

High-MCC Functions in the Linux Kernel

Ahmad Jbara Adam Matan Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

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. We attempt to explain this by analyzing the structure of such functions and showing that in many cases they are in fact well-structured. At the same time, we observe cases where developers indeed refactor the code in order to reduce complexity. These observations highlight the need to define more holistic notions of complexity, rather than using simple syntactic code metrics.

Keywords—Software Complexity, McCabe Cyclomatic Complexity, Linux Kernel

I. 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 [12], [3]. 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 [20], [5], [21]. Industrial testimony indicates that using complexity metrics provides real benefits over simple practices such as just counting lines of code (e.g. [11], [4]).

One early metric that has been used in many studies is McCabe’s Cyclomatic Complexity (MCC) [15]. This metric essentially counts the number of linear paths through the code (the precise definition is given below in Section II). 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 slightly higher thresholds have also been suggested by others [18], [26], [27], [29]. However, these are limited to about 50, and there appears to be some agreement that procedures with much higher values are extremely undesirable.

Nevertheless, we have found functions with MCC values in the hundreds in the Linux kernel [10]. This chance discovery immediately led to a set of research questions:

- 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?
- 2) What causes the high MCC counts? One may speculate that the high MCC counts 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.

- 3) 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.
- 4) Does a high MCC correlate with perceived complexity? In other words, does MCC indeed capture the essence of complexity? What other ingredients may be missing?
- 5) Altogether, do the high MCC functions indicate code quality problems with the Linux kernel?

To gain insight into these issues we analyzed the 100 highest MCC functions in kernel version 2.6.37.5, which turn out to have MCC values ranging from 112 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 a nutshell, we found that the most common source of high MCC counts is large trees of if statements, although several cases are indeed attributed to large switches. 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 switches 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. 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.

The remainder of the paper is structured as follows. In the next section we define MCC and review its use. Our findings concerning the Linux kernel are described in Sections III through V, roughly corresponding to the research questions above, and their significance is discussed in Section VI, which also identifies further research directions.

II. 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 [15]. 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. Structured programming requires that all constructs have single entry and exit points. The control-flow graph is then planar. For such planar graphs, the cyclomatic number is equal to the number of faces of the graph, including the "outside" area.

Importantly, McCabe also demonstrated a straight-forward intuitive meaning of the metric: it turns out to be equal to the number of predicates in the program plus 1. Here "predicate" should be understood as the full condition that is checked in an if or while statement. If this is actually composed of multiple atomic predicates, we could also count them individually; this approach is sometimes called the "extended" MCC [19]. Note that the metric counts points of divergence, but not joins. It is thus insensitive to unconditional jumps such as those induced by `goto`, `break`, or `return`.

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 [15], and this is also the value used by the code analysis tool sold by his company today [16]. The Eclipse Metrics plugin also uses a threshold of 10 by default, and suggests that the method be split if it is exceeded [22]. VerifySoft Technology suggest a threshold of 15 per function, and 100 per file [29]. The Logiscope tool also uses a threshold of 15 [27]. The STAN static analysis tool gives a warning at 15, and considers values above 20 as an error [17]. The complexity metrics module of Microsoft Visual Studio 2008 reports a violation of the cyclomatic complexity

metric if a value of more than 25 is found [18]. The Carnegie Mellon Software Engineering Institute defined a four-level scale as part of their (now legacy) Software Technology Roadmap [26]. High risk was associated with values of MCC above 20, and very high risk with values larger than 50.

All the above thresholds consider functions in isolation (except for VerifySoft who also suggest a threshold on the sum of all functions in the same file). An alternative approach is to consider the distribution of MCC values. This was suggested by Stark et al. as the basis for a decision chart that plots the 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 [28]; 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.

It should be noted that MCC is not universally accepted as a good complexity metric. In fact it has been challenged on both theoretical and experimental grounds. 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 [1]. Others show that MCC is strongly correlated with lines of code, or point out that it only measures control flow complexity but not data flow complexity [23], [30], [24]. There is, however, no other complexity metric that enjoys wider acceptance and is free of such criticisms. Given its wide use and availability in software development and testing environments, MCC therefore merits an effort to understand it better.

