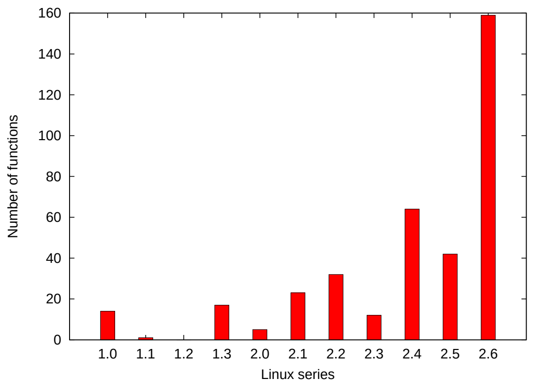


Fig. 9 The distribution of new high-MCC functions (defined as those with MCC>100) in Linux series. Note that the duration of the 2.6 series is much longer than the previous ones.

they were added at some point along the way. And there were also functions that were part of the kernel for some time and were later removed.

A function can achieve high MCC by incremental additions, or else a new function may already have a high MCC when it is added. In fact, this happened in all versions as shown in Fig. 9. (The relatively large number of new functions with MCC above 100 introduced during the 2.6 series is due to the length of this series, which was started in December 2003.) Regarding incremental growth, note that high-MCC functions are expected to be hard to maintain. It is therefore interesting to investigate their trajectory and check how often they are changed, and this was our research question 2. We did this for all Linux functions that achieved an MCC of 100 or more in any version of the kernel. There were 369 such functions.

To get an initial insight about the evolution of high-MCC functions, we calculate the coefficient of variation (CV) of the MCC of each function in different versions of Linux.

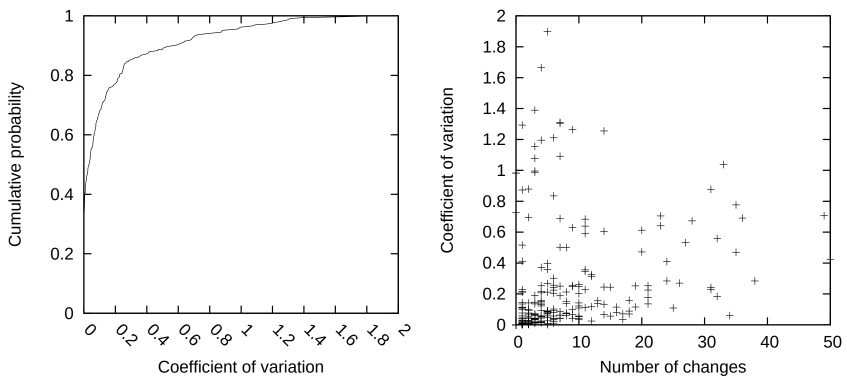


Fig. 10 Left: The distribution of the coefficient of variation of the MCC of 369 high-MCC functions. Right: Scatter plot showing relationship between number of times the MCC changed and the degree of change as measured by the coefficient of variation (if the coefficient of variation equals 0 it means the MCC of this function did not change).

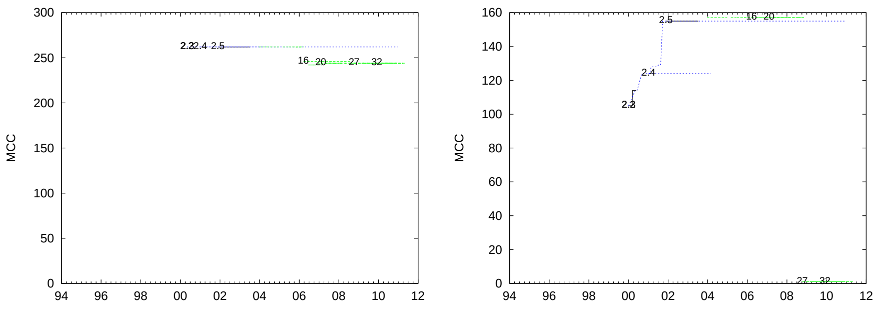


Fig. 11 *Examples of functions whose MCC changed somewhat over time: riocontrol, and ixj_ioctl*

The coefficient of variation is the standard deviation normalized by the average. Thus if a function never changes it will always have the same MCC, and the CV will be 0. If its MCC changes significantly with time, its CV can reach a value of 1 or even more. Fig. 10 shows the distribution of the calculated CVs. About 33% of the functions exhibit absolutely no change in the MCC across different versions of the kernel. Note that this does not necessarily mean that the functions were not modified at all, as we are only using data about the MCC. However it does indicate that in all likelihood the control structure did not change. Another large group of functions exhibit small to medium changes in MCC over time. Examples are shown in Fig. 11[2]. Finally, some functions exhibited significant changes in their MCC. Examples are shown in Fig. 12.

[2] In this and subsequent figures, we distinguish between development versions of Linux (1.1, 1.3, 2.1, 2.3, and 2.5), production versions (1.0, 1.2, 2.0, 2.2, and 2.4, shown as dashed lines), and the 2.6 series, which combined both types. These are identified only by their minor (third) number. The X axis is calendar years starting with the release of Linux in 1994.

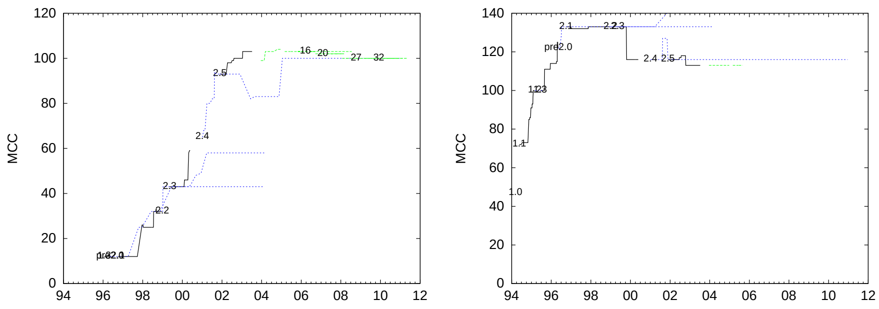


Fig. 12 *Examples of functions that exhibit significant changes over time:* vortex_probe1*, and* st_int_ioctl

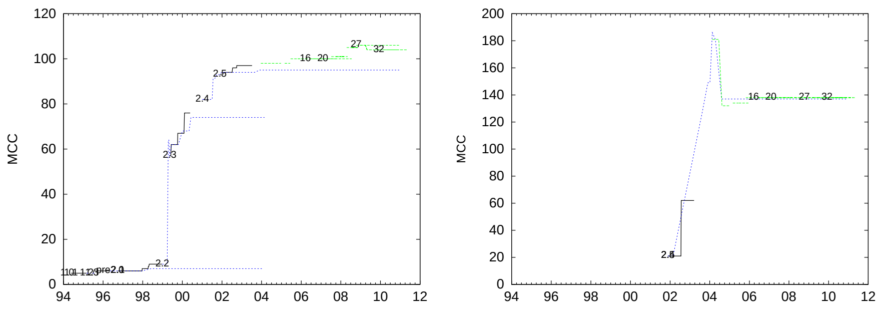


Fig. 13 *Examples of functions that exhibit large changes in production versions:* sg_ioctl *and* SiS_EnableBridge*.*

The degree to which the MCC changes is only one side of the story. In principle a very large change may occur all at once, or as a sequence of smaller changes. Therefore it is also interesting to check the number of times that the MCC was changed relative to the previous version. This has to be done carefully, because the Linux release scheme of using production and development versions (described below) implies that several versions may be current at the same time. Thus when a new branch is started, its previous version is typically near the start of the previous branch, not at its end.

Fig. 10 shows a scatter plot that compares the degree of change with the number of changes. The correlation between these two metrics turns out to be relatively strong, with a Spearman's rank correlation coefficient of 0.83. This shows that additional changes tend to accumulate. However, despite the rapid rate in which new releases of the Linux kernel are made, the high-MCC functions do not change often. The highest number we saw was a function whose MCC changed 50 times.

