Characterizing Software Maintenance Categories Using the Linux Kernel

Ayelet Israeli Dror G. Feitelson
Department of Computer Science
The Hebrew University, 91904 Jerusalem, Israel

***Abstract*—Software maintenance involves different categories of activities: corrective, adaptive, perfective, and preventive. However, research regarding these distinct activities is hampered by lack of empirical data that is labeled to identify the type of maintenance being performed. A promising dataset is provided by the more than 800 releases of the Linux kernel that have been made since 1994. The Linux release scheme differentiates between development versions (which may be expected to undergo perfective maintenance) and production versions (probably dominated by corrective maintenance). The structure of the codebase also distinguishes code involved in handling different architectures and devices (where additions reflect adaptive maintenance) from the core of the system's kernel. Assuming that these dissections of the codebase indeed reflect different types of activity, we demonstrate that corrective maintenance does not necessarily lead to code deterioration, that adaptive maintenance may improve some quality metrics, and that growth is largely the result of continued development as part of perfective maintenance.**

***Index Terms*—Perfective maintenance, Corrective maintenance, Adaptive maintenance, Linux kernel**

I. Introduction

Software maintenance is an active field of research enjoying growing recognition. To quote Parnas, "A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products" [20]. The Linux kernel provides a large-scale example of software maintenance[1]. It was first released in March 1994, and more than 800 releases have been made since then in multiple branches. Moreover, Linux is open source, and all the code of all the releases is freely available on the Internet (from www.kernel.org and numerous mirrors). This makes it suitable for a study of software maintenance at a scale and level of detail that has not been done before.

Software maintenance activities are commonly classified into four categories: corrective, adaptive, perfective, and preventive [26], [1]. To study these different activities, one must be able to classify commits of maintained code [8]. As an alternative, we make the observation that the first three categories map naturally to different parts of the cumulative Linux code releases, based on the Linux release scheme. In particular, we expect development versions to reflect perfective maintenance, production versions to reflect corrective maintenance, and the arch and drivers directories to reflect adaptive maintenance (these mappings are explained in more detail below).

Thus, assuming one accepts this dissection of the code, we can empirically characterize different types of maintenance activities, and how they affect the body of software being maintained. At the same time, we also note that the dissection cannot be pure, and sometimes one observes different types of activity applied to the same version of the code. Nevertheless, the maintenance activities observed on different parts of the code do indeed appear to be different in general, suggesting that the dissection is robust enough to make general observations about different maintenance activities.

Our results are naturally limited to these dissections of the versions and code, and may not generalize to other systems. However, Linux has served as a source of data for multiple studies of software development, and is undoubtedly interesting and important enough to be used as a case study. Our results may also generalize to other open and even closed source projects, but verifying this requires additional study.

The organization of this paper is as follows. The next section provides background regarding maintenance categories and how we relate them to the Linux kernel, as well as our methodology. Section III presents our measurements of how code metrics change with releases of the Linux kernel. This is then interpreted with regard to maintenance activities in Section IV. Section V presents threats to validity, and Section VI concludes and suggests future work.

II. Background

A. The Linux Kernel

The Linux operating system kernel was originally announced on the Internet in August 1991. There followed $2\frac{1}{2}$ years of development by a growing community or developers, and the first production version was released in March 1994. A 3-digit system was used to identify releases. The first is the generation, which changed from 1 to 2 in 1996. The second is the major kernel version number. Importantly, a distinction was made between even major numbers (1.0, 1.2, 2.0, 2.2, and 2.4) which represent stable production versions, and odd major numbers (1.1, 1.3, 2.1, 2.3, and 2.5) which are development versions used to test new features and drivers leading up to the next stable version. The third digit is the minor kernel version number. Releases with new minor numbers of production versions supposedly included only bug fixes and security patches, whereas new releases of development

[1]At better term for at least part of this activity is "software evolution", but in this paper we use "maintenance" in accordance with the common usage in the literature to describe the categorization of maintenance activities described below.

versions included new (but not fully tested) functionality. Importantly, several versions (one development and one or more production) may be "current" at the same time.

The problem with this scheme was the long lag time until new functionality (and new drivers) were released, as this was supposed to happened only on the next major version release. The scheme was therefore changed with the 2.6 kernel in December 2003. Initially, releases were simply managed at a relatively high rate. Then, with version 2.6.11, a fourth number was added. The third number now indicates new releases with added functionality, whereas the fourth number indicates bug fixes and security patches.

The Linux kernel sources are arranged in several major subdirectories [21]. Two notable ones for our purposes are the arch and drivers directories. These are practically external to the core of the kernel, and each specific kernel configuration uses only a small portion of these directories. However, these directories constitute a major part of the code, which grows not only due to improvements and addition of features, but also due to the need to support additional processors and devices.

B. Categories of Software Maintenance

"Maintenance" is used to describe whatever happens to a software product after the first release. According to the literature at least 50% of the effort invested in software, and perhaps 80% or even more, is dedicated to maintenance [15], [11], [13]. Maintenance is classified into four categories [26]:

Corrective: correction of discovered problems (fixing bugs).
Adaptive: adapting to changes in the environment.
Perfective: improving performance or maintainability.
Preventive: detection and correction of latent faults before they become effective faults.

The term "maintenance" by nature implies a preservation of the existing. However, software often undergoes further development after being released. To accommodate this, the addition of new features is usually included as part of perfective maintenance (albeit some authors suggest a separate category of feature addition [8]). Preventive maintenance is also sometimes bundled together with perfective maintenance.

Our goal is to use the versions of the Linux kernel to characterize these four categories of maintenance. Based on the structure of Linux releases, we expect corrective maintenance to be reflected in successive minor releases of production versions (and 4th digit releases of 2.6). Perfective maintenance, on the other hand, especially as it pertains to the addition of new features, is expected to be reflected in successive minor releases of development kernels (and 3rd digit releases of 2.6). Adaptive maintenance is also reflected in development versions, but specifically in the arch and drivers directories, as this is the locus for code that handles interaction with the environment — where we use the interpretation that this refers to the hardware environment, rather than to the users.

The identification of preventive maintenance is harder. We assume that preventive maintenance is related to code reorganization, and can therefore be identified, for example, by changes in the number of directories [2]. We will therefore look for isolated events in which many files are partitioned, removed, or moved, again mainly in development kernels.