III. ANALYSIS OF HIGH MCC FUNCTIONS IN LINUX

In the process of 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 [10]. Specifically, we found that the distribution of MCC values has a heavy tail, the absolute number of high-MCC functions is growing, but their fraction out of all functions in Linux is shrinking. 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.

To calculate the MCC we use the `pmccabe` tool [2]. This tool calculates the extended MCC, i.e. it also counts instances of logical operators in predicates (`&&` and `||`). 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 [14]. As we want to analyze the full code base and not just a specific configuration, we ignore such directives and attempt

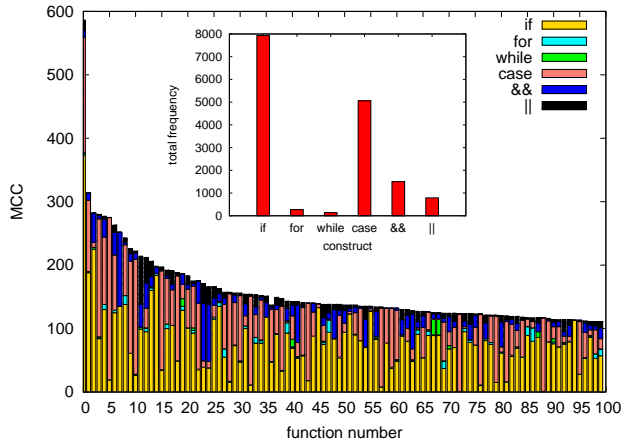


Figure 1. Distribution of constructs in high-MCC functions. Inset shows sums over all 100 functions.

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.

A. Statistics of High MCC

The 100 functions with highest MCC values in Linux kernel 2.6.37.5 have values ranging from 112 to 587. 76 of these functions come from the drivers subdirectory, with others coming from arch (8 functions), fs (9 functions), sound (3 functions), net (3 functions), and crypto (1 function).

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 elsels. Logical operators, which can also 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.

B. 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 we introduce control structure diagrams (CSD) to visualize the control structure and nesting.

In these graphs (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

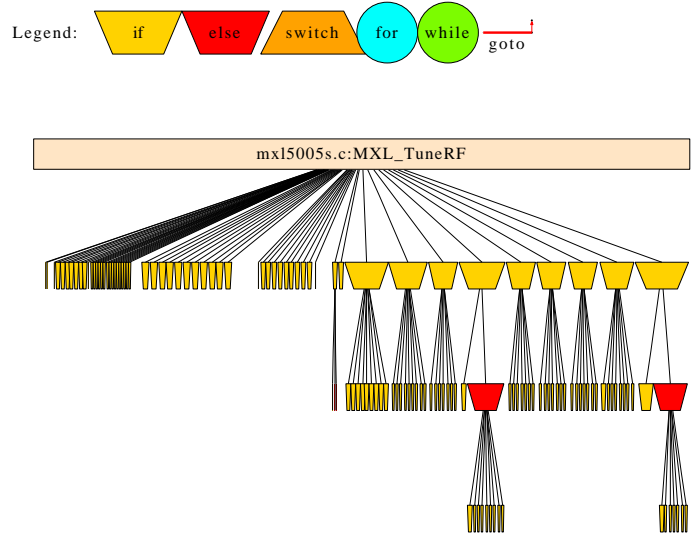


Figure 2. A function that is a largely flat sequence of ifs.

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 code¹. 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.

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 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 [6].

¹Graphs for all the functions are available at <http://www.cs.huji.ac.il/~ahmadjbara/hiMCCFuncs.html>