An especially interesting phenomenon is that sometimes very large changes occur in production versions. The Linux kernel, up to the 2.6 series, employed a release scheme that differentiated between development and production. Development versions had an odd major number and their minor releases were made in rapid succession. Production versions, with even major numbers, were released at a much slower rate, and these releases were only supposed to contain bug fixed and security patches. However, our data shows several instances of large changes in the MCC of a function that occur in the middle of a production version (Fig. 13 and vortex_probe1 from Fig. 12). Such behavior contradicts the "official" semantics of development vs. production versions. But at least in some of these cases the

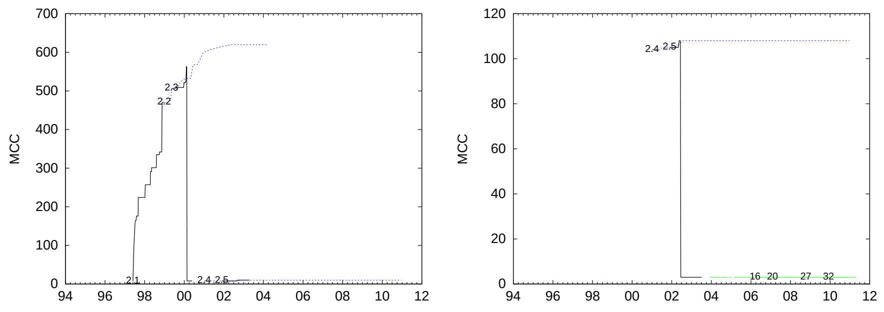


Fig. 14 *Examples of functions that exhibit a sharp drop in MCC resulting from a design change:* sys32_ioctl *and* usb_stor_show_sense*.*

change was done in a production version during the interval between two successive development versions.

In most functions that saw a significant change in MCC the MCC grew. But there were also cases where the MCC dropped as shown in Fig. 14. The largest drop is in function sys32_ioctl. This is the function with the highest MCC ever, peaking at 620 in the later parts of kernel version 2.2. At an earlier time, in version 2.3.46, it had reached an MCC value of 563, but then in version 2.3.47 this dropped to 8. The reason was a design change, where a large switch was replaced by a table lookup [16]. A similar change occurred in function usb_stor_show_sense, where a large switch statement was replaced by a call to a new function implementing a lookup table.

However, a sharp drop in MCC value does not necessarily mean a design change which yields reduced complexity. For example, the function isdn_tty_cmd_PLUSF_FAX had MCC 154 in version 2.2.14. In version 2.2.15 it dropped to 3 and the original code was replaced by conditional calls to two other new functions. One of these functions has MCC 154 exactly as the original function, and the other has MCC 15. Thus the high-MCC code just moved elsewhere. Likewise, in version 2.3.9 the function I2o_proc_read_lan_media_operation had MCC 102, which dropped to 12 in version 2.3.10. The original function had two large switchs which were cloned later in the same function. In version 2.3.10 the two switchs were replaced by two new functions. Each of the new functions contained one of the original switch statements and a new lookup table. The odd thing was that the lookup tables did not replace the switch statements and were not exploited to reduce complexity. Another example of an artificial reduction in MCC is function fd_ioctl_trans. The original function had many long compound if statements with heavy use of the or operator. In its reduced MCC version the logical or operator was replaced by the bitwise or which is not counted by the MCC metric.

The above examples may leave the impression that design changes to reduce MCC are purely technical. However, we also observed cases where the reduction resulted from a design change requiring a good understanding of the logic of the function, as the changes are small and deeply interwoven within the code. An example of such a function is main in versions 2.4.25 and 2.4.26. The chief change in MCC resulted from defining 13 new secondary functions ranging from 1 to 50 lines of code. While in the old version negative numbers were used to indicate an error code when returning from a secondary function, in the new version these numbers were replaced by positive ones. In addition, in the old version all exceptional cases were handled locally, whereas in the new version the goto mechanism was used; upon

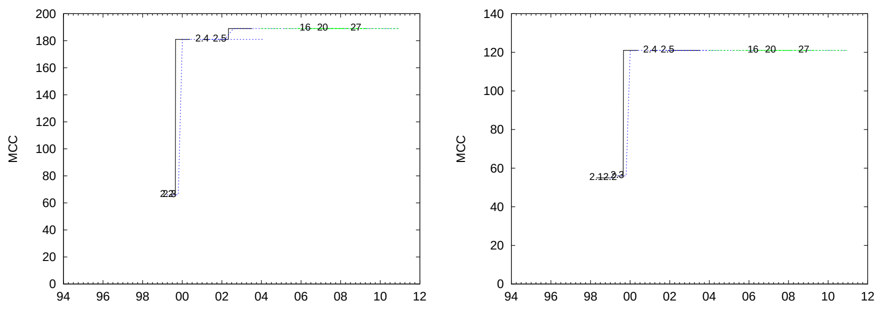


Fig. 15 *Co-evolution of two related functions.*

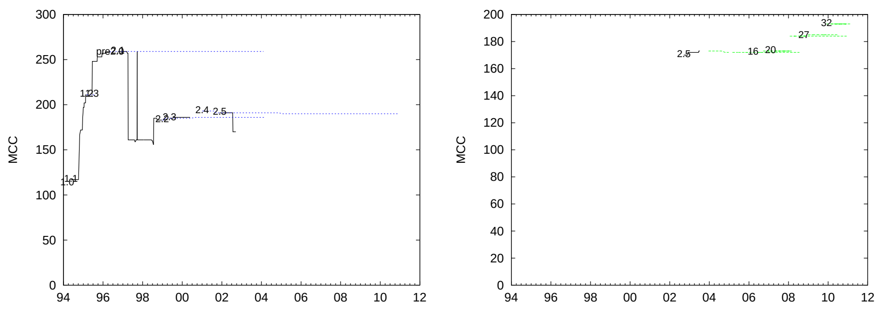


Fig. 16 *Evolution of the vt_ioctl function, which migrated from one file to another.*

exception execution jumps to a label which is located at the end of the function. All these changes require intimate understanding of the function.

Other interesting phenomena that occurred during maintenance were co-evolution and migration. Fig. 15 shows the co-evolution of two related functions. These functions are do_mathemu in /arch/sparc64/math-emu/math.c, and do_one_mathemu in /arch/sparc/math-emu/math.c. This occurs when two related functions evolve according to a similar pattern. In many cases this happens because one of the functions was originally cloned from the other. In the above example, these are analogous functions in 32-bit and 64-bit architectures; when a large change was implemented, it was done in both in parallel. Also, in both cases the change that was initially done in a development version was soonafter propagated to the contemporaneous production version.

An example of migration is shown in Fig. 16: the vt_ioctl function, which moved from /drivers/char/vt.c (MCC of 159 in kernel 2.5.35) to /drivers/char/vt_ioctl.c (same MCC of 159 in kernel 2.5.36). In fact, these two functions are indeed identical. As another example, cpia_write_proc from /drivers/char/cpia.c, with an MCC of 226 in kernel 2.2.26, morphed into cpia_write_proc in /drivers/media/video/cpia.c, with an MCC of 211 in kernel 2.4.0 (via the 2.3.99-pre*x* series, where it already was 211). The change in MCC reflects some changes in the structure of the function. Much larger changes occurred when x86_emulate_memop from /drivers/kvm/x86_emulate.c, with an MCC of 285 in 2.6.24.7, morphed into x86_emulate_insn in /arch/x86/kvm/x86_emulate.c with an MCC of 174 in 2.6.25. While the second function is partly based on the first, significant changes were made, and the MCC changed considerably as well.

To summarize, high-MCC functions in the Linux kernel evolve in a variety of ways. This includes cases where a function changes significantly over time in a series of individual changes, and cases where functions are split or completely restructured. Taken together, these observations provide evidence for the capability of developers to handle these seemingly complex functions. In the next two sections we investigate whether they are indeed so complex.

5 Survey of Perceived Complexity

The raison d'être of the definition of MCC is the desire to be able to identify complex code, with the further goal of avoiding or restructuring it. This is also the reason for specifying threshold values, and requiring functions that surpass these thresholds to have proper justification. But the question remains whether MCC indeed captures complexity as perceived by human programmers.

5.1 Correlation of MCC and Perceived Complexity

To gain some insight into this question, which is our research question 3.1, we conducted a survey of the perceived complexity of high-MCC functions. The survey included 92 high-MCC functions that had been identified at the time. It was based on 14 participants, 8 from a summer Linux kernel workshop (advanced undergraduates, some with industrial experience, but with no prior kernel experience), and 6 that were recruited later (all were good students after an advanced course in C programming). The goal was to identify notions of perceived complexity, not to quantify the effect of complexity on developer performance. Thus the survey was conducted in two hour-long sessions, in which participants were required to page through each function for one minute and then give it a grade based on how complex (hard to understand) it looked to them. Grades where given on a personal relative scale[3]. These individual scales where then linearly normalized to the range 0 to 10, and the average and standard deviation of the grades for each function were computed. The order in which the functions were presented was not related to MCC or any other attribute, but all participants received the list in the same order. At the end of the survey, participants were given an opportunity to comment in writing and some indeed provided notes with their insights.