C. Software Metrics

Our goal is to characterize the effect of maintenance on the codebase of a large project. Therefore we need to be able to quantify various size and structure attributes of the code. We use the most common metrics, which are available in practically all CASE tools [10], [17]:

1) Lines of code (LOC), including its related variants: comment lines and total lines.
2) Number of modules, as expressed by the number of directories, files, and functions.
3) McCabe's cyclomatic complexity (MCC), which is equivalent to the number of conditional branches in each function plus 1 [16], and its extended version (EMCC) where one counts the actual conditions and not only the conditional statements [18].
4) Metrics defined as part of Halstead's software science [7]. The building blocks used are the total number of operators N_1 and the number of unique operators n_1, as well as the total operands N_2 and unique operands n_2. using them, Halstead defined
   The volume $HV = (N_1 + N_2)\lg_2(n_1 + n_2)$, i.e. the total number of bits needed to write the program.
   The difficulty $HD = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$.
   The effort $HE = HV \cdot HD$: simply the product of the amount of code and the difficulty of producing it.
   In cases when the metrics are undefined (e.g. for an empty function $n_1 = n_2 = 0$) they were taken as 0. This happened in around 1% of the functions.
5) Oman's maintainability index (MI) [19], [29], which is a composite metric that attempts to fit empirical data from several software projects. Its definition is
$$MI = 171 - 5.2\ln(\overline{HV}) - 0.23\overline{MCC} \\ -16.2\ln(\overline{LOC}) + 50\sin(\sqrt{2.46\overline{pCM}})$$
   where $\overline{X}$ denotes the average of X over all modules, and pCM is the percentage of comment lines. However, following Thomas [27], we interpret pCM as a fraction (0 to 1) rather than a percentage (0 to 100) so that $\sqrt{2.46pCM}$ has the range of 0 to approximately $\frac{\pi}{2}$.
6) Files and directories handled (added/deleted/modified).
7) The rate of releasing new versions.

We acknowledge that practically all of these code metrics have been challenged on both theoretical and experimental grounds [24], [30], [25], [6]. We nevertheless use them for two reasons. First, there are no metrics that are universally accepted, so we opt for those that have been used the most and over the longest period, despite their shortcomings. Second, most objections relate to the actual values that the metrics assign to code. We are not interested so much in the values, but with how they change with time. By using a wide variety of metrics, which have all been used in empirical studies by

other researchers and are widely available in CASE tools, we hope to overcome their individual disadvantages and to be able to see the general picture.

A notable omission from the above list is common coupling, which has been used to assess the Linux code in several previous studies [22], [31], [27], [3]. However, all those studies neglected to fully follow inter-procedural pointer dereferences, and thus potentially miss many instances of coupling. As this is an extremely difficult issue, we leave its resolution to separate future work.

In plotting metric results, we use a simple visual inspection to comment on observed patterns. The alternative is to use statistical tests, as was done for example by Lawrence [12]. However, the results of such statistical tests depend on the test used, and on the precise metrics used — both of which are open to controversy. In addition we note that in many cases the overall observed behavior is erratic, sometimes including large discrete jumps. We therefore limit most of the conclusions to strong effects that are self-evident from the data.

D. Methodology

We examined *all* the Linux releases from March 1994 to August 2008. There are 810 releases in total (we only consider full releases, and ignore interim patches that were popular mainly in 1.x versions). This includes 144 production versions, 429 development versions, and 237 versions of 2.6 (up to release 2.6.25.11). This represents a significant extension of previous characterizations of Linux, such as the work of Godfrey and Tu [4] (their last versions were 2.2.14 and 2.3.39, released in January 2000, and even before that they used only a sample of releases).

We examined the *entire* source code of the Linux Kernel, including both .h and .c files. In this we follow Godfrey and Tu [4] and Thomas [27], as opposed to Schach et al. [22] who only included the .c files. Given that we are interested in code maintenance rather than in execution, we take the developer point of view and consider the full code base. Schach et al. [22] considered only the kernel subdirectory, that contains but a small fraction of the most basic code. Thomas [27] spent much effort on defining a consistent configuration that spans versions 2.0 to 2.4, thus excluding much of the code (mainly drivers). In any case, such a consistent version cannot be found spanning the full duration from 1.0 to 2.6. We ignore makefiles and configuration files, as was done by others.

In order to analyze our full dataset, we considered using a commercial CASE tool, but in the end developed a tool of our own. The main problem with the CASE tools was handling conditional compilation. Linux is littered with pre-processor directives (such as #ifdef) that cause specific compilations to include or exclude certain blocks of code. In addition, there are macros such as #define MAX(X,Y) (X>Y)?X:Y. If one is interested in the behavior of the code as it runs, all these have to be processed before the analysis starts.

In our work, however, we are interested in how the code is viewed by the maintainer. In case of conditional compilation, a maintainer must consider all the different possible compilations, not just one of them. On the other hand, a maintainer (or developer) can benefit from the cognitive simplifications that come from using macros, especially when the final expanded code is very complex. Therefore we need to analyze the code in its raw form, without pre-processing. In this we differ from Thomas [27], who used a CASE tool which requires pre-processing, and thus only files and code sections in the pre-processed configuration were examined.

The major problem with not performing pre-processing is that the resulting code is not always legal. For example, a function may have different signatures in different configuration, and this can be expressed using #ifdef and #else. With pre-processing, the compiler will only see the correct definition each time. But if we delete the pre-processing directives in order to analyze all the code, we will get two contradicting definitions of the same function, sharing the same function body. This tends to break CASE tools that measure various code metrics. We therefore developed our own tool, that could handle nearly all of the codebase. Overall, less than 1.5% of the source files were found to be so problematic that we had to remove them from the analysis.

We ran our tool on all the .c and .h files of all the versions. In order to aggregate the metrics at the kernel level, we used the same approach used in other studies (such as Thomas [27]) and as explained in the original metrics definitions. For example, LOC, MCC, and EMCC are simply summed across all files (note that files with no functions, such as some header files, will have an MCC of 0, because MCC is a function level metric). In order to compare the different versions, despite the addition of new files or functions, we will sometimes look at the average metric values of the files and functions of the kernel rather than at the aggregate values. The same is true for the function-level Halstead metrics. Oman's MI is defined at the file level. Since it has a 100-point scale we cannot aggregate the values. Instead, we will use only the average MI values across all the files in the kernel.

