Maintainability of the Linux kernel

S.R.Schach, B.Jin, D.R.Wright, G.Z.Heller and A.J.Offutt

Abstract: The authors have examined 365 versions of Linux. For every version, they counted the number of instances of common (global) coupling between each of the 17 kernel modules and all the other modules in that version of Linux. They found that the number of instances of common coupling grows exponentially with the version number. This result is significant at the 99.99% level, and no additional variables are needed to explain this increase. On the other hand, the number of lines of code in each kernel module grows only linearly with the version number. They conclude that, unless Linux is restructured with a bare minimum of common coupling, the dependencies induced by common coupling will, at some future date, make Linux exceedingly hard to maintain without inducing regression faults.

1 Introduction

Numerous articles in newspapers and popular magazines point out the many strengths of Linux, the open-source operating system [1]. Linux is also increasingly featured on television news programs. Typically, such items include an interview with a former user of Microsoft Windows who proudly asserts that Linux fails far less frequently on his or her PC than Windows did. Occasionally, a magazine article might mention that it is important for one to install a version of Linux that is appropriate for one's PC or that it is helpful to know a Linux guru, but most media coverage is largely uncritical of Linux.

A cynic might claim that these articles are just a manifestation of a worldwide campaign of 'Microsoft bashing'. A statistician would surely point out that the articles are anecdotal in nature and can hardly be considered to constitute scientific evidence. Nevertheless, the sheer volume of material praising Linux in the popular press and on television is difficult to ignore.

Turning now to software experts, their adulation of Linux is somewhat more muted. For example, in an interview in the May 1999 issue of *Computer*, Ken Thompson (co-creator of UNIX) stated: "I don't think [Linux] will be very successful in the long run. I've looked at the source and there are pieces that are good and pieces that are not. A whole bunch of random people have contributed to this source, and the quality varies drastically. My experience and some of my friends' experience is that Linux is quite unreliable. Microsoft is really unreliable but Linux is worse. In a non-PC environment, it just won't hold up. If you're using it on a single box, that's one thing. But if you want to use Linux in firewalls, gateways, embedded systems and so on, it has a long way to go" [2].

A key phrase in Thompson's remarks is "I've looked into the source". A critical difference between Linux and Windows is that Linux is open-source software; anyone can study the source code and comment on (say) its quality.

It has been claimed that open-source software is superior to proprietary software. One reason given for this assertion is that open-source software can be improved by anyone who has a copy of the program. A second reason, frequently put forward, is the fact that the name of the author of a module is usually incorporated into the source code; public knowledge of who wrote the software is viewed as an inducement for writing quality code. Finally, in the case of products such as Linux, yet another reason given is that most of the code has been written by volunteers working in their own time, as opposed to employees battling against management-imposed deadlines. On the other hand, Thompson's statements that "there are pieces [of Linux] that are good and pieces that are not" and "the quality varies drastically" cannot be disregarded.

Notwithstanding Ken Thompson's stature within the software engineering community, in a certain sense his opinion of the quality of Linux is as anecdotal as the views expressed by Linux users in press interviews. After all, Thompson apparently did not use a metric (such as number of faults detected) to measure quality. Furthermore, it is not clear whether his opinion is based on an exhaustive study of all of Linux, or on a sample of the code.

This paper presents results from an examination of available subversions of versions 1.0 through 2.3 of Linux, a total of 391 subversions. Table 1 summarises this data set. In Table 1 and throughout this paper, the term 'module' refers to a file containing executable C code (that is, a file with the suffix `.c`, as opposed to, say, a header file with suffix `.h`).

We have examined one aspect of the maintainability of the Linux kernel. Specifically, we have measured the 'common coupling' in successive versions of the Linux kernel, and observed that the common coupling increases exponentially. We conclude that if this trend continues, the maintainability of Linux will degrade in the future.