The results, shown in Fig. 17, indicate little correlation between MCC and perceived complexity for high-MCC functions. In particular, some functions with relatively low MCC (within this select set of high-MCC functions) were graded as having either very high or very low perceived complexity. In the following we focus on these functions that were perceived as very different but this was not reflected by their MCC.

The functions that had very low average scores (and to a lesser degree also those with very high scores) also had relatively low standard deviations, as indicated by the short error bars. This is partly a result of scores having a limited range of 0–10; an average of say 1 then implies that it is highly improbable to have any high scores. But some of the functions with a moderate average complexity actually had both low and high scores of perceived complexity, leading to a high standard deviation. This is partly due to the fact that complexity is not well defined and the grading was subjective. For example, one very long function with high MCC

[3] This was chosen to enable them to respond to surprises. Thus if they see a function they think is "very complex" and give it a high mark, and later another that is even much more complex, they can still express this using a value beyond their previously used range.

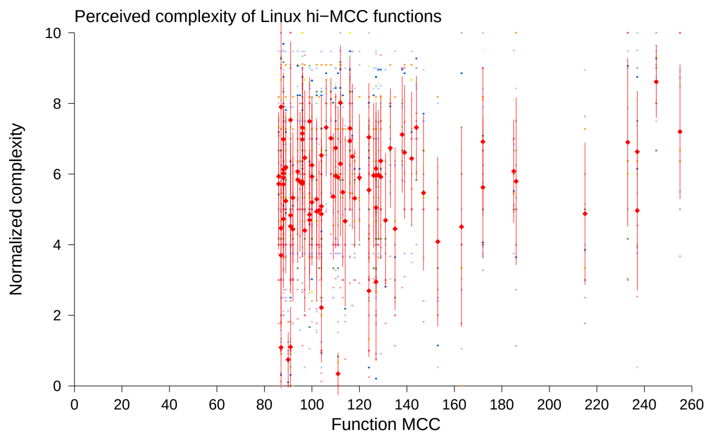


Fig. 17 *Scatter plot showing relationship between measured MCC and perceived complexity. The small markings are individual grades. The average grade for each function is marked by a larger diamond, and the error bars denote standard deviations.*

```
switch (mod_det_stat0) {
case 0x00: p = "mono"; break;
case 0x01: p = "stereo"; break;
case 0x02: p = "dual"; break;
case 0x04: p = "tri"; break;
case 0x10: p = "mono with SAP"; break;
case 0x11: p = "stereo with SAP"; break;
case 0x12: p = "dual with SAP"; break;
case 0x14: p = "tri with SAP"; break;
case 0xfe: p = "forced mode"; break;
default: p = "not defined";
}
```

Fig. 18 *Example of simple switch structure from log_audio_status.*

value suffered a strong disagreement among survey participants. This could be because this function is composed of a mix of simple as well as messy segments.

5.2 Aspects of Complexity Missed by MCC

The functions that were ranked as low complexity are relatively easy to characterize. These are generally functions dominated by a very regular switch construct, where the cases are very small and straightforward. For example, the switch may be used to assign error or status message strings to numerical codes, leading to a single instruction in each case as illustrated in Fig. 18.

In addition to these single-instruction cases, survey participants noted that long sequences of empty cases should not be counted as adding complexity; indeed, these are equivalent to predicates in which many options are connected by logical or (and of the tools we surveyed, VerifySoft indeed does not count empty cases). Furthermore, repeated use of the same code template (easily identified using a CSD), e.g. in a long sequence of small ifs that all have exactly the same structure, also reduces the perceived complexity considerably. An example is shown in Fig. 19.

```
bytes.high = 0x14;
bytes.low = j->m_DAAShadowRegs.SOP_REGS.SOP.cr4.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.SOP_REGS.SOP.cr3.reg;
bytes.low = j->m_DAAShadowRegs.SOP_REGS.SOP.cr2.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.SOP_REGS.SOP.cr1.reg;
bytes.low = j->m_DAAShadowRegs.SOP_REGS.SOP.cr0.reg;
if (!daa_load(&bytes, j))
return 0;

if (!SCI_Prepare(j))
return 0;

bytes.high = 0x1F;
bytes.low = j->m_DAAShadowRegs.XOP_REGS.XOP.xr7.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.XOP_xr6_W.reg;
bytes.low = j->m_DAAShadowRegs.XOP_REGS.XOP.xr5.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.XOP_REGS.XOP.xr4.reg;
bytes.low = j->m_DAAShadowRegs.XOP_REGS.XOP.xr3.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.XOP_REGS.XOP.xr2.reg;
bytes.low = j->m_DAAShadowRegs.XOP_REGS.XOP.xr1.reg;
if (!daa_load(&bytes, j))
return 0;

bytes.high = j->m_DAAShadowRegs.XOP_xr0_W.reg;
bytes.low = 0x00;
if (!daa_load(&bytes, j))
return 0;

if (!SCI_Prepare(j))
return 0;
```

Fig. 19 *Example of a sequence of independent ifs with the same structure, from ixj_daa_write. The full function includes 113 such ifs.*

At the other end of the spectrum, functions that received very high grades for perceived complexity tended to exhibit either of two features. One was the use of gotos to create spaghetti-style code, in which target labels are interspersed within the function's code in different locations. An example was shown in Fig. 5. Note that such a goto is deterministic, and therefore not counted by the MCC metric as a branch point. This should be contrasted with forward gotos that are used to break out of a complex control structure in case of an error condition. Such gotos were tolerated by survey participants and even considered as improving structure.

```
if (ret_val
    && !item_pos) {
        pasted =
            B_N_PITEM_HEAD
            (tb->L[0],
             B_NR_ITEMS
             (tb->
              L[0]) -
             1);
        l_pos_in_item +=
            I_ENTRY_COUNT
            (pasted) -
            (tb->
             lbytes -
             1);
}
```

Fig. 20 *Example of excessive line breaks that seem to make the code harder rather than easier to understand, from balance_leaf.*

The second feature that added to perceived complexity was unusual formatting. One manifestation of such formatting was using only 2 characters as the basic unit of indentation (instead of the common 8-character wide tab). This led to the code looking more dense and made it harder to decipher the control structure. Another manifestation was the use of excessive line breaks, even within expressions, as illustrated in Fig. 20. These observations hark back to the work of Soloway and Ehrlich [39], who show that even expert programmers have difficulty comprehending code that does not conform to structural conventions. Obviously the problem could be avoided by using a pretty-printing routine to reformat the code, but evidently this was not done.

5.3 Comparing Functions to Identify Elements of Complexity

The functions that were found to have the lowest perceived complexity provide an especially interesting case study. These functions are generally based on large switch statements, and most if not all of their MCC score is derived from cases in these switchs. We start by ranking these functions according to their perceived complexity. By comparing neighboring functions in this ranking we can then identify code characteristics that led to discrete increases in perceived complexity (thus answering research question 3.2). This could be done in the first 7 functions; beyond that, it was not possible to identify individual discrete changes any more.

The function with the lowest perceived complexity score is indeed very simple. This function has one parameter, and its body comprises a single switch statement with a long sequence of cases that are compared against the function's parameter. The values of the cases are numeric constants and their bodies are single-line blocks that return a string value. Moreover, the cases are grouped into sets of logically related cases. These sets are paragraphed (separated by blank lines) and headed by a single-line comment.