III. Measurements of the Linux Kernel

In this section we present the results of the code measurements; the next section will interpret these results in the light of maintenance categories. To observe the changes in metric values from one version to the next, we typically graph the results for all 810 versions, distinguishing between the arch and drivers directories and the core of the kernel. The X axis in these graphs represents the release date [28], and we make a distinction between the development versions (1.1, 1.3, 2.1, 2.3, and 2.5), the production versions (1.0, 1.2, 2.0, 2.2, and 2.4), and the 2.6 series.

A. Total and Average Size

The size of a software system may be measured in different ways. For brevity, we'll focus on lines of code (LOC), and specifically non-comment non-blank lines of code. The results

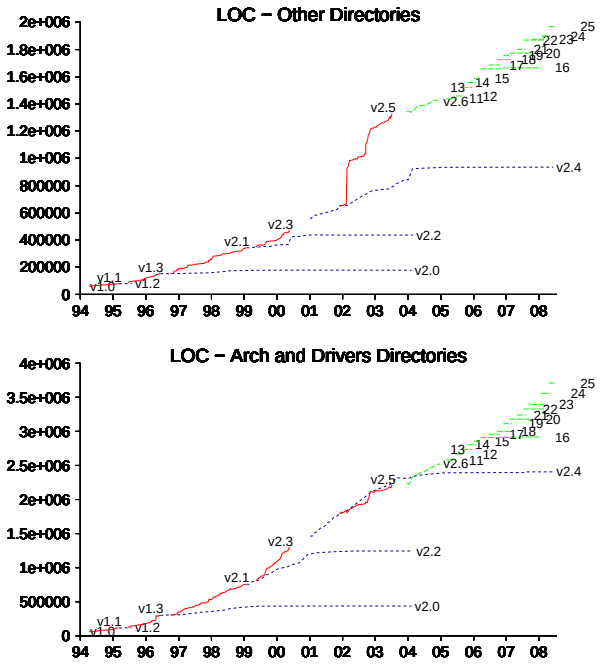

Fig. 1. *Evolution of LOC in Linux.*

are shown in Fig. 1. Very similar results are obtained for the number of files or functions, or the Halstead volume.

Obviously, Linux has grown considerably since its first release. Most of the growth occurs in the development versions, whereas the production versions are usually stable, except for some increase at the beginning. This is especially noticeable in 2.4, which continues to grow till 2.6 is released, indicating that much code was added. The 2.6 series demonstrates a combination of the development and production versions: for each minor version (third digit) the size is constant as in production versions, but it grows between them as in development versions.

An interesting observation is that the pattern of growth in the arch and drivers directories can be different from the rest of the kernel, especially in select points [4]. For example, version 2.5 exhibits two large "jumps" in the core directories, only one of which exists in arch and drivers, where it is much smaller. The first of these jumps is explained by the addition of the sound subdirectory. In previous versions there was a much smaller sound directory under drivers, but generally the sound project was developed separately. In 2.5.5 a new sound directory was created at the top level, and the include directory was updated with header files and also grew. The drivers directory lost the old sound directory, but it did not shrink because other unrelated additions were made.

The second jump is actually a combination of two effects. One is in the arch and drivers directories, where improvements and developments of the arch/um directory were made in version 2.5.35. In the consecutive version (2.5.36) a new file system support (xfs) was added, generating an increase in the core kernel number of files. Thus the two seemingly correlated jumps are actually unrelated.

It is also worthwhile to notice the smaller jumps in 2.2 (mainly 2.2.18). These increases are due to many individual changes, including improved USB support and the addition of network drivers. Note that these features were added to 2.2 despite the fact that it is a production version.

Next, we calculated average LOC at the file and the function level. The most obvious observation regarding the average size of files (Fig. 2) is that they are somewhat volatile and relatively large in the arch and drivers directories, but much smaller and more stable in the other directories. Also, within the arch and drivers directories production versions tend to have higher values. This is due to a combination of initial growth in these directories, which is larger than in contemporary development versions, and stability while the values for subsequent development versions decline. The end result is that the arch and drivers directories in the 2.0, 2.2, and 2.4 production versions have the highest average file sizes.

The LOC per function data (Fig. 3) presents a slightly different picture. The average sizes of functions are similar in all directories, but with some fluctuations and jumps. And there is a distinctive downwards trend in 2.6, indicating that the number of functions (but not files) is growing faster than the LOC. There are also a couple of interesting singularities, especially in the "core" directories. One is the significant drop in LOC per function in version 1.1. Another is the big jump in version 2.2.16. This is en example of the effect of outliers: it was caused by the addition of only 4 files with 7 functions each to the fs directory — but with total of over 54,000 LOC.

B. Complexity and Maintainability

McCabe's Cyclomatic Complexity (MCC) quantifies complexity as the number of linearly independent paths in the program's control flow graph. We also examined the EMCC (Extended MCC) and Halstead effort, but the results were qualitatively very similar to those of the MCC, so we do not discuss them. Our results (Fig. 4) show that MCC grows with time, and moreover that the pattern of growth is very similar to that of LOC. This matches previous results that noted the general correlation of MCC with LOC [24].

Next, we looked at the average MCC per function, since MCC was originally proposed as a metric for independent code units. The results (Fig. 5) indicate a pronounced decrease over time, especially in the core directories of early development versions and in the arch and drivers directories of all development versions and 2.6. This gives the impression that with time the average complexity of the functions is decreasing, and thus maybe the quality of the kernel is improving — in contrast to other results using different metrics (but also quite different methodology) [22], [28].

This conclusion remains also after a more extensive analysis of the whole distribution of MCC values of all functions, and how the distribution changes with time [9]. However, this doesn't say whether the improvement is in existing functions (which can be interpreted as preventive maintenance) or simply that new functions that are added have lower complexity.

