



Quality Impacts of Clandestine Common Coupling

STEPHEN R. SCHACH, BO JIN and DAVID R. WRIGHT

srs@vuse.vanderbilt.edu; bo.jin@vanderbilt.edu; davidwrightnyc@hotmail.com

Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN, USA

GILLIAN Z. HELLER

gheller@efs.mq.edu.au

Department of Statistics, Macquarie University, Sydney, Australia

JEFF OFFUTT*

ofut@ise.gmu.edu

Department of Information and Software Engineering, George Mason University, Fairfax, VA, USA

Abstract. The increase in maintenance of software and the increased amounts of reuse are having major positive impacts on the quality of software, but are also introducing some rather subtle negative impacts on the quality. Instead of talking about existing problems (faults), developers now discuss “potential problems,” that is, aspects of the program that do not affect the quality initially, but could have deleterious consequences when the software goes through some maintenance or reuse. One type of potential problem is that of common coupling, which unlike other types of coupling can be clandestine. That is, the number of instances of common coupling between a module M and the other modules can be changed without any explicit change to M. This paper presents results from a study of clandestine common coupling in 391 versions of Linux. Specifically, the common coupling between each of 5332 kernel modules and the rest of the product as a whole was measured. In more than half of the new versions, a change in common coupling was observed, even though none of the modules themselves was changed. In most cases where this clandestine common coupling was observed, the number of instances of common coupling increased. These results provide yet another reason for discouraging the use of common coupling in software products.

Keywords: clandestine coupling, common coupling, coupling, dependencies, Linux, open-source software

1. Introduction

The *coupling* between two units of a software product is a measure of the degree of interaction between those units (Stevens et al., 1974) and, hence, of the dependency between the units. Although all types of coupling are sometimes useful in design, it has been demonstrated that some types have greater potential for introducing faults into software (Wulf and Shaw, 1973; Kafura and Henry, 1981; Troy and Zweben, 1981; Selby and Basili, 1991). 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.

Consider the case where two modules M and N share reference to the same global variable. This was originally called *common coupling* (Stevens et al., 1974), based on the Fortran keyword used to define global variables. Although this type of coupling has also been called *global coupling* (Offutt et al., 1993), this paper uses the traditional

* Corresponding author.

term. Notwithstanding the differences in nomenclature, all categorizations of coupling that we have seen include common coupling.

Furthermore, there also appears to be general agreement that common coupling is more likely to introduce faults than most other forms of coupling. A broad spectrum of different reasons have been put forward, including that common coupling has a negative influence on both maintainability and reusability (Schach, 2002). It has also been experimentally shown (Briand et al., 1998) that common coupling is correlated to fault-proneness.

Notwithstanding the extensive literature on coupling, there is one aspect of common coupling that has not yet been published: The number of instances of common coupling between a module M and the other modules in a software product can change drastically while module M itself remains unchanged. Given the problems that have been identified with the use of common coupling, this can cause serious problems in the quality of software. Even worse, aspects of software development practices that are in large part designed to *improve* the quality of software exacerbate the situation.

For example, suppose modules M and N both reference global variable gv . There is then at least one instance of common coupling between module M and the other modules of the product. But if 50 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 at least 51, even though module M itself is unchanged. We term this *clandestine common coupling*, because M is coupled to modules of which the designers of M were not aware or that did not even exist when M was created.

More precisely, let P_n and P_{n+1} be successive versions of a software product P . Let $M_n \subset P_n$ and $M_{n+1} \subset P_{n+1}$ be successive versions of module $M \subset P$. Let $CC(M)$ denote the number of instances of common coupling between M and the rest of P , where $CC(M_n) > 0$. If $M_n \equiv M_{n+1}$ but $CC(M_n) \neq CC(M_{n+1})$, then there is *clandestine common coupling* between M_{n+1} and the rest of P .

Related work is described in Section 2. Section 3 describes a measurement of clandestine common coupling in Linux. Our results appear in Section 4, and our conclusions in Section 5.

2. Related work

In their 1974 paper, Stevens et al. outlined six levels of coupling. These were presented as an ordered list by Page-Jones (1980), who gave three principal reasons why low coupling between modules is desirable:

- (1) fewer interconnections between modules reduce the chance that a fault in one module will cause a failure in other modules;
- (2) fewer interconnections between modules reduce the chance that changes in one module cause problems in other modules, which enhances reusability; and
- (3) 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. These include (Binkley and Schach, 1998; Offutt et al., 1993;

Table 1. Growth in the size of Linux

| Version # | Year of release | Lines of code | Modules |
|-------------|-----------------|---------------|---------|
| Ver. 1.0 | 1994 | 165,165 | 487 |
| Ver. 2.3.51 | 2000 | 1,690,233 | 3,857 |

Page-Jones, 1980). However, as stated in the previous section, clandestine coupling has not yet been reported.