The next function, which was graded as twice more complex than the first one, accepts one non-scalar parameter, and again contains one switch statement with a long sequence of cases. The values of the cases are symbolic constants (except a few cases of numeric constants) and their bodies assign string values to a shared variable and then break. There is no paragraphing nor comments. After the switch statement there are a very simple loop and

```
char *capi_info2str(u16 reason)
{
    switch (reason) {

/*-- informative values (corresponding message was processed) -----*/
 case 0x0001:
    return "NCPI␣not␣supported␣by␣current␣protocol,␣NCPI␣ignored";
 case 0x0002:
    return "Flags␣not␣supported␣by␣current␣protocol,␣flags␣ignored";
 case 0x0003:
    return "Alert␣already␣sent␣by␣another␣application";

/*-- error information concerning CAPI_REGISTER -----*/
 case 0x1001:
    return "Too␣many␣applications";
 case 0x1002:
    return "Logical␣block␣size␣too␣small,␣must␣be␣at␣least␣128␣Bytes";
 case 0x1003:
    return "Buffer␣exceeds␣64␣kByte";
 case 0x1004:
    return "Message␣buffer␣size␣too␣small,␣must␣be␣at␣least␣1024␣Bytes";
 case 0x1005:
    return "Max.␣number␣of␣logical␣connections␣not␣supported";
 case 0x1006:
    return "Reserved";
 case 0x1007:
    return "The␣message␣could␣not␣be␣accepted␣because␣of␣an␣internal␣
         busy␣condition";
 case 0x1008:
    return "OS␣resource␣error␣(no␣memory␣?)";
 case 0x1009:
    return "CAPI␣not␣installed";
 case 0x100A:
    return "Controller␣does␣not␣support␣external␣equipment";
 case 0x100B:
    return "Controller␣does␣only␣support␣external␣equipment";

/*-- error information concerning message exchange functions -----*/
 case 0x1101:
    return "Illegal␣application␣number";
 case 0x1102:
    return "Illegal␣command␣or␣subcommand␣or␣message␣length␣less␣than␣12
         ␣bytes";
 case 0x1103:
    return "The␣message␣could␣not␣be␣accepted␣because␣of␣a␣queue␣full␣
         condition␣!!␣The␣error␣code␣does␣not␣imply␣that␣CAPI␣cannot␣
         receive␣messages␣directed␣to␣another␣controller,␣PLCI␣or␣NCCI";
 case 0x1104:
    return "Queue␣is␣empty";
 case 0x1105:
    return "Queue␣overflow,␣a␣message␣was␣lost␣!!␣This␣indicates␣a␣
         configuration␣error.␣The␣only␣recovery␣from␣this␣error␣is␣to␣
         perform␣a␣CAPI_RELEASE";
 case 0x1106:
    return "Unknown␣notification␣parameter";
 case 0x1107:
    return "The␣Message␣could␣not␣be␣accepted␣because␣of␣an␣internal␣
         busy␣condition";
 case 0x1108:
    return "OS␣Resource␣error␣(no␣memory␣?)";
 case 0x1109:
    return "CAPI␣not␣installed";
 case 0x110A:
    return "Controller␣does␣not␣support␣external␣equipment";
 case 0x110B:
    return "Controller␣does␣only␣support␣external␣equipment";

         ... // 4 paragraphs of cases removed to save space

 default: return "No␣additional␣information";
    }
}
```

Listing 1 Listing of the function with the lowest perceived complexity.

```
void usb_stor_show_command(struct scsi_cmnd *srb)
{
 char *what = NULL;
 int i;

 switch (srb->cmnd[0]) {
 case TEST_UNIT_READY: what = "TEST_UNIT_READY"; break;
 case REZERO_UNIT: what = "REZERO_UNIT"; break;
 case REQUEST_SENSE: what = "REQUEST_SENSE"; break;
 case FORMAT_UNIT: what = "FORMAT_UNIT"; break;
 case READ_BLOCK_LIMITS: what = "READ_BLOCK_LIMITS"; break;
 case REASSIGN_BLOCKS: what = "REASSIGN_BLOCKS"; break;
 case READ_6: what = "READ_6"; break;
 case WRITE_6: what = "WRITE_6"; break;
 case SEEK_6: what = "SEEK_6"; break;
 case READ_REVERSE: what = "READ_REVERSE"; break;
 case WRITE_FILEMARKS: what = "WRITE_FILEMARKS"; break;
 case SPACE: what = "SPACE"; break;
 case INQUIRY: what = "INQUIRY"; break;
 case RECOVER_BUFFERED_DATA: what = "RECOVER_BUFFERED_DATA"; break;
 case MODE_SELECT: what = "MODE_SELECT"; break;
 case RESERVE: what = "RESERVE"; break;
 case RELEASE: what = "RELEASE"; break;
 case COPY: what = "COPY"; break;
 case ERASE: what = "ERASE"; break;
 case MODE_SENSE: what = "MODE_SENSE"; break;
 case START_STOP: what = "START_STOP"; break;
 case RECEIVE_DIAGNOSTIC: what = "RECEIVE_DIAGNOSTIC"; break;
 case SEND_DIAGNOSTIC: what = "SEND_DIAGNOSTIC"; break;
 case ALLOW_MEDIUM_REMOVAL: what = "ALLOW_MEDIUM_REMOVAL"; break;
 case SET_WINDOW: what = "SET_WINDOW"; break;
 case READ_CAPACITY: what = "READ_CAPACITY"; break;
 case READ_10: what = "READ_10"; break;
 case WRITE_10: what = "WRITE_10"; break;
 case SEEK_10: what = "SEEK_10"; break;
 case WRITE_VERIFY: what = "WRITE_VERIFY"; break;
 case VERIFY: what = "VERIFY"; break;
 case SEARCH_HIGH: what = "SEARCH_HIGH"; break;
 case SEARCH_EQUAL: what = "SEARCH_EQUAL"; break;
 case SEARCH_LOW: what = "SEARCH_LOW"; break;
 case SET_LIMITS: what = "SET_LIMITS"; break;
 case READ_POSITION: what = "READ_POSITION"; break;
 case SYNCHRONIZE_CACHE: what = "SYNCHRONIZE_CACHE"; break;
 case LOCK_UNLOCK_CACHE: what = "LOCK_UNLOCK_CACHE"; break;
 case READ_DEFECT_DATA: what = "READ_DEFECT_DATA"; break;
 case MEDIUM_SCAN: what = "MEDIUM_SCAN"; break;
 case COMPARE: what = "COMPARE"; break;
 case COPY_VERIFY: what = "COPY_VERIFY"; break;
 case WRITE_BUFFER: what = "WRITE_BUFFER"; break;
 case READ_BUFFER: what = "READ_BUFFER"; break;
 case UPDATE_BLOCK: what = "UPDATE_BLOCK"; break;
 case READ_LONG: what = "READ_LONG"; break;
 case WRITE_LONG: what = "WRITE_LONG"; break;
 case CHANGE_DEFINITION: what = "CHANGE_DEFINITION"; break;

        ... // cases removed to save space

 default: what = "(unknown command)"; break;
 }
 US_DEBUGP("Command %s (%d bytes)\n", what, srb->cmd_len);
 US_DEBUGP("");
 for (i = 0; i < srb->cmd_len && i < 16; i++)
  US_DEBUGPX(" %02x", srb->cmnd[i]);
 US_DEBUGPX("\n");
}
```

Listing 2 Listing of the second function with low perceived complexity.

a call to a macro. The loop references (for the first time) a variable which was defined before the switch statement, and the macro uses the variable which was previously assigned within the switch statement. Listings 1 and 2 represent the first and second functions.

The third and forth functions were very similar to each other and received very close perceived complexity grades. They accept one non-scalar parameter and are composed of many separate switch statements with paragraphing but no comments. The values of the cases are numeric constants and the bodies are assignments to a shared variables, followed by break. A few of these switch statements are governed by a very simple if and else, so we see some nesting. Nevertheless, the structure of these functions is still quite flat and regular.

The fifth function introduces several new elements for the first time in this series, and its average grade is again double that of the previous one. Its header is much more complex than previous functions, and contains an additional modifier besides the traditional structure. Moreover, it contains more parameters than before where some are simple and scalar and others are aggregate. These parameters are listed over multiple lines, and in one case the type of a parameter was defined in one line and its name in the next line. This function is still dominated by a large switch statement with mostly (80%) consecutive empty cases. The rest of the cases contain one if statement or a for loop with a nested if statement. In both cases the blocks of statements are very simple, but the conditions in the ifs span multiple lines.

The sixth function is composed of one large switch statement where each of its cases is composed of another large switch statement with one simple line for each of its cases. Moreover, the first case of the outer switch actually contains an if/else construct with two switch statements in them.

The last function, which was graded as a bit more complex than the previous one, is composed of two large switch statements that are controlled by if and else. The cases of these switch statements are composed of nested ifs and elses with simple conditions. Roughly, the blocks within the different cases create five categories of regular blocks. Despite the deep nesting in the different cases, the impression is that this nesting is used to break up ifs with very complicated conditions. This is obvious because each of these blocks performs a single statement in its innermost level.