IEE Proceedings online no. 20020198

DOI: 10.1049/ip-sen:20020198

Paper first received 14th August 2001 and in revised form 15th January 2002

S.R.Schach, B.Jin and D.R.Wright are with the Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN, USA

G.Z.Heller is with the Department of Statistics, Macquarie University, Sydney, Australia

A.J.Offutt is with the Department of Information and Software Engineering, George Mason University, Fairfax, VA, USA

Table 1: Summary of Linux versions and subversions

Version number	Number of subversions	LOC (modules)	Number of modules	Total number of files
1.0	1	141 255	282	572
1.1	36	141 068	282	561
1.2	14	234 704	400	909
1.3	100	258 621	431	991
2.0	40	563 104	779	2018
2.1	130	580 698	785	2059
2.2	18	1 310 807	1891	4599
2.3	52	1 385 026	1946	4721
Total	391			

2 Dependencies

The 'coupling' between two units of a software product is a measure of the degree of interaction between those units [3] and, hence, of the dependency between the units. In their 1974 paper, Stevens, Myers and Constantine outlined six levels of coupling. These were presented as an ordered list by Page-Jones [4], who gave three principal reasons why reducing the number of instances of coupling between modules is desirable: first, fewer interconnections between modules reduce the chance that a fault in one module will cause a failure in other modules; secondly, fewer interconnections between modules reduce the chance that changes in one module cause problems in other modules, which enhances reusability and thirdly, fewer interconnections between modules reduce programmer time in understanding the details of other modules. Various modifications and extensions to these levels of coupling have been proposed over the past 25 years [4–6]. Although all types of coupling are sometimes useful in a design, it has been demonstrated that some types have greater potential for introducing faults into software [7–10]. Because some types of coupling are more likely to lead to faults than others, it is widely accepted that some coupling types should be limited in use.

In the 11-level categorisation of [5], the two lowest levels of coupling are call coupling and scalar data coupling. There is 'call coupling' between modules `P` and `Q` if `P` calls `Q` or `Q` calls `P`, but there are no parameters, common variable references or common references to external media between `P` and `Q`. There is 'scalar data coupling' if a scalar variable in `P` is passed as an actual parameter to `Q` and it is used for computation purposes ('C-use'), but not for control purposes ('P-use') or indirect purposes ('I-use'). This paper considers the classical coupling category 'common coupling', which corresponds to level 10, 'global coupling', in the categorisation of Offutt *et al.* [5]. Modules `P` and `Q` are global coupled if `P` and `Q` share references to the same global variable.

If there were no coupling at all in a software product, then that product would consist of one large module, so some amount of coupling clearly is needed. That is, coupling is a necessary consequence of modularisation. However, where there is coupling between two modules, there is some degree of dependence between those modules. The resulting degree of dependence between two modules may be high 'strong coupling' or low 'weak coupling'. A well designed software product makes considerable use of weak coupling and avoids, as far as possible, strong coupling. For example, a well designed product utilises coupling categories such as call coupling and scalar data coupling, and eschews common coupling as much as is feasible [3, 11].

It has been shown [12] that coupling is related to fault proneness. Coupling has not yet been explicitly shown to be related to maintainability. On the other hand, we do not yet have a precise definition of maintainability, and therefore there are no generally accepted metrics for maintainability. Nevertheless, if a module is fault-prone then it will have to undergo repeated maintenance, and these frequent changes are likely to compromise its maintainability. Furthermore, these frequent changes will not always be restricted to the fault-prone module itself; it is not uncommon to have to modify more than one module to fix a single fault. Thus, the fault proneness of one module can adversely affect the maintainability of a number of other modules. In other words, it is easy to believe that strong coupling can have a deleterious effect on maintainability.

As previously mentioned, in this paper we consider common coupling. There are three reasons for this. Firstly, it was shown in a case study on the maintainability of multiversion real-time software that the overwhelming preponderance of strong coupling introduced during the maintenance phase was common coupling [13]. Secondly, there is considerable controversy regarding what precisely constitutes weak or strong coupling, let alone which categorisation of coupling should be followed. However, all categorisations we have seen include a form of coupling that corresponds to classical common coupling, and there seems to be unanimity that common coupling is undesirable.

The third reason why we concentrated on common coupling is that common coupling possesses the unfortunate property that the number of instances of common coupling between module `M` and the other modules can change drastically, even if module `M` itself never changes; this is termed 'clandestine common coupling' [14]. For example, if modules `M` and `N` both reference global variable `gv`, then there is one instance of common coupling between module `M` and the other modules. But if 10 new modules are written, all of which reference global variable `gv`, then the number of instances of common coupling between module `M` and the other modules increases to 11, even although module `M` itself is unchanged. Bearing in mind that the size of Linux has increased nearly 1000% since version 1.0 (see Table 1), we suspected that common coupling between a module in the kernel and the rest of the modules might increase dramatically, even although the kernel module itself did not change hugely.