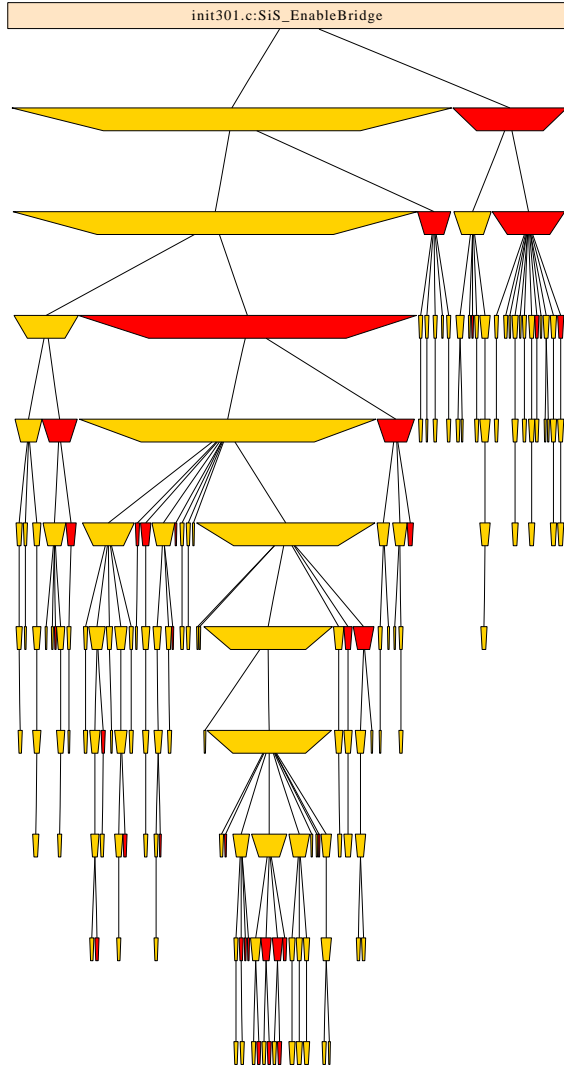


Figure 3. A function with irregular ifs and relatively deep nesting.

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.

C. Correlation of MCC with Other Metrics

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. We checked this on our sample of 100 functions, distinguishing between PLOC, the raw number of lines, and LLOC, the non-comment non-blank

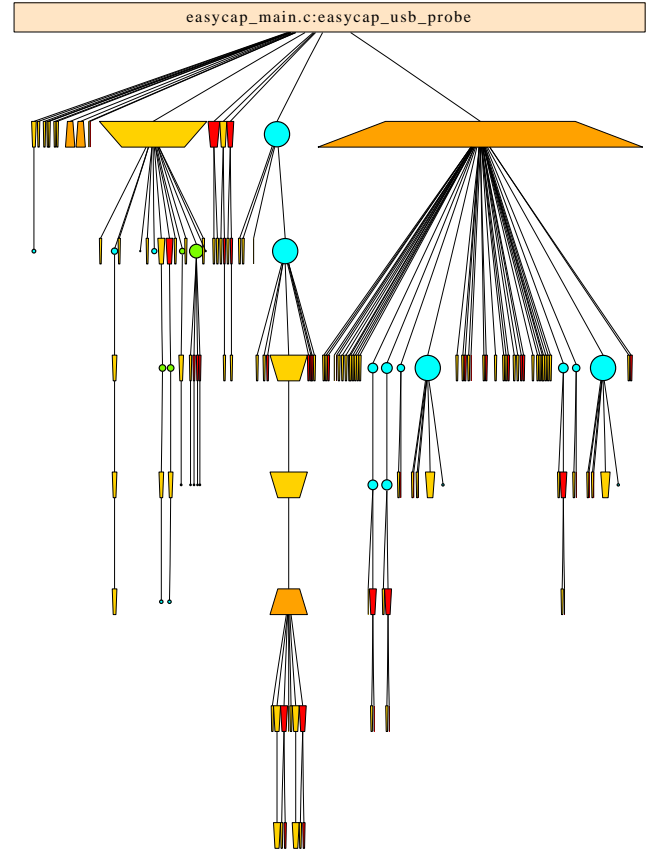


Figure 4. A function with relatively many loops.

lines of code. The results are shown in Fig. 6. While this indicates that indeed the two metrics are correlated, this is far from being a simple linear relationship, and in fact some functions have a relatively low MCC but high LOC, or vice versa. Moreover, the relatively high correlation coefficients are largely attributed to the single large outlier function. After removing this function, they drop from 0.728 and 0.688 to 0.485 and 0.448, respectively.

The correlation of MCC with LOC means that MCC can be anticipated to some degree by code volume. A different question is whether it 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 [9]. 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. 7. Obviously there is almost no correlation of MCC with the average level of indentation or nesting in each function (even if the outlier is removed). These results reflect our findings regarding the structure of high-MCC functions, namely that they could be either flat switches and sequences of ifs, or else deep trees of nested ifs.