The above allows us to identify the following elements of complexity, which are generally not acknowledged by metrics like MCC:

– Ending a case with a break, followed by some additional processing after the switch, is more complex than having a return directly in the case.
– Several small switchs (probably switching on different variables) are more complex than one large switch.
– Using constructs of different types, e.g. ifs in addition to a switch, increases complexity.
– Adding parameters to a function increases complexity.
– Increasing the nesting of constructs in each other increases complexity.
– Embedding switch statements within ifs and elses is more complex than having the switch at the top level.

In some of these cases the more complex version cannot be avoided due to the logic of the program. But still we can suggest the following *Do*s and *Don't*s lessons:

– Don't separate processing, localize whenever it is possible.
– If possible, prefer one large switch rather than splitting across many smaller ones.
– Try to avoid mixing constructs of different types.
– Use paragraphing (empty lines separating blocks of code) and comments.

```
{for{switch{casecasecasecasecasecasecasecasecasecasecase}if}{if{}}if{}else{}
if{}else{}if{}else{}for{switch{caseifcaseifcaseifcaseifcasecaseifcasecaseifcase
caseifcaseifcaseif}ifif}for{switch{caseifcaseifcaseifcaseifcasecaseifcasecaseif
casecaseifcaseifcaseif}ifif}for{if}ifelseif{ifelseifelseifif}else{}if{}else{if{}
else{}}if{}else{switch{casecasecasecasecasecasecasecasecasecasecaseif{}else{}}
ifelse}}
```

Fig. 21 *An example of the control structure of a function, used as the input for the compression algorithm.*

6 Regularity and Perceived Complexity

As we have already stated, High-MCC functions are quite long. Therefore, a visual representation such as that provided by CSDs may ease capturing their code as a whole, and may help in grasping structural properties and regularities. This raises the empirical question of whether a visual view of high-MCC functions has an advantage over a simple listing of the code, from a human point of view. This is research question 3.3.

As noted above, using CSDs exposed some functions as being very regular while others appear to have irregular code structure. This reflects a combination of the sequence of constructs used, their nesting pattern, and formatting aspects such as indentation and paragraphing. It seems likely that these factors contribute to the perception of complexity, even though they are not taken into account by the MCC metric. A second question is therefore whether regularity correlates with perceived complexity. If it does, this would answer our research question 4 in the affirmative.

To answer these questions we conducted an experiment with 15 experienced programmers. We required that the subjects must have experience in the C language. All subjects were males except two, with an average age of 31, and an average of 4.8 years experience with C.

The experiment consisted of 30 high-MCC functions, presented in two different formats. In one phase the code listing of the functions was presented, and in the other phase the CSD diagrams of the functions were presented. The two phases were performed separately with a break of at least one day between them. Which phase (code or CSD) was done first was randomized across subjects. The task was to assign each function with a perceived complexity score, as in the previous experiment. Before starting, participants were presented with a short description of CSDs and an example showing the code and CSD of the same function side by side.

Somewhat surprisingly, the results show that the CSD view had no advantage over the code view. In fact, in two-thirds of the cases there was no significant difference in the average scores of the two types of view. Thus it seems that experienced programmers can achieve a good impression of code by paging through it, and seeing a graphical representation of the code structure did not provide much additional information.

To answer the question of whether perceived complexity correlates with regularity, we used the Lempel-Ziv compression algorithm [46] to compress each of the 30 functions and computed the percentage of reduction in size for each one. As this algorithm is based on identifying recurring substrings in the input, we expect that functions with high regularity will be compressed better than irregular functions. But it is still unclear what parts of the code should be compressed to better reflect regularity. To check this we compressed each function twice: once using its full code as is, and again using a reduced representation of its control structure. In the control structure compression we remove the function's content and retain only its keywords and braces, as shown in Fig. 21. In each case we compare the results

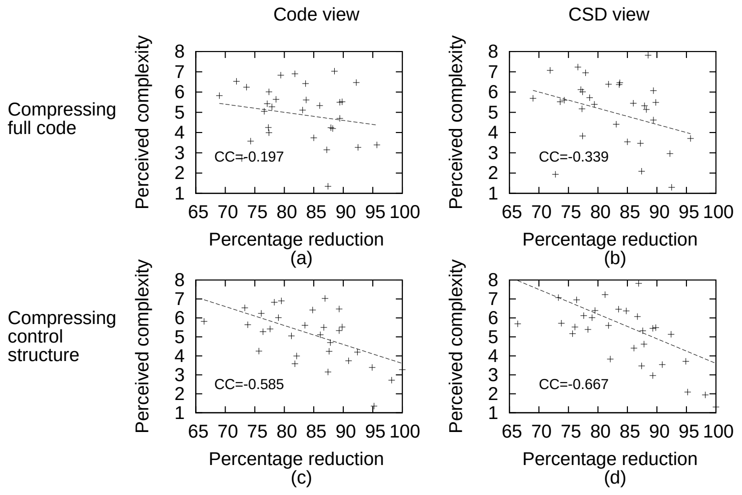


Fig. 22 *Correlation of regularity with perceived complexity. Top row (a and b) assess regularity using percent reduction of size when compressing the original code, while the bottom row (c and d) use compression of the control structure. In the left column (a and c) perceived complexity is based on the code view, and in the right column (b and d) on the CSD view.*

against the perceived complexities (code view and CSD view) as reflected in the experiment. Thus we got four combinations that are shown in Fig. 22.

The results indicate that there is a weak negative correlation between percentage reduction for raw code (regularity) and perceived complexity in code view, with a correlation coefficient of -0.197. Better correlation was achieved when contrasting the raw reduction against perceived complexity using the CSD view. Here the correlation coefficient was -0.339.

Fig. 22(c) and Fig. 22(d) show the results of comparing the percentage reduction of the control structure and the perceived complexity in both views (code and CSD). In this case the correlation coefficient was much stronger than in the raw reduction case. For Fig. 22(c) plot the correlation coefficient was -0.585 and for Fig. 22(d) it was -0.667.

These results suggest that low compressibility correlates with high perceived complexity, and by implication, that irregularity correlates with high perceived complexity.

7 Possibility of Replacing or Refactoring high-MCC Functions

To answer research question 5 (is all the high-MCC code really necessary) we surveyed all 369 such functions that were collected from more than a thousand versions of the Linux kernel, using the version with the highest MCC value for each one. This complements the quantitative metrics discussed above with a qualitative discussion aimed to gain some insights about their nature. We checked cloning, replacement of code by a lookup table, and the option of factoring out some functionality to a subroutine.

To find possible instances of cloning we compared the source code of each pair of high-MCC functions. For doing this we used the `diff` Linux command, with parameters to disregard differences in spaces and blank lines. We also allowed up to 10% of the total lines of the compared functions in each pair to be different. We repeated this process twice: Once for the full source code, and again based on the skeleton of the functions (only the keywords and braces, as explained in Section 6) while preserving formatting and nesting. Using the code structure comparison, we found 56 sets of clones, where 34 are pairs of functions, 13 involve 3 functions, 5 have 4 functions, and 4 include no less than 5 clones. For the full code, we found 51 sets of clones. These results indicate that nearly a quarter of the high-MCC functions are clones of other high-MCC functions. The existence of so many clones indicates that developers found it better to create clones with small changes rather than to abstract away the common functionality and adjust it for different uses by parameterization.

As we stated earlier, some of the high-MCC functions are written in a way that enables replacing them by a lookup table. We manually examined the 369 functions and counted those that are likely candidates for replacement by a lookup table. We found 19 such functions. In addition, we observed some functions that can be partially replaced (meaning that they contain a few code segments that can be replaced with a lookup table). There were 23 such functions. These include two sets of size 2 and 3 which also appeared in the clone list. As demonstrated in Section 4, replacement of a high-MCC function by a simpler function based on table lookup is a transformation that indeed occurs in practice.