There were two reasons why we decided to concentrate our efforts on the Linux kernel. First, there are only 17 kernel modules and 6506 versions of those modules; in contrast, the current version of Linux has nearly 2000 modules, and there are up to 390 previous versions of each of those modules. In other words, the research project was manageable because we restricted our efforts to analysing 'only' 6506 modules. Secondly, in the case study on repeated maintenance we previously referenced [13], the major discriminating factor was differences in individual programmer abilities. In the case of Linux, the original versions of all the kernel modules were written by Linus Torvalds, and he has either maintained them himself or in conjunction with one or two other programmers. There is therefore no need to correct for individual programmer skills.

3 Successive versions

We measured the change in the number of lines of code and in the common coupling between successive versions of Linux kernel modules. The term 'successive versions' needed to be clearly defined, because some software products undergo parallel development, that is, the baseline version of the software is extended in more than one direction at the same time. Sometimes, one or more of these branches later coalesce, whereas other branches are terminated.

In the case of Linux, an additional complication is that, in some sense, there are two sets of versions of Linux kernel modules. Kernel modules that have a version number whose second, or minor, digit is even (for example, the '2' is even in version 1.2.9) are considered stable, and those whose minor number is odd (for example, version 2.1.132) are considered developmental. When the developmental version appears to be mature, it becomes part of the stable tree with the minor number incremented; for example, with only a few small changes, version 2.1.132 became version 2.2.0.

In order to handle the issue of parallel development, we decided to discard all versions with a date stamp later than a version with a higher version number. The remaining versions with successive version numbers were then defined to be 'successive versions'.

After we had downloaded all 391 versions of Linux available on the Web [1], we discovered that versions 2.0.30 through 2.0.39 had date stamps later than the date stamp on version 2.1.0, so we discarded those 10 versions. Similarly, versions 2.2.2 through 2.2.17 had date stamps later than the date stamp on version 2.3, so we ignored those 16 versions, also. That left 365 versions of Linux to investigate.

We comment on the implications of our definition of successive versions in Section 5.2.

4 Counting instances of common coupling

As stated in Section 2, modules `P` and `Q` are common (global) coupled if `P` and `Q` share references to the same global variable. We downloaded all the modules of each version of Linux. For each of the 17 modules that constitute the kernel, we manually determined which variables are global. We then determined in how many modules each global variable in a kernel module is referenced. The counting was done at the module level, so multiple references to the same common variable within a given module were ignored. We also ignored the common coupling of constants.

We then determined whether the code had been modified from the previous version and, if so, we noted the number of lines of code in that new version of that kernel module and when it was released.

Data for the 365 versions of Linux we investigated are shown in Table 2. A blank in the lines of code (LOC) or date column denotes that the code has not changed between successive versions. Thus, for example, version 2.1.104 of kernel module `Panic.c` was released on the 21st May 1998. That version had 79 lines of code, and there were 914 instances of common coupling between module `Panic.c` and all the other modules in version 2.1.104 of Linux. The number of instances of common coupling steadily increased to 946 in version 2.1.109, even although the code for `Panic.c` did not change at all, an example of clandestine common coupling [14].

Finally, for simplicity in the statistical analysis, we renumbered the versions as consecutive integers between 1 and 365. Thus, version 2.1.104 above became version number 196, as shown in Table 2, and similarly for the other versions.

5 Results

We present models for the relationship between LOC and version number; and between instances of common coupling and version number. A fundamental assumption of normal regression models is the independence of observations; our observations are by their nature sequential and hence have a temporal dependency. The appropriate statistical tool is therefore a 'growth curve' [15]. A separate growth curve is needed for each module because the changes in LOC and in instances of common coupling are module-specific. We found, however, that normal regression models produce very similar results to the growth curves, and have the advantage that all modules can be accommodated in a single model. For ease of presentation, therefore, we present in this paper the results of normal regression models [16]

5.1 Lines of code

We first considered the change in the LOC through the versions. A linear regression of the LOC versus version number was fitted, allowing different intercepts and slopes for each of the 17 different modules. Version number, module and a version number–module interaction were all significant ($p < 0.0001$), as shown in the analysis of variance (ANOVA) of Table 3.