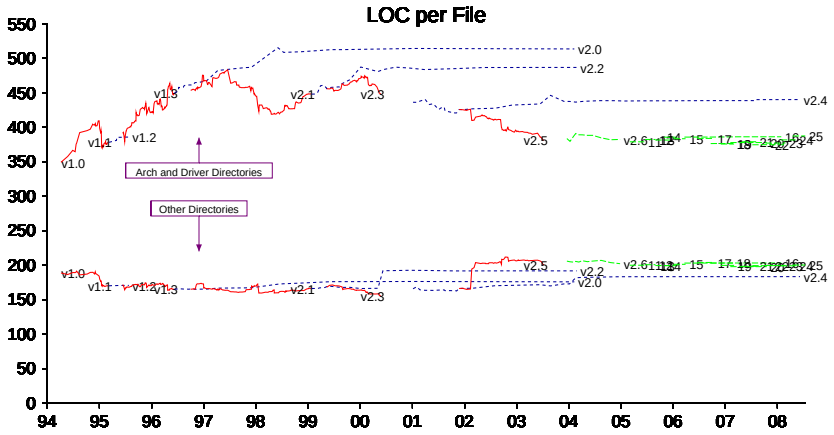

Fig. 2. *The average LOC per file.*

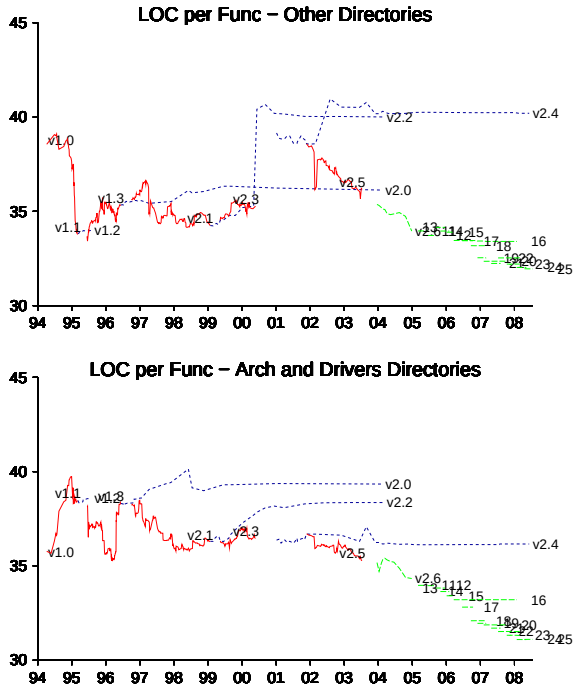

Fig. 3. *The average LOC per function.*

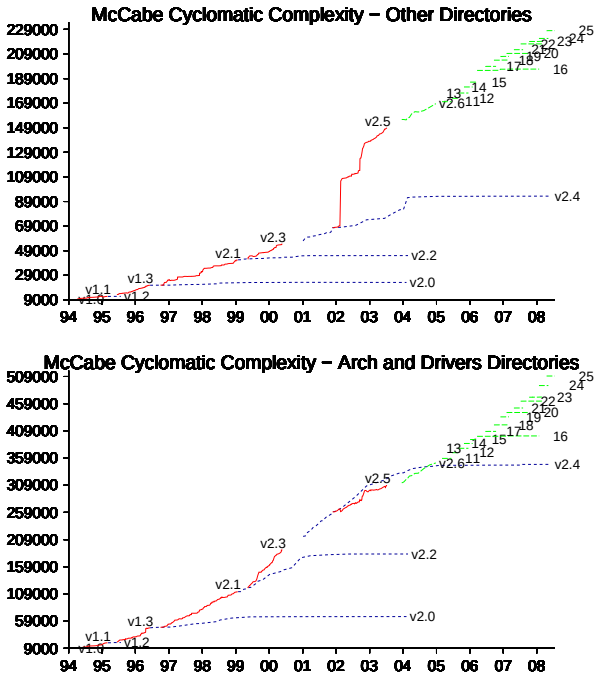

Fig. 4. *Total McCabe's cyclomatic complexity (MCC).*

We examined this directly by looking at the new files that are added each year separately from the older files (Fig. 6), focusing on development versions. This shows that new files have a decidedly lower complexity in the arch and drivers directories, and most of the time also in the core directories.

Moving to maintainability, the percentage of comment lines gives some perspective on the relation between total lines and those that actually contain code (Fig. 7). On average comments comprise around 25% (Similar to the results of Godfrey and Tu who found that the percentage of comments and blank lines is almost constant at between 28–30% [4]), with a general decreasing trend, and slightly higher values in production kernels. Dissecting this behavior we find that in the arch and drivers directories there is a big increase in comments in version 1.1, and then a general decreasing trend (to around 20%), where again the production version's values are more constant and higher. The trend in the core kernel directories is also a big increase in 1.1, but then it stays relatively flat, except for big jumps in 2.2 (upwards) and 2.5 (downwards).

Each of the previous metrics has a following of researchers who believe in it, while others criticize its deficiencies. Oman's Maintainability Index (MI) is an attempt to pool them together in a way that matches empirical data. Its value is on a scale from 25 to 125, where low values correspond to lower maintainability and high values to higher maintainability.

As MI is measured per module (or in our case, function), the data used is average LOC, MCC, and HV per function. As we saw above, these tend to decrease with time, thus