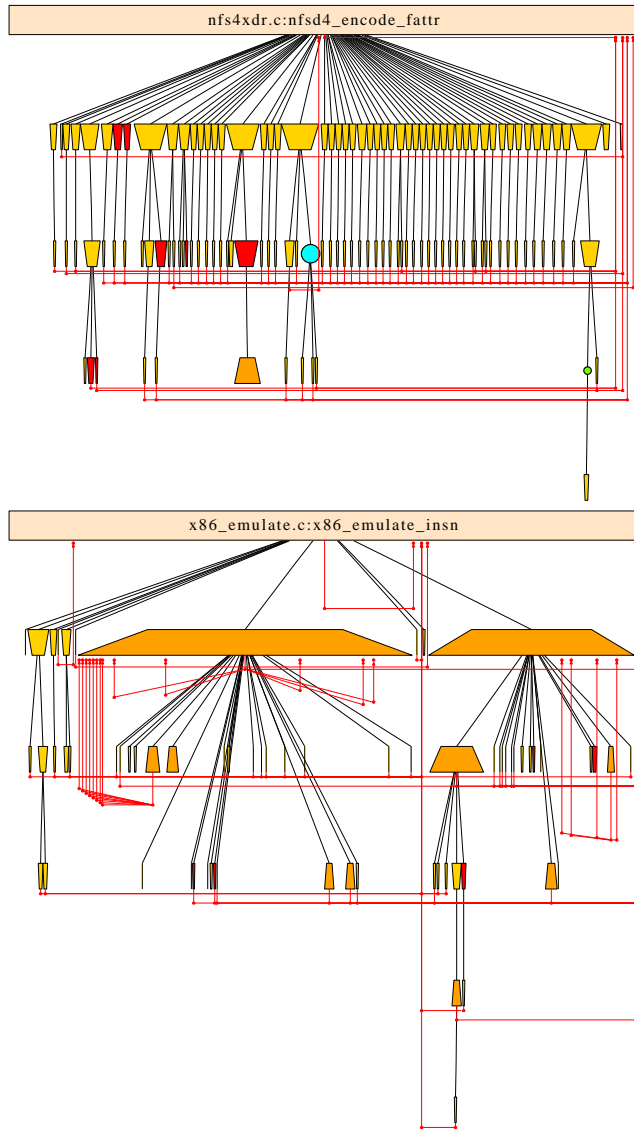


Figure 5. Examples of functions using goto.

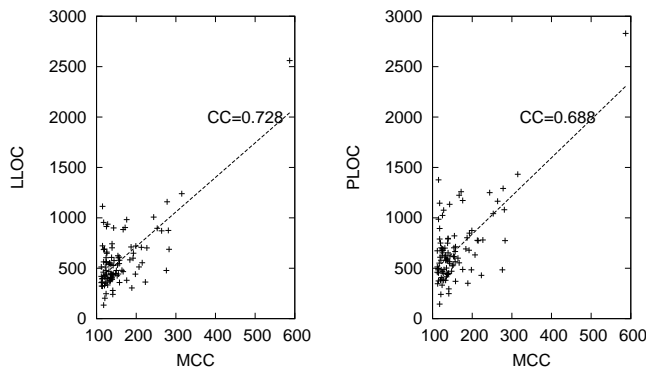


Figure 6. Correlation of MCC with LLOC and PLOC.

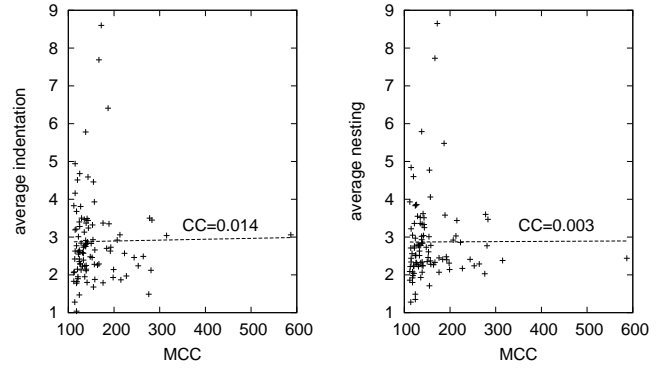


Figure 7. Correlation of MCC with indentation and nesting.