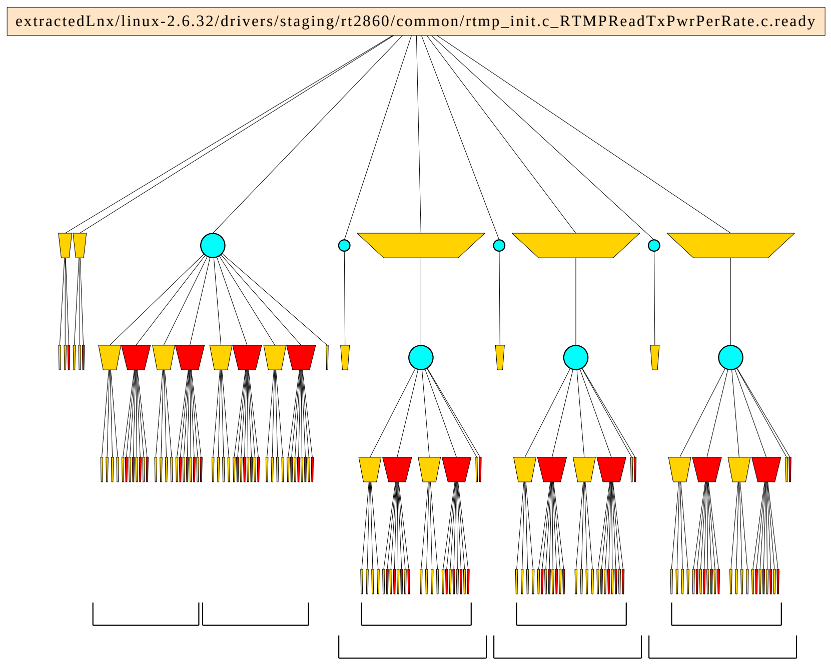
An especially interesting question is whether well-known refactoring techniques may be applied to high-MCC functions. As high-MCC functions are long, there is a good chance for applying refactoring techniques such as *function extraction*. As an initial check, we tried to identify clone code segments within a given function. We were assisted by the CSD diagram of each function to get initial impression about cloned segments. We reviewed all 369 CSDs manually in a single-evaluator style, and subjectively extracted 61 functions that have what appear to be large cloned segments. Two examples are shown in Fig. 23. These 61 functions have no overlap with the lookup table functions, but may overlap with the clone list. This indicates that about 1 in 6 high-MCC functions may be amenable to function-extraction refactoring, but at the same time, that developers prefer to replicate code rather than doing so.

8 Analysis of High-MCC Functions in Other Operating Systems and Domains

The discovery of high-MCC functions in Linux immediately raised the question whether this phenomenon is unique to Linux (with its free-for-all open source development methodology), or maybe such functions occur also in other systems and domains. This was research question 6.

To answer this, we analyzed the source code of three additional operating systems, and three open source systems from other domains. In the operating systems domain we chose Windows, FreeBSD, and OpenSolaris. From non-OS domains we chose GCC (compilers), Firefox (browsers), and the OpenSSL toolkit. The Windows Research Kernel (WRK) contains the source code for the NT-based kernel which is compatible with Windows Server 2003. Its source code includes core sources for object management, processes, threads, virtual memory, and the I/O system. It does not include Plug-and-Play, power management, virtual DOS machine, and the kernel debugger engine. The other five are the full codebases of FreeBSD, OpenSolaris, GCC, Firefox, and OpenSSL respectively.

Table 2 summarizes initial results of our analysis. We see that all these systems contain high-MCC functions with extreme values at their upper bound. For example, in the FreeBSD system the highest MCC value was 1316 which is 26 times higher than the highest threshold that was ever defined. It is true that the absolute number of these functions in each system is small, but they represent a non-negligible fraction of the control flow constructs in the


extractedLnx/linux-2.6.32/drivers/staging/rt2860/common/rtmp_init.c_RTMPReadTxPwrPerRate.c.ready

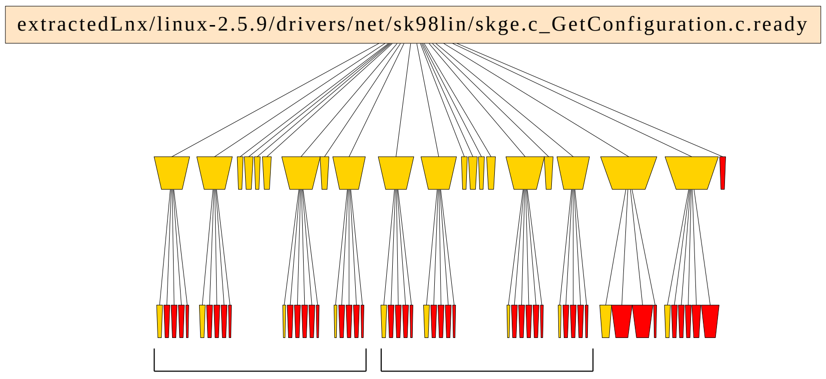

extractedLnx/linux-2.5.9/drivers/net/sk98lin/skge.c_GetConfiguration.c.ready

Fig. 23 *Functions with repeated structures that may be factored out.*

respective systems. This is listed in the table under ‘percent of MCC values’, meaning what fraction of the total MCC summed over all the functions in the system is contained in the high-MCC functions. For example, in the Windows system functions with an MCC above 50 account for more than 18% of the total MCC in the system.

According to Table 2 there are relatively few high-MCC functions and a large number of low-MCC functions. This observation indicates that the distribution of MCC values is skewed in all of the systems. An important class of skewed distributions are distributions with heavy tails. The common definition of heavy-tailed distributions is that their tail is governed by a power law, so $\Pr(X > x) \propto x^{-\alpha}$. To test the existence of a power-law tail one can use the log-log complementary distribution plot. This plot should produce a straight line for a perfect power law tail where its slope corresponds to the tail index α. Such plots for

Name	*Version*	*Total funcs*	*Max MCC*	*# high-MCC funcs* ≥ 100	> 50	*% of MCC values* ≥ 100	> 50
Windows	WRK-v1.2	4074	246	18	84	7.0	18.6
FreeBSD	9 (stable)	67528	1316	103	490	5.3	11.8
OpenSolaris	8	21259	506	34	202	4.2	11.6
Linux	2.6.37.5	259137	587	138	765	1.6	5.1
Firefox	9 (stable)	26444	699	27	181	3.3	9.9
GCC	4.8.0	72542	1301	248	938	10.2	21.3
OpenSSL	1.0.0k	6560	371	22	78	9.0	18.2

Table 2 High-MCC function characteristics of four operating systems and three open source projects from other domains.

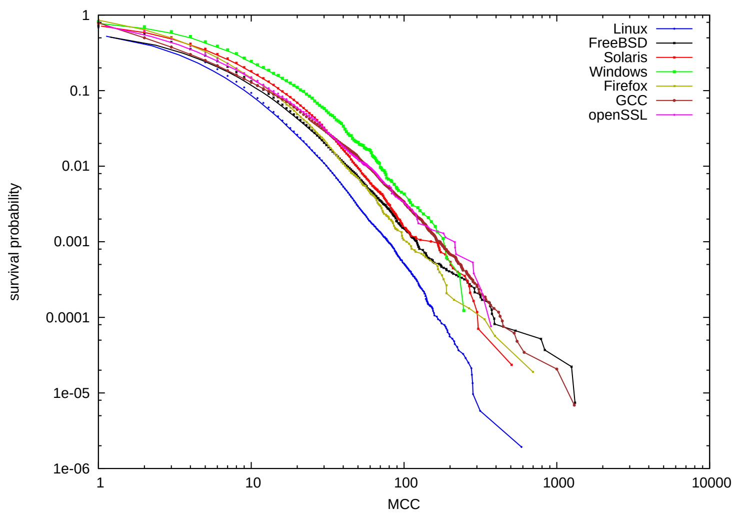


Fig. 24 *log-log complementary distribution plot of MCC values in four different systems.*

the MCC values of functions in our four opperating systems and three other applications are shown in Fig. 24. In all cases the lines look straight, and do not plunge downwards as they would for short-tail distributions. Because we are interested in the tail of the distribution we focused on the top 1% of the values and performed a linear regression. The results show that the tail indices for all the systems are a bit higher than 2, so these distributions have a bounded variance. The common definition of heavy tailed distributions requires a tail index in the range between 0 and 2, which leads to unbounded variance. Thus these distributions are a border case: they have a power law tail, but with a tail index of slightly more than 2.

9 Discussion and Conclusions

We have shown that the practice as reflected in the Linux kernel regarding large and complex functions diverges from common wisdom as reflected by thresholds used in various automatic tools for measuring MCC. This is not surprising, as a simplistic threshold cannot of course capture all the considerations involved in structuring the code. However, it does serve to point out an issue that deserves more thorough empirical research. We now turn to the implications of our findings.