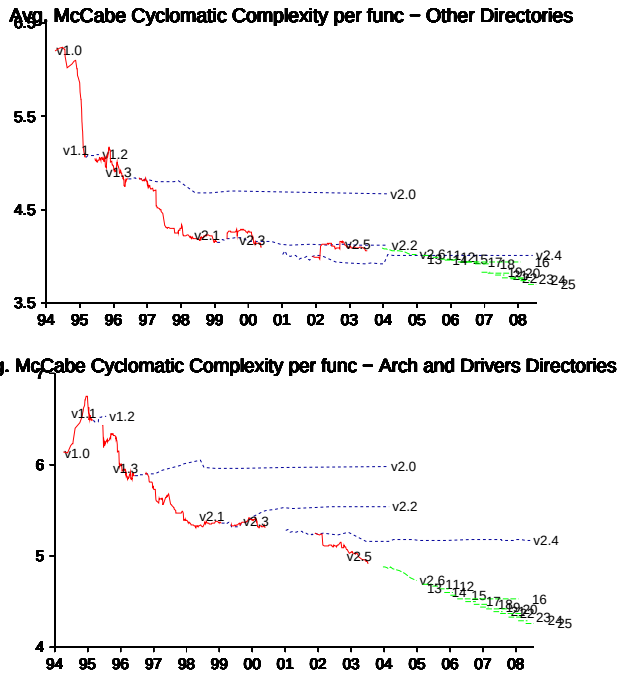

Fig. 5. *Average MCC per function.*

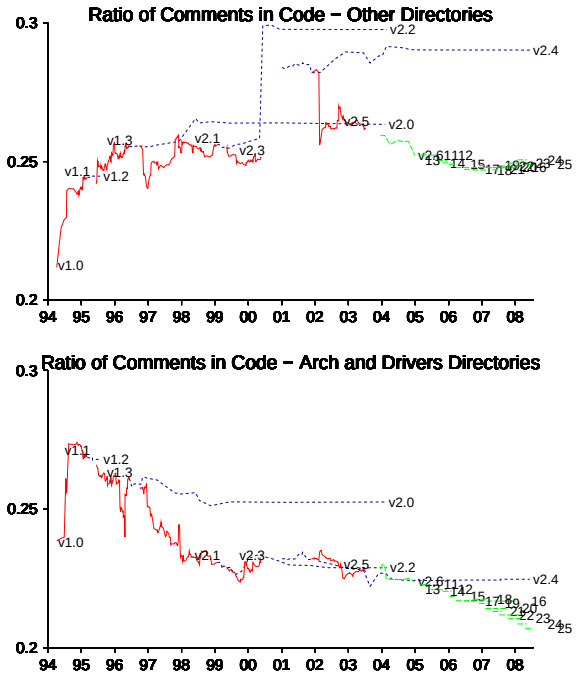

Fig. 7. *Percentage of comment lines.*

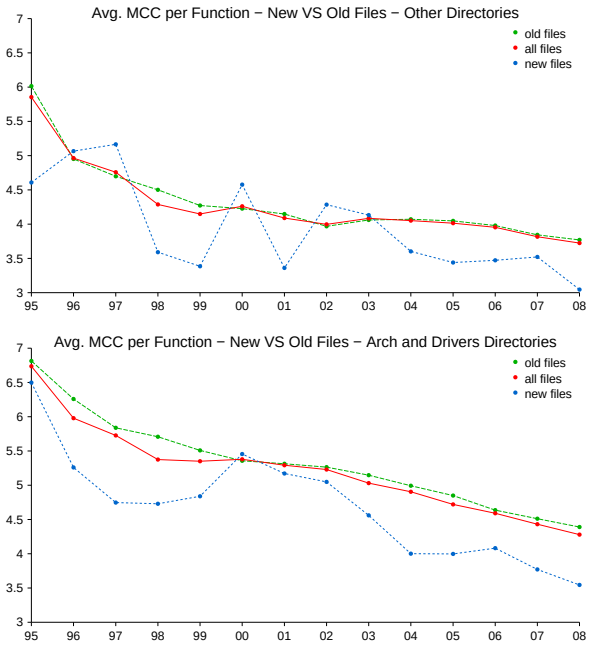

Fig. 6. *Average MCC per function in new and old files each year.*

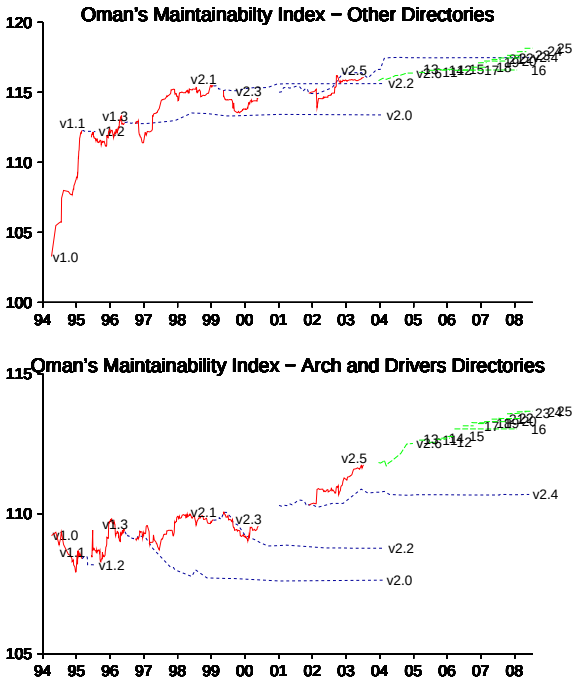

Fig. 8. *Oman's maintainability index.*

contributing to a higher MI. The percentage of comment lines, however, has a slight downwards tendency. However the change is small, so we do not expect it to have a large negative effect. Indeed the general result is a slight increasing trend of MI, as shown in Fig. 8.

Dissecting the general results according to directories, we observe the sharp initial improvement in 1.1 which is confined to the core directories. Subsequent slower improvements appear more in the arch and drivers directories. The lower values attributed to production versions are also due to these two directories. Another interesting point is that since the quality values for the core kernel directories are typically better than those of arch and drivers (i.e. less LOC, lower values for HV

and MCC, and slightly more comments), we also see that the MI for these directories is higher — meaning that the core has "better quality" than arch and drivers. However, arch and drivers are showing a larger improvement with time.

IV. Analysis of Maintenance Activities

We now use the above results to reflect on the four categories of maintenance activities: corrective, perfective, adaptive, and preventive.

A. Corrective Maintenance

As indicated previously, we analyze corrective maintenance as reflected in successive versions of production kernels, since changes in successive versions of production kernels are usually corrective, due to the structure of the releases in Linux.

As we have seen in the results, for each of the different metrics calculated — number of files, number of functions, LOC, MCC, Halstead's metrics, and Oman's maintainability index — the metric values for the production versions are essentially constant. This is seen in 1.2, 2.0, 2.2, and 2.4, and also in each of the minor versions of 2.6.

However, the metric values in production kernels do change in two cases. One is the large "jumps" seen in versions 2.2 and 2.4. These are explained by changes in functionality, where significant new functionality was propagated into a production version, but without calling this a new major version. The main examples are the improved USB support and additional drivers that were added to the 2.2.18 kernel, and the introduction of the xfs file system to the 2.4.25 kernel. Thus these jumps testify that the assumption that production versions represent only corrective maintenance is not always correct. They do not contradict the finding that corrective maintenance does not induce large changes to metric values.

The other metric change observed in production versions is that both size and complexity metrics tend to grow initially and only then become constant. This initial growth may seem to indicate that corrective maintenance — and specifically the combination of extensive testing and attempting to respond to user bug reports regarding new production versions — tends to add code and complexity to the existing structure, without a commensurate investment in restructuring and refactoring. However, it may also be just another case of creeping functionality updates, especially considering that most of the growth often occurs in the arch and drivers directories. Note also that the average size and complexity per function do not grow, indicating that the overall growth is a result of added functions and not changes to existing code. Resolving this will require a detailed analysis of the actual modifications done in the initial part of production versions.