The 'degrees of freedom' column gives the number of parameters to be estimated for each effect in the model. The effect 'version number × module' consists of the version number–module interaction terms. These terms

Table 2: Data for six successive versions of three kernel modules

Version number	Panic.c			Module.c			Ksyms.c		
	CC	LOC	date	CC	LOC	date	CC	LOC	date
2.1.104 (or 196)	914	79	21/05/98	963	1018	21/05/98	359	397	04/06/98
2.1.105 (or 197)	921			974	1019	07/06/98	360		
2.1.106 (or 198)	933			989			368	398	13/06/98
2.1.107 (or 199)	935			992			369		
2.1.108 (or 200)	942			992			369		
2.1.109 (or 201)	946			999			373		

Table 3: Analysis of variance for lines of code

Effect	Degrees of freedom	F	p
Version number	1	4517.3	< 0.0001
Module	16	901.9	< 0.0001
Version number × module	16	511.2	< 0.0001
$R^2 = 0.951$			

allow for differing gradients of the LOC–version number regression line for each module. The parameters for the interaction terms are the differences between the gradient of one of the modules (we arbitrarily chose `Printk.c`) and each of the other modules. Consequently, the number of parameters to be estimated is 16, the number given in the 'degrees of freedom' column. The effect 'version number' has one parameter, the gradient of the `Printk.c` module LOC–version number regression line, and the effect 'module' has 16 parameters, each of these being the difference between the intercept of the LOC–version number regression line for each module, and the intercept for the `Printk.c` module regression line. The F statistics are the ratios of the variation explained by each effect, to the residual variation or noise. If this statistic is sufficiently large, we conclude that the effect is statistically significant. We judge the size of the F ratio for each effect by comparing it against the distribution we would expect it to have if the effect were in fact not present, namely, the F distribution. If the F ratio falls in the upper tail of the F distribution, we conclude that the effect is present, or statistically significant.

The model explains 95.1% of the variation in the LOC. That is, the two variables and their interaction account for 95.1% of the observed behaviour of the LOC. This result is deduced from the value of R^2 in Table 3, which is the ratio (variation explained by model)/(total variation), that is, the

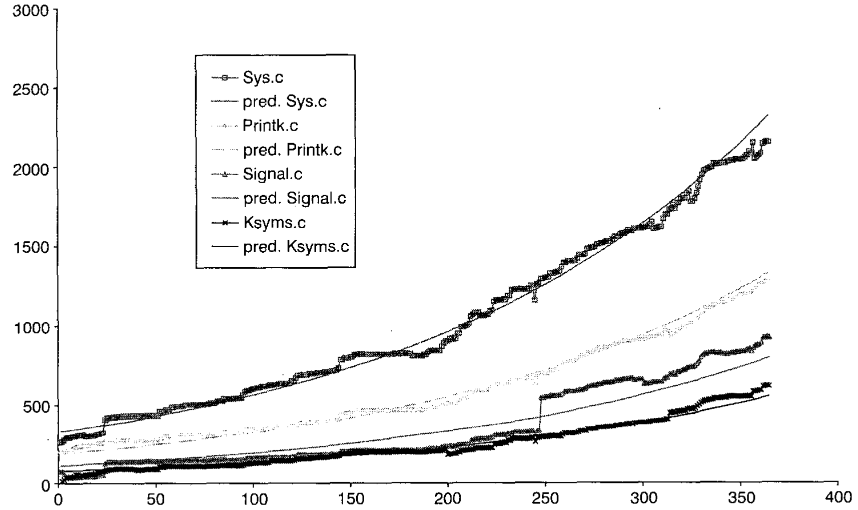

Fig. 1 *Graphs of measured and predicted common coupling versus version number. The measured common coupling is represented by discrete points, the predicted common coupling by a line*