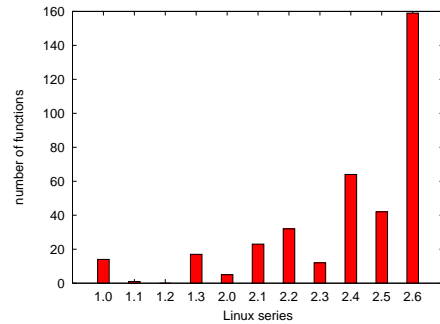


Figure 8. The distribution of new high-MCC functions in Linux series.

IV. MAINTENANCE AND EVOLUTION OF HIGH-MCC FUNCTIONS

Linux is an evolving system [10]. 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 [13]. Obviously, most of the functions in the current release didn’t exist in the first release—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. 159 new functions with MCC above 100 were introduced during the 2.6 series, as shown in Fig. 8. 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. 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. The

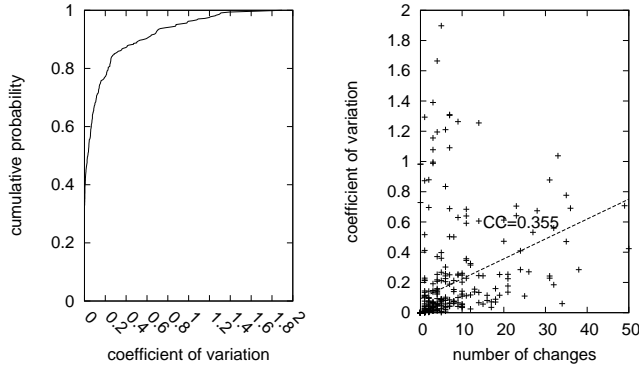


Figure 9. Left: The distribution of the coefficient of variation of the MCC of 369 high-MCC functions. Right: Scatter plot showing relationship between number of times the MCC changed and the degree of change as measured by the coefficient of variation.

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 it's MCC changes significantly with time, its CV can reach a value of 1 or even more. Fig. 9 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. Finally, some functions exhibited significant changes in their MCC. Examples are shown in Fig. 10².

The degree to which the MCC changes is only one side of the story. In principle a very large change may occur, but this is a one-time event. 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 in the middle of the previous branch, not at its end.

Fig. 9 shows a scatter plot that compares the degree of change with the number of changes. The correlation between these two metrics is weak, and in fact functions that change many times have $CV < 1$, whereas functions with high CV typically change no more than 10 times. Also, 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

²In 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, i.e. 16 means 2.6.16.

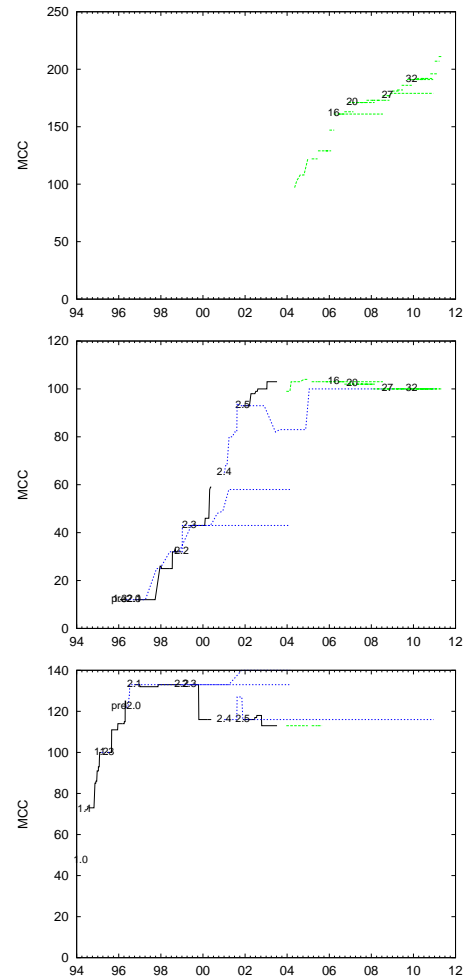


Figure 10. Examples of functions that exhibit significant changes over time: *cifs_parse_mount_options*, *vortex_probe1*, and *st_int_ioctl*

50 times. We also calculated the correlation of MCC with the CV and number of changes. Interestingly, these were low *positive* correlations (0.214 and 0.104 respectively, without the outlier). Thus functions with high MCC tend to be changed more.