One can thus tentatively conclude that corrective maintenance does not have a strong effect on the different kinds of metrics. If at all, there may be a mild degradation initially, but then the values of the metrics are constant. We can also state that corrective maintenance is much less frequent than the others, as the releases of production versions are much less frequent. This may have implications regarding the total effort invested in corrective maintenance [15], [23].

B. Perfective Maintenance

We analyze perfective maintenance as reflected in successive versions of development kernels, since according to the structure of the Linux versions, the main motivation for new development versions is to improve and add functionality.

Our results show that the different metrics for successive versions in development kernels usually change more than for production kernels, with strong growth when the full kernel is considered. However, this is not always the case when only the core kernel is considered. Also, average values display considerable volatility.

On one hand, it seems like less work is being done to "maintain" the code in development kernels. For example, the ratio of comments tends to be slightly lower for development versions relative to production versions. On the other hand, when comparing metrics such as average MCC per function, the values for development kernels are lower and exhibit a decreasing trend. As a consequence, the MI is higher for the development kernels, possibly indicating that the functions in development kernels are on average less complex and more maintainable than in production kernels.

The trend of improved average metrics over time in development kernels has two possible interpretations: either the code in development versions is indeed being improved, or else many small files and functions are simply being added at a higher rate than the overall complexity grows, so the averages shift in the desirable direction. Our results indicate that both effects indeed occur. For example, by considering new functions separately from older ones, we showed that the new functions added each year had better metric values. Likewise, we have seen specific examples of improvements to code complexity and structure of functions with high MCC values [9]. Moreover, these improvements were limited to the development versions. Thus we indeed have preliminary evidence for activity that improves code structure in development versions. This is discussed further below when we deal with preventive maintenance.

Another aspect of perfective maintenance is the quest to improve maintainability. Such activity may be expected to lead to an improvement in code quality metrics. This consideration indicates that version 1.1 is special, as there seems to be especially significant improvement in the code in that version: the fraction of comments grew, and all the complexity metrics (MCC, Halstead) decreased considerably. One may therefore conjecture that this version saw much perfective maintenance, being the first version after $2\frac{1}{2}$ years of development from the initial announcement in August 1991 to the first release in March 1994. Verifying this conjecture will require detailed scrutiny of the code to assess the reasons for the improvement in the metric values.

C. Adaptive Maintenance

We analyze adaptive maintenance as reflected in the arch and drivers directories (especially in the development kernels), since they best reflect the adaptation to changes in the envi-

ronment — the addition of new architectures and new devices that need to be supported.

As we have seen above, the arch and drivers directories usually have the same trends as the rest of the kernel, although many times with higher magnitude (for example, the LOC in the arch and drivers directories is higher and grows faster than in other directories — possibly due to significant code cloning [5]). However, there are differences. For example, the changes in LOC and number of files in 2.5 are more noticeable in the core kernel, whereas in 2.2 they are more noticeable in the arch and drivers directories.

When looking at average LOC and MCC (and other metrics) per function, the observed trends are quite different in the different directories: in the arch and drivers directories they exhibit a steady decline throughout the development versions, whereas in other directories the values are more volatile or constant. This seems to indicate that adaptive maintenance, as reflected in the arch and drivers directories, leads to an improvement in code quality.

An especially interesting observation is the apparent interaction between adaptive and corrective maintenance. In practically all the average metrics, the production versions have consistently higher values than contemporary development versions when only the arch and drivers directories are considered. This leads to somewhat lower MI values for the production versions, as seen in Fig. 8.

D. Preventive maintenance

Preventive maintenance is conjectured to be related to code churn [6], that is the partitioning, moving, or deletion of code. In particular, we can look for isolated events where a large churn is observed [2].

We focused on changes between successive development versions (including version 2.4, since it seems to have served, at least in the first year, for development; see below). Changes were quantified by looking at the number of files which were changed (i.e. added, deleted, grown, or shrunk; Fig. 9). Similar results were obtained for directories. The "deleted" group are files that were removed, and "added" is the difference between the versions plus the number deleted (thus counting all additions to the second version). "Grown" and "shrunk" refer to the sizes of the files. The comparison for each is continuous, i.e. we compared minor versions within the same major version, and also the first release of a major version and the previous release on which it is based. Thus version 1.3.0 was compared to version 1.2.10, version 2.1.0 to 2.0.21, version 2.3.0 to 2.2.8, and version 2.5.0 to 2.4.15. Version 2.4.0 was compared with the last 2.3 version (2.3.99-pre9), as it emerged from that version. The last comparison in each graph is between the last version in 2.5 (2.5.75) and the first in 2.6 (2.6.0). The spaces between the bars are times with no development versions.

The results indicate that the number of files changed grows with time. But if we normalize this by the number of files in the system, we find that the fraction of files changed appears to be constant or maybe even slightly decreasing. Thus the rate

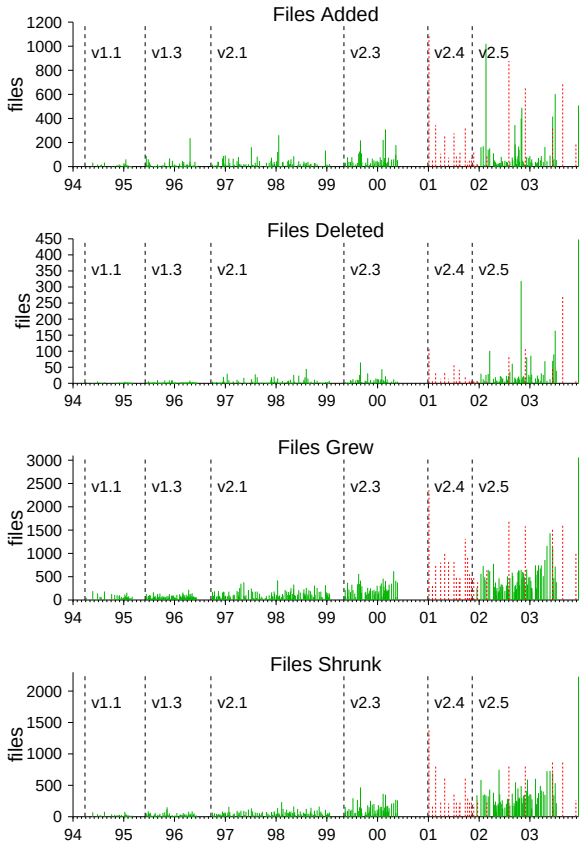

Fig. 9. *Files added, deleted, grown, or shrunk among development versions.*

of change is pretty constant between consecutive development versions. These results are slightly different from those found by Lehman [14], who claimed that the percentage of added files tends to decline, and that the percentage of changed files tends to grow.