9.1 MCC and Linux Quality

The basic underlying question we faced was whether the high-MCC functions in the Linux kernel constitute a code quality problem, or maybe such functions are actually acceptable and the warnings against them are exaggerated. This was our final research question, 7, and we can now discuss it based on all our findings.

Linux provides several examples where long and sometimes complex functions with a high MCC seem to be justified. It is of course possible to split such functions into a sequence of smaller functions, but this will be an artificial measure that only improves the MCC metric, and does not really improve the code. On the contrary, it may even be claimed that such artificial dissections degrade the code, by fragmenting pieces of code that logically belong together.

For example, one class of functions that tend to have very high MCC values are those that parse the options of some operation, in many cases the flag values of an ioctl (I/O control) system call for some device. There can be very many such flags, and the input parameter has to be compared to all of them. Once a match is found, the appropriate action is taken. Splitting the list of options into numerous shorter lists will just add clutter to the code.

Another class of functions that tend to have high MCC values are functions concerned with the emulation of hardware devices, typically belonging to unavailable (possibly legacy) architectures. The device may have many operations that each needs to be emulated, and furthermore this needs to take into account many different attributes of the device. Thus there are very many combinations that need to be handled, but partitioning them into meaningful subgroups may not be possible.

Despite the inherent size (and high MCC) of these functions, in many cases it may be claimed that they do not in fact cause a maintenance burden. This can happen either because they need not be maintained, or because they are actually not really complex.

As we saw in Section 4, more than a third of our functions exhibited no or negligible changes during the period of observation. In some of the other functions, which had larger changes, there was only a single large-change event. Thus most functions actually displayed strong stability the vast majority of the time. On average these functions do not require much effort to maintain.

Alternatively, functions with a high MCC may not really be so difficult to comprehend and maintain. MCC counts branch points in the code. If the cumulative effect of many branch points is to describe a complex combination of concerns, it may be hard for developers and maintainers to keep track of what is going on. But if the branching is used to separate concerns, as in the example of handling different flag values in an ioctl, this actually makes the code readable.

Our conclusion is therefore that for the most part the high-MCC functions found in Linux do not constitute a serious problem. On the contrary, they can serve as examples of situations where prevailing dogmas regarding code structure may need to be lifted.

9.2 Refinements to the MCC Metric

The observation that the MCC value of a function may not reflect "real" complexity as it is perceived by developers has been made before. Based on this, there have been suggestions to modify the metric to better reflect perceived complexity. Two previously suggested refinements are the following:

– Do not count cases in a large switch statement. This was mentioned already in McCabe's original paper [25], and is re-iterated in the MSDN documentation [28].
– Also do not count successive if statements, as successive decisions are not as complex as nested ones [12].

Both of these modifications together define McCabe's "essential" complexity metric, leading to a reduced value that assigns complexity only to more convoluted structures. But at the same time McCabe suggests a lower threshold of only 4 for this metric [26].

Generalizing the above, we suggest that one should not penalize "divide and conquer" constructs where the point is to distinguish between multiple *independent* actions. This may include nested decision trees in addition to switch statements and sequences of if statements. Note, however, that this refines the simple syntactic definition, as it is crucial to ensure that the individual conditions are indeed independent. For example, a switch statement in which a *non-empty* case falls through to the next case violates this independence, and thus adds complexity to the code.

The above suggestions are straightforward consequences of applying the principle of independence to basic blocks of code. However, this does not yet imply that they lead to any improvements in terms of measuring complexity. This would require a detailed study of code comprehension by human developers, which we leave for future work.

One more aspect that should be considered in MCC refinement is regularity. It is reasonable to think that regular functions need less effort to comprehend than irregular ones. As we have already seen compression algorithms tend to reflect the regularity extent in functions. This can be used to help in counterbalancing the exaggerated values of the MCC metric. In addition, we note based on our experience with Linux scheduling (e.g. [11]) that at least in some cases complexity is much more a result of how the logic of the code is expressed than a result of its syntactical structure. For example, even knowing the scheduling algorithm, it was hard to understand how the code implements this algorithm, despite the fact that its MCC was reasonably low. Thus syntactic metrics like MCC cannot be expected to give the full picture.

9.3 Threats to Validity

Our results are subject to several threats to validity.

Linux uses #ifdefs to enable configuration to different circumstances. Analyzing code that contains such directives may be problematic due to unbalanced braces. We are aware of this and dropped files that were tagged as syntactically incorrect by the pmccabe tool. In spite of their low percentage, these files may contain interesting functions with high MCC values that we would have missed.

While pmccabe is a well known tool for calculating MCC values, we found a bug in it: it counted the caret symbol (bitwise xor) as adding to the MCC value. We wrapped pmccabe with code that fixed this bug, and manually confirmed the results for selected functions. However, other bugs may exist in this and other tools.

In assessing the evolution of high-MCC functions, we actually rely on the MCC values. This is not necessarily right because a function may change without affecting the control constructs, or it may be that one construct was deleted but another was added. Thus our counts of changes may err on the conservative side. Our survey on perceived complexity also suffers from a few threats. For example, grading 92 functions within 2 hours is difficult and causes fatigue, which may affect the grading of the last functions. Moreover, a learning effect may also occur.

The survey of perceived complexity suffers from being subjective. It would have been good to also include some low-MCC functions in this survey, to see whether subjects distinguish between them and the high-MCC functions. In subsequent work we are also complementing this work by using a controlled experiment involving tasks related to code comprehension, specifically understanding, fixing bugs, and adding features [18].

Regarding external validity, we have verified that high-MCC functions exist also in other operating systems and in some specific systems from other domains, and are not unique to Linux. However, these are only preliminary results as we only examined one specific system from each domain. Also, our results are limited to systems coded in C, and do not necessarily generalize to systems written in an object-oriented style.

9.4 Future Work

One avenue for additional work is to assess the prevalence of high-MCC functions. It is plausible that an operating system kernel is more complex than most applications, due to the need to handle low-level operations. Although our results have shown that such functions also exists in other domains it would be interesting to repeat this study for more systems in these domains and even move to new domains.

Another important direction of additional research is empirical work on comprehension and how it correlates with MCC. This is especially needed in order to justify or refute suggested modifications to the metric, and indeed alternative metrics and considerations, and improve the ability to identify complex code. For example, our perceived complexity survey identified formatting and backwards gotos as factors that should most probably be taken into account. An interesting challenge is to try and see whether the functions with spaghetti gotos could have been written concisely in a more structured manner.

Regarding the correlation between perceived complexity and regularity as reflected by the Lempel-Ziv algorithm we think that retaining formatting attributes (such as indentation and linebreaks) besides the control structure is a reasonable direction as these attributes may affect regularity and perceived complexity.

Finally, in the context of studying Linux, the main drawback of our work is its focus on a purely syntactic complexity measure. It would be interesting to follow this up with semantic analysis, for example what happens to the functionality of high-MCC functions that seem to disappear into thin air. Thus this study may be useful in pointing out instances of interesting development activity in Linux.

References

1. B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan, "*Can we refactor conditional compilation into aspects?*" In 8th *Intl. Conf. Aspect-Oriented Softw. Dev.*, pp. 243–254, Mar 2009, doi: 10.1145/1509239.1509274.
2. R. Baggen, J. P. Correia, K. Schill, and J. Visser, "*Standardized code quality benchmarking for improving software maintainability*". *Softw. Quality J.* **20(2)**, pp. 287–307, Jun 2012, doi:10.1007/s11219-011-9144-9.
3. T. Ball and J. R. Larus, "*Using paths to measure, explain, and enhance program behavior*". *Computer* **33(7)**, pp. 57–65, Jul 2000, doi:10.1109/2.869371.
4. P. Bame, "*pmccabe*". URL http://parisc-linux.org/~bame/pmccabe/overview.html. (Visited 18 Sep 2011).
5. A. B. Binkley and S. R. Schach, "*Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures*". In 20th *Intl. Conf. Softw. Eng.*, pp. 452–455, Apr 1998, doi: 10.1109/ICSE.1998.671604.