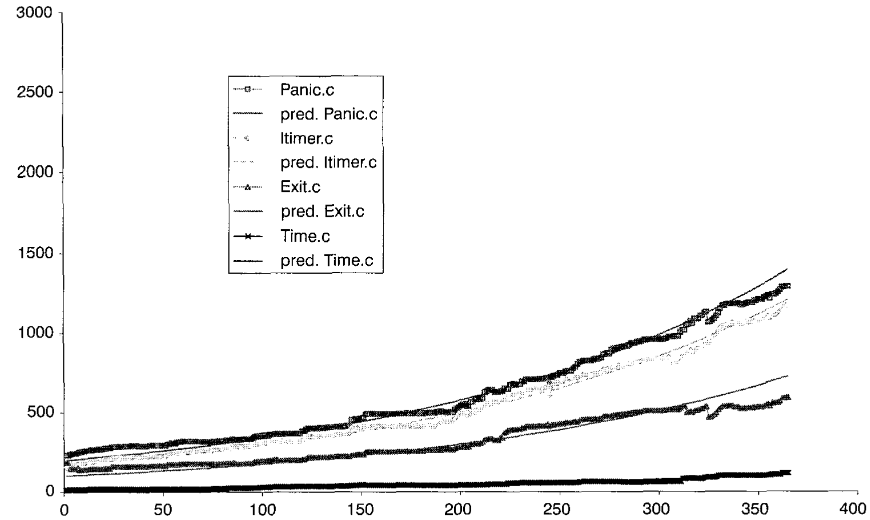

Fig. 2 *Graphs of measured and predicted common coupling versus version number. The measured common coupling is represented by discrete points, the predicted common coupling by a line*

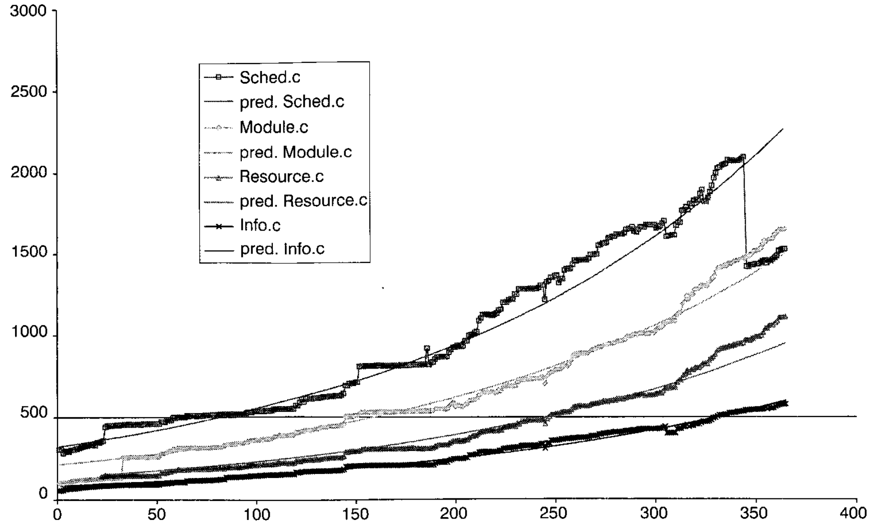

Fig. 3 *Graphs of measured and predicted common coupling versus version number. The measured common coupling is represented by discrete points, the predicted common coupling by a line*

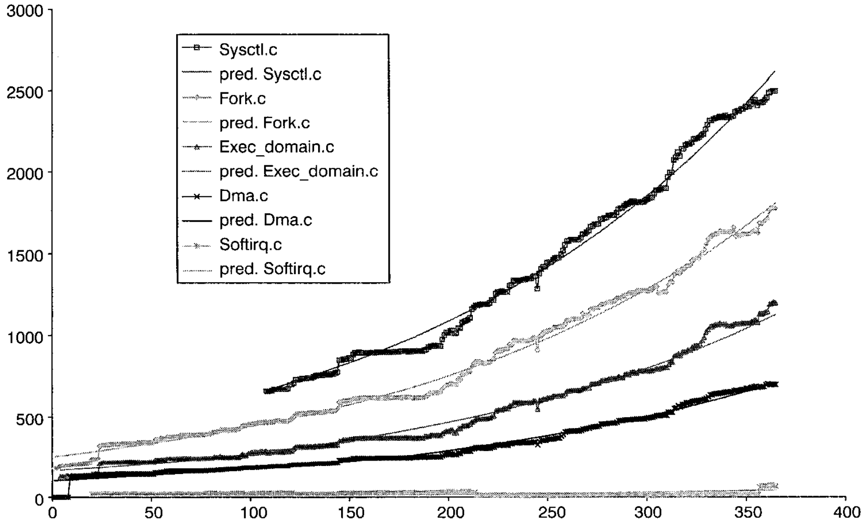

Fig. 4 *Graphs of measured and predicted common coupling versus version number. The measured common coupling is represented by discrete points, the predicted common coupling by a line*

proportion of total variation in the LOC explained by the model.