The graphs include only a few discrete peaks of activity. For example, in version 2.5 we see a large number of files deleted around November 2002. This is the result of replacing the configuration system in version 2.5.45: the old configuration files Config.help and Config.in were deleted in many directories and were replaced by a single Kconfig file. We also see a relatively large amount of directories added and deleted around March of 2002. This is due to changes in the structure in the arch and drivers directories at that time and also the addition of the sound subdirectory to the kernel. Except for these events, however, preventive maintenance as reflected by code churn in Linux seems to be continuous rather than concentrated in isolated events, as opposed to the finding of Capiluppi for the ARLA system [2]. However, this may also be interpreted as resulting from the fact that Linux is in effect an agglomeration of numerous subsystems: it may be that preventive maintenance is applied to subsystems in discrete events, but this is masked by the rest of the system which is not subject to such activity at the same time.

V. Threats to Validity

Our analysis of the four categories of maintenance activities is based on our dissection of the versions and directories of Linux. This is justified by the fact that the different branches and directories have well-defined purposes, which match our needs. In addition, the different branches and directories are indeed obviously different from each other, as reflected by several of our measurements.

However, the question remains whether practice completely complies with the prescriptions. Regrettably, it may be suspected that in some cases, especially in the initial period of production versions, the roles are "mixed". The most extreme example of such mixing occurs at the beginning of 2.4. The last version of 2.3 was released on 24 May 2000. The first version of 2.4 was only released on 5 January 2001, and the first version of 2.5 only on 23 November 2001. Thus there is a gap of some *18 months* between successive development versions. However, it seems that the initial part of 2.4 served for development (or at least reflected development activity that was being done without officially being released in a development version), as all our graphs indicate development-like growth and that 2.5 branched out of 2.4. The remaining gap between 2.3 and 2.4 may have been filled at least partially by 2.2. Specifically, 2.2 exhibits strong growth in this period, especially in the arch and drivers directories, which matches the difference in size between the end of 2.3 and the beginning of 2.4.

In addition to large-scale events like those listed above, there have been specific instances of "feature creep" from development to production versions. The main reason for this appears to be the desire to reduce the delay until new functionality becomes available in production releases — the same force that later led to the 2.6 release scheme. An additional reason is the contribution of reasonably stable code from third parties [4], for example the incorporation of the xfs file system in 2.4.25 (in parallel to its addition to 2.5.36). Luckily, such events are easy to identify as jumps in the graphs, and they do not have a significant effect on our conclusions.

A related question is whether perfective and corrective maintenance are completely separable activities. In this we do not mean that there are cases when the two activities are interleaved, e.g. when a commit includes both the correction of a bug and an addition of a feature. Rather, we refer to the situation surrounding a release. Fig. 10 shows the the times at which minor releases are made within each major version. The steeper the line, the higher the release rate [28]. Development versions exhibit a steady high rate, whereas production versions tend to start with a similar high rate and then taper off when the next development version is started. The question is whether the initial phase may serve as part of the development path, or maybe this is just continued activity to stabilizing the code, and should therefore indeed be considered to be corrective in nature [28].

Note that despite the possible debate regarding the nature of production versions, this has little impact on our conclusions. First, the roles of other parts of the code appear to be relatively clear. And even regarding the production versions, most of our conclusions regarding corrective maintenance refer to the stable period beyond the initial growth.

The main external threat to validity is that our work is limited to Linux — both due to its unique evolution, and due to our interpretation of the roles of different parts of the code. Additional research is required to determine whether our observations generalizes to other systems as well.

VI. Conclusions and Future Work

The Linux kernel is one of the most successful open-source software projects in the world. It has been maintained continuously over the last 14 years in order to satisfy the needs of its users. We have presented a detailed characterization of this process, including over 800 releases which represent new developments, major production releases, and minor updates. Many interesting phenomena are only seen at this fine resolution, and would be lost if using the traditional approach of studying only major production releases. Our results are of course specific to Linux, but some observations may generalize to other software systems as well.

We exploited the branches and structure of Linux to dissect different categories of maintenance activities. Looking at development versions, we found, as expected, that Linux is growing strongly. However, the average LOC per function is decreasing, so the number of functions is growing faster than the LOC. A similar pattern is seen for metrics such as MCC and the Halstead metrics. This appears to be at least partly due also to real improvements and preventive maintenance. As a result, Oman's maintainability index was generally increasing. A special case of this is version 1.1, which had improved significantly over time, perhaps since there was much effort on preventive and perfective maintenance.

In the production versions the various metrics usually stabilized at a constant level after a while. They typically tend to suffer some initial degradation before stabilizing, raising the question of whether there are distinct phases of corrective activity: first intensive corrections needed to stabilize the code after a major new release, and then more relaxed correction of latent bugs that show up. We also found specific cases where production versions include perfective maintenance and not only corrective maintenance, contradicting our presumption. Such cases where the different versions did not follow the general trend were analyzed and explained in detail. These were usually instances of adding a large module that was developed elsewhere into the kernel.

As for the distinction between the arch and drivers directories and the core kernel, we found that arch and drivers usually displayed more persistent trends, but still had "worse" values (e.g. higher LOC and MCC, and lower MI). Thus it seems that these directories originally suffered from relatively poor code, but that adaptive maintenance is improving them with time. It remains to be seen whether they will reach or even surpass the quality of code in the core directories.

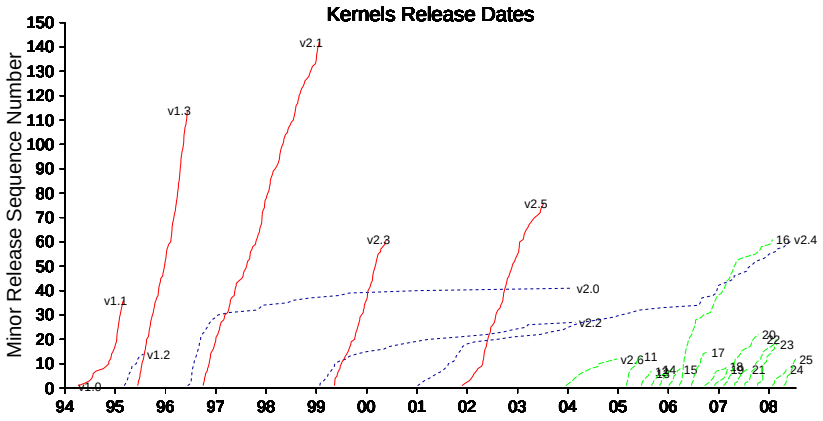

Fig. 10. *Minor number as a function of time, such that the slope of the line indicates the release rate.*

Some of our results indicate that a globally observed behavior may be attributed to a very local change in the codebase. In particular, it may be that various subsystems, not only the arch and drivers directories, exhibit distinct behaviors [4]. This suggests that in future work we should look more closely at distinct subsystems. This is interesting in the context of understanding the evolution of Linux [4], but is not expected to shed light on the comparison between maintenance activities.