6. A. Capiluppi and D. Izquierdo-Cortázar, "*Effort estimation of FLOSS projects: A study of the Linux kernel*". *Empirical Softw. Eng.* **18(1)**, pp. 60–88, Feb 2013, doi:10.1007/s10664-011-9191-7.
7. B. Curtis, J. Sappidi, and J. Subramanyam, "*An evaluation of the internal quality of business applications: Does size matter?*" In 33rd *Intl. Conf. Softw. Eng.*, pp. 711–715, May 2011, doi:10.1145/1985793.1985893.
8. B. Curtis, S. B. Sheppard, and P. Milliman, "*Third time charm: Stronger prediction of programmer performance by software complexity metrics*". In 4th *Intl. Conf. Softw. Eng.*, pp. 356–360, Sep 1979.
9. G. Denaro and M. Pezzè, "*An empirical evaluation of fault-proneness models*". In 24th *Intl. Conf. Softw. Eng.*, pp. 241–251, May 2002, doi:10.1145/581339.581371.
10. E. W. Dijkstra, "*GoTo statement considered harmful*". *Comm. ACM* **11(3)**, pp. 147–148, Mar 1968, doi:0.1145/362929.362947.
11. Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Process prioritization using output production: scheduling for multimedia*". *ACM Trans. Multimedia Comput., Commun. & App.* **2(4)**, pp. 318–342, Nov 2006, doi:10.1145/1201730.1201734.
12. W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "*Applying software complexity metrics to program maintenance*". *Computer* **15(9)**, pp. 65–79, Sep 1982, doi:10.1109/MC.1982.1654138.
13. I. Heitlager, T. Kuipers, and J. Visser, "*A practical model for measuring maintainability*". In 6th *Intl. Conf. Quality Inf. & Comm. Tech.*, pp. 30–39, Sep 2007, doi:10.1109/QUATIC.2007.8.
14. I. Herraiz and A. E. Hassan, "*Beyond lines of code: Do we need more complexity metrics?*" In *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson (eds.), pp. 125–141, O'Reilly Media Inc., 2011.
15. A. Hindle, M. W. Godfrey, and R. C. Holt, "*Reading beside the lines: Indentation as a proxy for complexity metrics*". In 16th *IEEE Intl. Conf. Program Comprehension*, pp. 133–142, Jun 2008, doi:10.1109/ICPC.2008.13.
16. A. Israeli and D. G. Feitelson, "*The Linux kernel as a case study in software evolution*". *J. Syst. & Softw.* **83(3)**, pp. 485–501, Mar 2010, doi:10.1016/j.jss.2009.09.042.
17. A. Jbara and D. G. Feitelson, "*Characterization and assessment of the Linux configuration complexity*". In 13th *IEEE Intl. Working Conf. Source Code Analysis & Manipulation*, Sep 2013.
18. A. Jbara and D. G. Feitelson, "*Code regularity may compensate for high MCC and LOC: Initial results*". 2013. (In preparation).
19. A. Jbara, A. Matan, and D. G. Feitelson, "*High-MCC functions in the Linux kernel*". In 20th *IEEE Intl. Conf. Program Comprehension*, pp. 83–92, Jun 2012, doi:10.1109/ICPC.2012.6240512.
20. C. Jones, "*Software metrics: Good, bad, and missing*". *Computer* **27(9)**, pp. 98–100, Sep 1994, doi:10.1109/2.312055.
21. H. Koziolek, B. Schlich, and C. Bilich, "*A large-scale industrial case study on architecture-based software reliability analysis*". In 21st *Intl. Symp. Software Reliability Eng.*, pp. 279–288, Nov 2010, doi:10.1109/ISSRE.2010.15.
22. D. L. Lanning and T. M. Khoshgoftaar, "*Modeling the relationship between source code complexity and maintenance difficulty*". *Computer* **27(9)**, pp. 35–40, Sep 1994, doi:10.1109/2.312036.
23. M. M. Lehman and J. F. Ramil, "*Software evolution—background, theory, practice*". *Inf. Process. Lett.* **88(1-2)**, pp. 33–44, Oct 2003, doi:10.1016/S0020-0190(03)00382-X.
24. J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "*An analysis of the variability in forty preprocessor-based software product lines*". In 32nd *Intl. Conf. Softw. Eng.*, vol. 1, pp. 105–114, May 2010, doi:10.1145/1806799.1806819.
25. T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **2(4)**, pp. 308–320, Dec 1976, doi:10.1109/TSE.1976.233837.
26. McCabe Software, "*Metrics & thresholds in McCabe IQ*". URL www.mccabe.com/pdf/McCabe%20IQ%20Metrics.pdf, undated. (Visited 23 Dec 2009).
27. T. Mens, J. Fernández-Ramil, and S. Degrandsart, "*The evolution of Eclipse*". In *Intl. Conf. Softw. Maintenance*, pp. 386–395, Sep 2008, doi:10.1109/ICSM.2008.4658087.
28. MSDN Visual Studio Team System 2008 Development Developer Center, "*Avoid excessive complexity*". URL msdn.microsoft.com/en-us/library/ms182212.aspx, undated. (Visited 23 Dec 2009).
29. G. J. Myers, "*An extension to the cyclomatic measure of program complexity*". *SIGPLAN Notices* **12(10)**, pp. 61–64, Oct 1977, doi:10.1145/954627.954633.
30. N. Nagappan, T. Ball, and A. Zeller, "*Mining metrics to predict component failures*". In 28th *Intl. Conf. Softw. Eng.*, pp. 452–461, May 2006, doi:10.1145/1134285.1134349.
31. N. Ohlsson and H. Alberg, "*Predicting fault-prone software modules in telephone switches*". *IEEE Trans. Softw. Eng.* **22(12)**, pp. 886–894, Dec 1996, doi:10.1109/32.553637.

32. H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "*Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes*". *IEEE Trans. Softw. Eng.* **33(6)**, pp. 402–419, Jun 2007, doi: 10.1109/TSE.2007.1015.
33. P. Oman and J. Hagemeister, "*Construction and testing of polynomials predicting software maintainability*". *J. Syst. & Softw.* **24(3)**, pp. 251–266, Mar 1994, doi:10.1016/0164-1212(94)90067-1.
34. F. Sauer, "*Eclipse metrics plugin 1.3.6*". URL metrics.sourceforge.net/, Jul 2005. (Visited 23 Dec 2009).
35. N. Schneidewind and M. Hinchey, "*A complexity reliability model*". In 20th *Intl. Symp. Software Reliability Eng.*, pp. 1–10, Nov 2009, doi:10.1109/ISSRE.2009.10.
36. M. Shepperd, "*A critique of cyclomatic complexity as a software metric*". *Software Engineering J.* **3(2)**, pp. 30–36, Mar 1988, doi:10.1049/sej.1988.0003.
37. M. Shepperd and D. C. Ince, "*A critique of three metrics*". *J. Syst. & Softw.* **26(3)**, pp. 197–210, Sep 1994, doi:10.1016/0164-1212(94)90011-6.
38. Q. D. Soetens and S. Demeyer, "*Studying the effect of refactorings: A complexity metrics perspective*". In 7th *Intl. Conf. Quality Inf. & Comm. Tech.*, pp. 313–318, Sep 2010, doi:10.1109/QUATIC.2010.58.
39. E. Soloway and K. Ehrlich, "*Empirical studies of programming knowledge*". *IEEE Trans. Softw. Eng.* **SE-10(5)**, pp. 595–609, Sep 1984, doi:10.1109/TSE.1984.5010283.
40. SRI, "*Software technology roadmap: Cyclomatic complexity*". In URL www.sei.cmu.edu/str/str.pdf, 1997. (Visited 28 Dec 2008).
41. I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "*Code quality analysis in open source software development*". *Inf. Syst. J.* **12(1)**, pp. 43–60, Jan 2002, doi:10.1046/j.1365-2575.2002.00117.x.
42. G. Stark, R. C. Durst, and C. W. Vowell, "*Using metrics in management decision making*". *Computer* **27(9)**, pp. 42–48, Sep 1994, doi:10.1109/2.312037.
43. R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "*Comparative analysis of evolving software systems using the Gini coefficient*". In 25th *Intl. Conf. Softw. Maintenance*, pp. 179–188, Sep 2009, doi: 10.1109/ICSM.2009.5306322.
44. VerifySoft Technology, "*McCabe metrics*". URL www.verifysoft.com/en_mccabe_metrics.html, Jan 2005. (Visited 23 Dec 2009).
45. E. J. Weyuker, "*Evaluating software complexity measures*". *IEEE Trans. Softw. Eng.* **14(9)**, pp. 1357–1365, Sep 1988, doi:10.1109/32.6178.
46. J. Ziv and A. Lempel, "*Compression of individual sequences via variable-rate coding*". *IEEE Trans. Information Theory* **IT-24(5)**, pp. 530–536, Sep 1978, doi:10.1109/TIT.1978.1055934.