5.2 Common coupling

Figs. 1–4 show how common coupling varies with version number. Four of the Linux kernel modules appear in the first, second and third graphs and five in the fourth. All four graphs show both the measured value of the common coupling and the values predicted by our model. Figs. 1–4 reveal an extremely clear exponential trend, which is again module-specific. In passing, we note that the smoothness of the graphs of Figs. 1–4 supports the definition of 'successive modules' we gave in Section 3. Similar observations also hold for the graphs showing lines of code against version number (not shown here for reasons of space).

Table 4: Analysis of variance for common coupling

Effect	Degrees of freedom	F	p
Version number	1	2.1×10^4	< 0.0001
Module	16	824.44	< 0.0001
Version number × module	16	163.50	< 0.0001
$R^2 = 0.946$			

Because of the exponential nature of the relationship, a linear model was used, with the natural logarithm of the number of instances of common coupling as the response variable. The constant 0.1 was added before taking logarithms, because eight of the common coupling values were zero. Because of the strong linear dependency between the LOC and version number, the inclusion of the LOC in this model would have resulted in severe numerical instability, and the LOC were therefore not included in this model. Version number, module and version number–module interaction were all found to be significant. The analysis of variance table is given in Table 4.

The model has $R^2 = 0.946$, meaning that 94.6% of the variation in the number of instances of common coupling can be explained by the two variables and their interaction.

6 Conclusions and future work

As described in Section 3, we downloaded 365 versions of Linux. For each version in turn, we looked at the 17 kernel modules and counted the number of lines of code in each module. Then we counted the number of instances of common (global) coupling between each of the kernel modules and all the other modules in that version of Linux. We also recorded the version number as an integer between 1 and 365. We obtained two primary results.

First, we found a module-specific linear dependency between the lines of code and version number that is significant at the 99.99% level; 95.1% of that dependency can be explained by the three effects: version number, module and their interaction. In other words, the number of lines of code in each kernel module increases linearly with version number, and no additional variables are needed to explain this increase; it is an inherent feature of successive versions of Linux. This result is not surprising. After all, successive versions of Linux provide additional functionality. One would expect this increase of functionality to be achieved by both inserting additional code into existing modules and adding new modules. The fact that the size of the kernel grows only linearly could be an indication that the kernel modules are well designed; only a small amount of additional code needs to be inserted to interface the kernel with modified existing modules and new modules that provide the additional functionality.

Secondly, we found that the number of instances of common coupling grows exponentially with version number. This result is also significant at the 99.99% level. In this case, 94.6% of the observed growth can be explained by the three effects: version number, module and their interaction. That is, the exponential growth in common coupling is again an inherent feature of successive versions of Linux.

In Section 2, we related common coupling to fault proneness. Consequently, combining our two results reveals a disturbing trend. Even although the number of lines of code in the kernel grows only linearly, the number of instances of common coupling between each kernel module and all the other Linux modules grows exponentially. Suppose that every statement added to a kernel module were a call to another module. Because the number of lines of code grows only linearly, the number of new instances of coupling induced by these calls, even in this extreme case, can grow only linearly. However, as explained in Section 2, common coupling can increase even when a module does not change. That is how the common coupling increases exponentially even although the number of lines of code increases only linearly.

Common coupling was introduced into Linux from the very beginning, and the nature of common coupling led to an exponential growth in the number of instances in successive versions of Linux. There is no reason to suppose that this growth will be slowed in the future unless Linux is completely restructured with a bare minimum of common coupling. It could be argued that this restructuring of a huge product will mean that the development of Linux will have to be put on hold for many months until the restructuring is complete. On the other hand, if this restructuring is not performed, it seems inevitable that, at some future date, the dependencies between modules induced by common coupling will render Linux extremely hard to maintain. It will then be exceedingly hard to change one part of Linux without inducing a regression fault (an apparently unrelated fault) elsewhere in the product. The only alternative will then be to restructure what, by that time, will be an even larger software product.