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. 11 and *vortex_probe1* from Fig. 10). At least in some of these cases it seems that the change was done in a production version

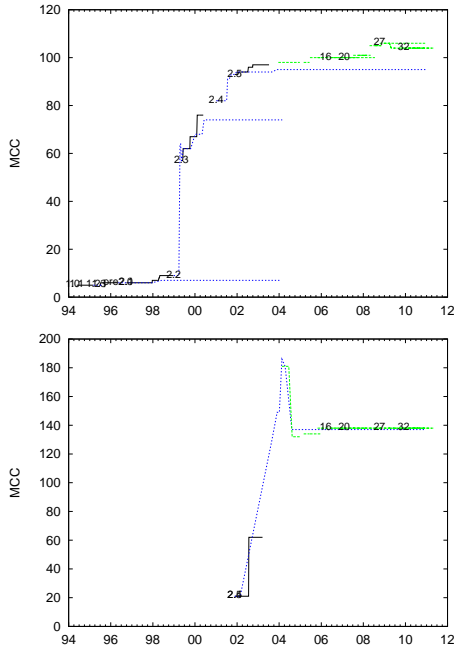


Figure 11. Examples of functions that exhibit large changes in production versions: `sg_ioctl` and `SiS_EnableBridge`.

during the interval between two successive development versions. Such behavior contradicts the “official” semantics of development vs. production 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. 12. 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 [10]. 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, in version 2.2.14 the function `isdn_tty_cmd_PLUSF_FAX` had MCC 154. 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. There were also cases where the whole function just moved to another location, possibly undergoing revisions in the process. Another example is function `fd_ioctl_trans`, where the original function had many long compound if statements with heavy use of the or operator. In its reduced MCC version the or operator was replaced by

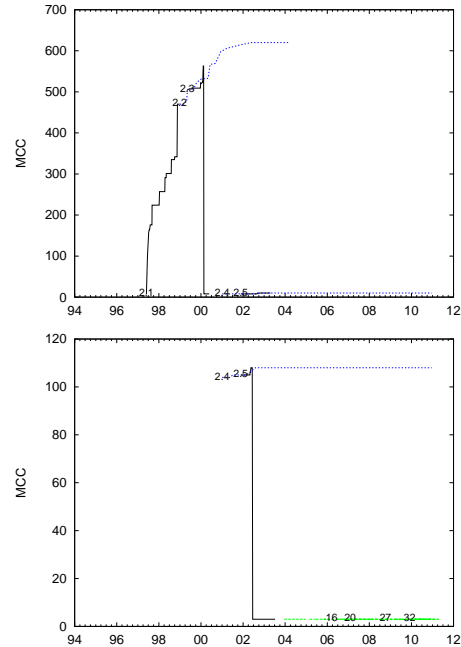


Figure 12. Examples of functions that exhibit a sharp drop in MCC resulting from a design change: `sys32_ioctl` and `usb_stor_show_sense`.

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

Another interesting phenomenon that occurred during maintenance was co-evolution. This occurs when two related functions, e.g. `do_mathemu` in `/arch/sparc64/mathemu/math.c` and `do_one_mathemu` in `/arch/sparc/mathemu/math.c`, 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 soon after propagated to

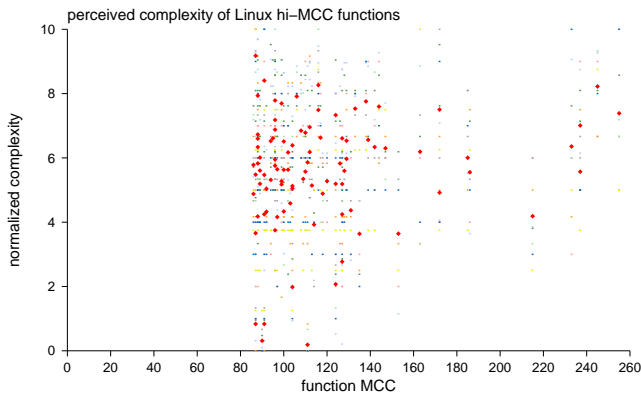


Figure 13. Scatter plot showing relationship between measured MCC and perceived complexity. The small markings are individual grades, and the average grade for each function is marked by a larger diamond.

the contemporaneous production version.