Another direction for future work is to try and glean the amount of work invested in different maintenance activities. For example, a survey of maintenance managers yielded the result that 17.4 percent of maintenance is corrective in nature [15], but a study that analyzed changes to source code obtained a figure more than three times larger [23]. It would be interesting to try to figure out proportions for Linux by quantifying work done on different versions.

Acknowledgments

This work was supported by the Dr. Edgar Levin Endowment Fund.

References

[1] *ISO/IEC 14764:2006 software engineering – software life cycle processes – maintenance*. www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39064, Aug 2006.

[2] A. Capiluppi, M. Morisio, and J. F. Ramil, "*Structural evolution of an open source system: A case study*". In 12th *IEEE Intl. Workshop Program Comprehension*, pp. 172–182, Jun 2004.

[3] D. G. Feitelson, T. O. S. Adeshiyan, D. Balasubramanian, Y. Etsion, G. Madl, E. P. Osses, S. Singh, K. Suwanmongkol, M. Xia, and S. R. Schach, "*Fine-grain analysis of common coupling and its application to a Linux case study*". *J. Syst. & Softw.* **80(8)**, pp. 1239–1255, Aug 2007.

[4] M. W. Godfrey and Q. Tu, "*Evolution in open source software: A case study*". In 16th *Intl. Conf. Softw. Maintenance*, pp. 131–142, Oct 2000.

[5] M. W. Godfrey, D. Svetinovic, and Q. Tu, "*Evolution, growth, and cloning in Linux: A case study*". In *CASCON workshop on Detecting Duplicated and Near Duplicated Structures in Large Software Systems: Methods and Applications*, 2000.

[6] G. A. Hall and J. C. Munson, "*Software evolution: Code delta and code churn*". *J. Syst. & Softw.* **54(2)**, pp. 111–118, Oct 2000.

[7] M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.

[8] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "*Automatic classification of large changes into maintenance categories*". In 17th *IEEE Intl. Conf. Program Comprehension*, pp. 30–39, May 2009.

[9] 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.

[10] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2nd ed., 2004.

[11] C. F. Kemerer and S. Slaughter, "*An empirical approach to studying software evolution*". *IEEE Trans. Softw. Eng.* **25(4)**, pp. 493–509, Jul/Aug 1999.

[12] M. J. Lawrence, "*An examination of evolution dynamics*". In 6th *Intl. Conf. Softw. Eng.*, pp. 188–196, Sep 1982.

[13] M. M. Lehman, "*Programs, life cycles, and laws of software evolution*". *Proc. IEEE* **68(9)**, pp. 1060–1076, Sep 1980.

[14] M. M. Lehman, D. E. Perry, and J. F. Ramil, "*Implications of evolution metrics on software maintenance*". In 14th *Intl. Conf. Softw. Maintenance*, pp. 208–217, Nov 1998.

[15] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "*Characteristics of application software maintenance*". *Comm. ACM* **21(6)**, pp. 466–471, Jun 1978.

[16] T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **2(4)**, pp. 308–320, Dec 1976.

[17] E. Mills, *Software Metrics*. Tech. Rep. Curriculum Module SEI-CM-12-1.1, Software Engineering Institute, December 1988.

[18] G. J. Myers, "*An extension to the cyclomatic measure of program complexity*". *SIGPLAN Notices* **12(10)**, pp. 61–64, Oct 1977.

[19] P. Oman and J. Hagemeister, "*Construction and testing of polynomials predicting software maintainability*". *J. Syst. & Softw.* **24(3)**, pp. 251–266, Mar 1994.

[20] D. L. Parnas, "*Software aging*". In 16th *Intl. Conf. Softw. Eng.*, pp. 279–287, May 1994.

[21] D. A. Rusling, "*The Linux kernel*". URL tldp.org/LDP/tlk/.

[22] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt, "*Maintainability of the Linux kernel*". *IEE Proc.-Softw.* **149(2)**, pp. 18–23, 2002.

[23] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt, "*Determining the distribution of maintenance categories: Survey versus measurement*". *Empirical Softw. Eng.* **8(4)**, pp. 351–365, Dec 2003.

[24] M. Shepperd, "*A critique of cyclomatic complexity as a software metric*". *Software Engineering J.* **3(2)**, pp. 30–36, Mar 1988.

[25] M. Shepperd and D. C. Ince, "*A critique of three metrics*". *J. Syst. & Softw.* **26(3)**, pp. 197–210, Sep 1994.

[26] E. B. Swanson, "*The dimensions of maintenance*". In 2nd *Intl. Conf. Softw. Eng.*, pp. 492–497, Oct 1976.

[27] L. Thomas, *An Analysis of Software Quality and Maintainability Metrics with an Application to a Longitudinal Study of the Linux Kernel*. Ph.D. thesis, Vanderbilt University, 2008.

[28] L. G. Thomas, S. R. Schach, G. Z. Heller, and J. Offutt, "*Impact of release intervals on empirical research into software evolution, with applications to the maintainability of Linux*". *IET Softw.* **3(1)**, pp. 58–66, Feb 2008.

[29] E. VanDoren, *Maintainability Index Technique for Measuring Program Maintainability*. Tech. rep., Software Engineering Institute, Mar 2002.

[30] E. J. Weyuker, "*Evaluating software complexity measures*". *IEEE Trans. Softw. Eng.* **14(9)**, pp. 1357–1365, Sep 1988.

[31] L. Yu, S. R. Schach, K. Chen, and J. Offutt, "*Categorization of common coupling and its application to the maintainability of the Linux kernel*". *IEEE Trans. Softw. Eng.* **30(10)**, pp. 694–706, Oct 2004.