In conclusion, our analysis of the growth of common coupling within successive versions of Linux tends to support Ken Thompson's remark quoted in Section 1: "I don't think [Linux] will be very successful in the long run" [2]. However, the future problems we have identified can be averted if Linux is restructured with common coupling reduced to a bare minimum, and if a careful watch is kept to ensure that virtually no additional instances are introduced after the restructuring has been performed.

If general open-source software suffers from a heavy reliance on common coupling, this would be a problem for the future adoption of open source software. However, we have no data or intuition to support such a conclusion. We are currently building a CASE tool to automate the process described in this paper. We will use this CASE tool to measure the growth of common coupling between successive versions of other open-source software to determine whether the potential maintenance problem we have identified in Linux is also present in other open-source software.

7 Acknowledgments

The work of Stephen R. Schach was supported in part by the National Science Foundation under grants CCR-9900662 and CCR-0097056, and the work of A. Jefferson Offutt under grant CCR-0097056.

8 References

1 'What is Linux?' Linux Online, Inc., http://www.linux.org/info/index.html, 2000
2 COOKE, D., URBAN, J., and HAMILTON, S.: 'Unix and beyond: an interview with Ken Thompson', *Computer*, 1999, **32**, (5), pp. 58–64
3 STEVENS, W.P., MYERS, G.J., and CONSTANTINE, L.L.: 'Structured design', *IBM Syst. J.*, 1974, **13**, (2), pp. 115–139
4 PAGE-JONES, M.: 'The practical guide to structured systems design' (Yourdon Press, New York, 1980)
5 OFFUTT, J., HARROLD, M.J., and KOLTE, P.: 'A software metric system for module coupling', *J. Syst. Softw.*, 1993, **20**, (3), pp. 295–308
6 BINKLEY, A.B., and SCHACH, S.R.: 'Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures'. Proceedings of the International Conference on *Software engineering*, ICSE'98, Kyoto, Japan, 1998, pp. 452–455
7 WULF, W., and SHAW, M.: 'Global variables considered harmful', *SIGPLAN Not.*, 1973, **8**, (2), pp. 28–34
8 KAFURA, D., and HENRY, S.: 'Software quality metrics based on interconnectivity', *J. Syst. Softw.*, 1981, **2**, (2), pp. 121–131
9 TROY, D.A., and ZWEBEN, S.H.: 'Measuring the quality of structured designs', *J. Syst. Softw.*, 1981, **2**, (2), pp. 112–120
10 SELBY, R.W., and BASILI, V.R.: 'Analyzing error-prone system structure', *IEEE Trans. Softw. Eng.*, 1991, **17**, (2), pp. 141–152
11 SCHACH, S.R.: 'Object-oriented and classical software engineering' (WCB/McGraw-Hill, Boston, 2002), 5th edn., pp. 181–189, 426
12 BRIAND, L.C., DALY, J., PORTER, V., and WÜST, J.: 'A comprehensive empirical validation of design measures for object-oriented systems'. Proceedings of the 5th International Software Metrics Symposium, Bethesda, MD, 1998, pp. 246–257
13 WANG, S., SCHACH, S.R., and HELLER, G.Z.: 'A case study in repeated maintenance', *J. Softw. Maint. Evol. Res. Prat.*, 2001, **13**, (2), pp. 127–141
14 SCHACH, S.R., JIN, B., WRIGHT, D.R., HELLER, G.Z., and OFFUTT, A.J.: 'Clandestine common coupling'. Computer science technical report 01-02, Vanderbilt University, Nashville, TN, 2001
15 GEISSER, S.: 'Growth curve analysis' in KRISHNAIAH, P.R. (Ed.): 'Handbook of statistics, Vol. 1' (North Holland, Amsterdam, 1980), pp. 89–115
16 KLEINBAUM, D.G., KUPPER, L.L., MULLER, K.E., and NIZAM, A.: 'Applied regression analysis and other multivariable methods' (Duxbury Press, Pacific Grove, CA, 1998), 3rd edn.