V. 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.

To gain some insight into this question, 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 8 participants of a summer Linux kernel workshop. The goal was to identify notions of perceived complexity, not to quantify the actual effect of complexity. 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 grade its perceived complexity on a personal scale, where a higher grade signifies higher perceived complexity. These scales were then normalized to the range 0 to 10, and average grades were computed. The order in which the functions were presented was randomized.

The results, shown in Fig. 13, indicate little correlation between MCC and perceived complexity. 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 paragraphs we focus on these extreme grades.

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

```
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";
}
```

Figure 14. Example of simple *switch* structure from *log_audio_status*.

in each case as illustrated in Fig. 14. These simple and short functions also contributed to the somewhat stronger correlation of perceived complexity with LOC than MCC with LOC.

In addition to these single-instruction cases, survey participants also 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). In addition, repeated use of the same code template, e.g. in a long sequence of small if trees that all have exactly the same structure, also reduces the perceived complexity considerably. An example is shown in Fig. 15.

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.

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. 16. These observations hark back to the work of Soloway and Ehrlich [25], 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.

VI. DISCUSSION AND CONCLUSIONS

We have shown that the practice as reflected in the Linux kernel regarding large and complex functions diverges from

```

bytes.high = 0x00;
bytes.low = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_1[7];
if (!daa_load(&bytes, j))
    return 0;

bytes.high = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_1[6];
bytes.low = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_1[5];
if (!daa_load(&bytes, j))
    return 0;

bytes.high = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_1[4];
bytes.low = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_1[3];
if (!daa_load(&bytes, j))
    return 0;

bytes.high = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_1[2];
bytes.low = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_1[1];
if (!daa_load(&bytes, j))
    return 0;

bytes.high = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_1[0];
bytes.low = 0x00;
if (!daa_load(&bytes, j))
    return 0;

if (!SCI_Control(j, SCI_End))
    return 0;
if (!SCI_WaitLowSCI(j))
    return 0;

bytes.high = 0x01;
bytes.low = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_2[7];
if (!daa_load(&bytes, j))
    return 0;

bytes.high = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_2[6];
bytes.low = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_2[5];
if (!daa_load(&bytes, j))
    return 0;

bytes.high = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_2[4];
bytes.low = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_2[3];
if (!daa_load(&bytes, j))
    return 0;

bytes.high = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_2[2];
bytes.low = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_2[1];
if (!daa_load(&bytes, j))
    return 0;

bytes.high = j->m_DAAShadowRegs.COP_REGS.COP.THFilterCoeff_2[0];
bytes.low = 0x00;
if (!daa_load(&bytes, j))
    return 0;

if (!SCI_Control(j, SCI_End))
    return 0;
if (!SCI_WaitLowSCI(j))
    return 0;

```

Figure 15. Example of a sequence of independent ifs with the same structure, from `ixj_daa_write`. The full function includes 113 such ifs.

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, as high MCC functions are widespread. We now turn to the implications of our findings.

A. MCC and Linux Quality

The basic underlying question we faced was whether the high-MCC functions in the Linux kernel constitute a

```

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);
}

```

Figure 16. Example of excessive line breaks that seem to make the code harder rather than easier to understand, from `balance_leaf`.

code quality problem, or maybe such functions are actually acceptable and the warnings against them are exaggerated.

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 IV, 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.

B. 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:

- Don’t count cases in a large switch statement. This was mentioned already in McCabe’s original paper [15], and is re-iterated in the MSDN documentation [18].
- Also don’t count successive if statements, as successive decisions are not as complex as nested ones [8].

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 [16].

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.

In addition, we note based on our experience with Linux scheduling (e.g. [7]) that at least in some cases complexity is much more a result of the logical involvement of the code than a result of its syntactical structure. Thus syntactic metrics like MCC cannot be expected to give the full picture.