3. Clandestine common coupling in Linux

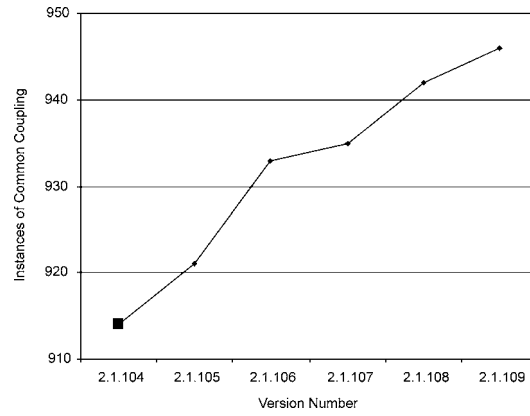
In order to observe the extent to which clandestine common coupling actually occurs in practice, we examined Linux, an open-source operating system (What is Linux, 2000). We looked at all of the available versions of Linux, namely, versions 1.0 through version 2.3.51, a total of 391 versions. Table 1 shows the growth in the size of Linux from 1994 to 2000, making it a good candidate for exhibiting clandestine common coupling.

We successively 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. We ignored multiple references to the same global variable within a given module; counting was performed only at the module level. We also ignored common coupling of constants. In this way we determined the number of instances of common coupling between each kernel module and all the other modules in each version of Linux.

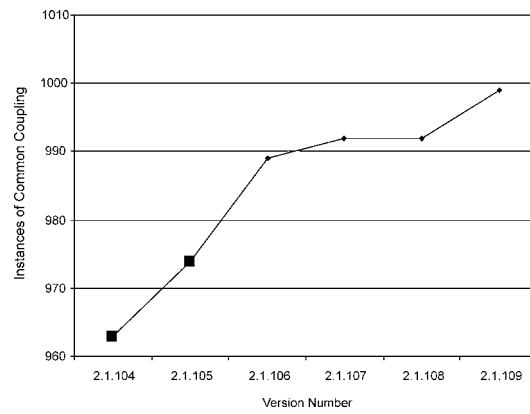
Data for six successive versions of three kernel modules are shown in Table 2, and also in Figure 1, which displays the data of Table 2 in graphical form. A blank in the date column denotes that the code has not changed between successive versions. Thus, for example, version 2.1.104 of Linux kernel module `Panic.c` was released on 21 May 1998. There were 914 instances of common coupling between module `Panic.c` and all the other modules in version 2.1.104 of Linux. As can be seen in the second column of Table 2, the number of instances of common coupling then steadily increased to 946 in version 2.1.109, even though the code for `Panic.c` did not change at all. This is also reflected in Figure 1(a). In each of the three graphs of Figure 1, a large square denotes that the corresponding code version was changed, whereas a small dot means that the

Table 2. Data for six successive versions of three Linux kernel modules

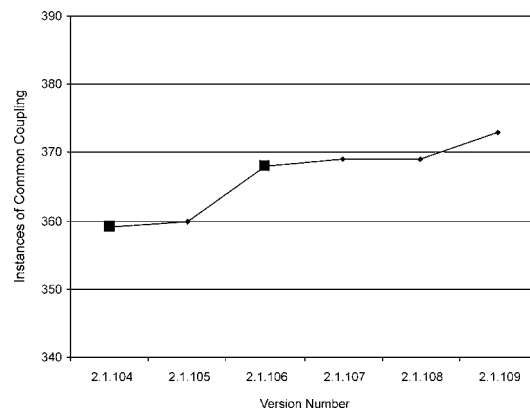
| Version number | Panic.c | | Module.c | | Ksyms.c | |
|----------------|---------|----------|----------|----------|---------|----------|
| | CC | Date | CC | Date | CC | Date |
| 2.1.104 | 914 | 05/21/98 | 963 | 05/21/98 | 359 | 06/04/98 |
| 2.1.105 | 921 | | 974 | 06/07/98 | 360 | |
| 2.1.106 | 933 | | 989 | | 368 | 06/13/98 |
| 2.1.107 | 935 | | 992 | | 369 | |
| 2.1.108 | 942 | | 992 | | 369 | |
| 2.1.109 | 946 | | 999 | | 373 | |



(a)



(b)



(c)

Figure 1. The number of instances of common coupling for three Linux kernel modules. The large squares denote that the code was changed. The small dots show the number of instances of common coupling even though the code itself was not changed, that is, clandestine common coupling.

Table 3. Details of the Linux kernel modules

| | | |
|--|------|-------|
| Total number of modules examined | 6499 | |
| Number of modules modified from the previous version | 1167 | 18.0% |
| Number of modules reused, unchanged, from the previous version | 5332 | 82.0% |

code was not changed between versions. Figure 1 shows that, even when the code is unchanged, the number of instances of common coupling can progressively increase.