C. 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 is 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 may miss.

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.

Finally, all the results are naturally only true for Linux. Generalizations to other systems need to be checked.

D. 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. Thus it would be interesting to repeat this study for other large-scale support and infrastructure software (such as compilers) and user-level applications (such as web browsers).

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 the 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.

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. Investigating them may shed light on the dynamics of the development process as a whole.

REFERENCES

- [1] T. Ball and J. R. Larus, “Using paths to measure, explain, and enhance program behavior”. *Computer* **33(7)**, pp. 57–65, Jul 2000.
- [2] P. Bame, “pmccabe”. URL <http://parisc-linux.org/~bame/pmccabe/overview.html>. (Visited 18 Sep 2011).
- [3] 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.
- [4] 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.
- [5] G. Denaro and M. Pezzè, “An empirical evaluation of fault-proneness models”. In *24th Intl. Conf. Softw. Eng.*, pp. 241–251, May 2002.
- [6] E. W. Dijkstra, “GoTo statement considered harmful”. *Comm. ACM* **11(3)**, pp. 147–148, Mar 1968.
- [7] Y. Etsion, D. Tsafir, 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.
- [8] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, “Applying software complexity metrics to program maintenance”. *Computer* **15(9)**, pp. 65–79, Sep 1982.
- [9] 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*, Jun 2008.
- [10] 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.
- [11] C. Jones, “Software metrics: Good, bad, and missing”. *Computer* **27(9)**, pp. 98–100, Sep 1994.
- [12] 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.
- [13] M. M. Lehman and J. F. Ramil, “Software evolution—background, theory, practice”. *Inf. Process. Lett.* **88(1-2)**, pp. 33–44, Oct 2003.
- [14] 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.
- [15] T. McCabe, “A complexity measure”. *IEEE Trans. Softw. Eng.* **2(4)**, pp. 308–320, Dec 1976.
- [16] McCabe Software, “Metrics & thresholds in McCabe IQ”. URL www.mccabe.com/pdf/McCabe%20IQ%20Metrics.pdf, undated. (Visited 23 Dec 2009).
- [17] T. Mens, J. Fernández-Ramil, and S. Degrandart, “The evolution of Eclipse”. In *Intl. Conf. Softw. Maintenance*, pp. 386–395, Sep 2008.
- [18] 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).
- [19] G. J. Myers, “An extension to the cyclomatic measure of program complexity”. *SIGPLAN Notices* **12(10)**, pp. 61–64, Oct 1977.
- [20] N. Ohlsson and H. Alberg, “Predicting fault-prone software modules in telephone switches”. *IEEE Trans. Softw. Eng.* **22(12)**, pp. 886–894, Dec 1996.
- [21] 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.
- [22] F. Sauer, “Eclipse metrics plugin 1.3.6”. URL metrics.sourceforge.net/, Jul 2005. (Visited 23 Dec 2009).
- [23] M. Shepperd, “A critique of cyclomatic complexity as a software metric”. *Software Engineering J.* **3(2)**, pp. 30–36, Mar 1988.
- [24] M. Shepperd and D. C. Ince, “A critique of three metrics”. *J. Syst. & Softw.* **26(3)**, pp. 197–210, Sep 1994.
- [25] E. Soloway and K. Ehrlich, “Empirical studies of programming knowledge”. *IEEE Trans. Softw. Eng.* **SE-10(5)**, pp. 595–609, Sep 1984.
- [26] SRI, “Software technology roadmap: Cyclomatic complexity”. In URL www.sei.cmu.edu/str/str.pdf, 1997. (Visited 28 Dec 2008).
- [27] 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.
- [28] G. Stark, R. C. Durst, and C. W. Vowell, “Using metrics in management decision making”. *Computer* **27(9)**, pp. 42–48, Sep 1994.
- [29] VerifySoft Technology, “McCabe metrics”. URL www.verifysoft.com/en_mccabe_metrics.html, Jan 2005. (Visited 23 Dec 2009).
- [30] E. J. Weyuker, “Evaluating software complexity measures”. *IEEE Trans. Softw. Eng.* **14(9)**, pp. 1357–1365, Sep 1988.