4. Results

Details of the 6499 kernel modules appear in Table 3. (Five of the kernel modules do not appear in the earlier versions of Linux, so there are only 6499 kernel modules instead of 6647.) In Table 4 we summarize our results regarding clandestine common coupling in the 5332 unchanged modules in all the versions of the Linux kernel available to us. (We considered only the kernel modules to ensure a manageable number of modules to examine manually.)

We define *upward coupling* as coupling in which the number of instances increases between successive versions. Nearly half (46.5%) of the unchanged modules exhibit upward clandestine common coupling. The mean upward clandestine common coupling is 6.9 instances, with a standard deviation of 11.2. The largest observed upward clandestine common coupling is 230 instances (in version 2.2.17 of kernel module `Sysctl.c`).

Downward clandestine common coupling was observed in 379 of the 5332 unchanged modules (7.1%). The mean downward clandestine common coupling was 8.3 instances with a standard deviation of 19.9. The largest decrease in clandestine common coupling was 156 instances (in version 2.3.0 of kernel module `Exec_domain.c`). In general, these decreases were a consequence of reorganization of the code between kernel modules or between kernel and nonkernel modules, rather than the removal of global variables.

Finally, only 2471 of the 5332 unchanged modules (46.3%) exhibited no clandestine common coupling at all. That is, in those modules there was no change in the number of instances of common coupling between successive versions with identical code.

Table 4. Clandestine common coupling in the 5332 unchanged modules

| | Instances | Percentage |
|--|-----------|------------|
| Modules with upward clandestine common coupling | 2482 | 46.5% |
| Mean upward clandestine common coupling | 6.9 | |
| Standard deviation of upward clandestine common coupling | 11.2 | |
| Largest increase in clandestine common coupling | 230 | |
| Modules with downward clandestine common coupling | 379 | 7.1% |
| Mean downward clandestine common coupling | 8.3 | |
| Standard deviation of downward clandestine common coupling | 19.9 | |
| Largest decrease in clandestine common coupling | 156 | |
| Modules with no clandestine common coupling | 2471 | 46.3% |

Suppose that a programmer is responsible for developing and then maintaining a module *M*. Suppose further that the programmer is fully cognizant of the deleterious effects of common coupling, and therefore makes every effort to minimize the common coupling between *M* and the rest of the product. The results of this paper show that, even if the programmer does not change *M* in any way, subsequent changes made by other programmers to other modules can increase the common coupling between *M* and the rest of the product. Thus, even a single instance of common coupling can insidiously grow until it permeates the entire product, without the knowledge (let alone the cooperation) of the programmer who was originally responsible for that sole instance. This is the danger posed by clandestine common coupling.

A necessary condition for the existence of clandestine common coupling in a software product is that some programmer has previously introduced common coupling into that product. Conversely, clandestine common coupling can be avoided if programmers refrain from using common coupling.

5. Conclusions

Clandestine common coupling is not a theoretical idea; it occurs in real-world software. Specifically, in more than half of the 5332 Linux kernel modules we examined, the number of instances of common coupling increased or decreased even though these kernel modules themselves were unchanged. There were considerably more modules that exhibited clandestine common coupling in the upward direction (2482) than downward (379).

A variety of different reasons have been put forward to explain why and how common coupling is deleterious to the quality of software. The fact that common coupling can be clandestine is an additional reason why the use of common coupling in software products should be discouraged.

Two aspects of the way software is developed turn out to exacerbate this problem. When the ideas of coupling were being formulated in the seventies (Wulf and Shaw, 1973; Stevens et al., 1974), researchers assumed that software systems would be designed and created as complete systems and that each component (function, module, subsystem, etc.) would remain within a single system. It was further assumed that maintenance changes, although an important consideration, would usually be relatively minor modifications to solve specific problems (such as errors or changes in the environment). In practice, however, maintenance takes software through major revisions that make drastic changes in the software such as adding new features, as well as major reorganizations within and between components.

The other aspect of software development is the increase in software reuse. The amount of software component reuse in current software projects has led to large cost savings and increased levels of quality. However, some difficulties still remain, one being that when we pull a software component from its original system and reuse it in another, we must carefully consider all aspects of how the component integrates with the new system. The couplings determine a major part of this integration, and common coupling is easy to overlook.

The danger with the increase of reuse is even more severe if components whose source code is not readily available are reused, as evidenced by the following example. One of the authors (Offutt) was teaching a class recently where a student had a very strange failure in a program that was processing form data in a Java servlet. Specifically, sometimes the data were lost, but sometimes they were saved. After several weeks of effort, the root cause was determined to be that her program had accidentally overridden a public variable that was defined in a Java library class from which she was inheriting. The variable did not need to be public (common) and should not have been, but it was. The student's servlet introduced clandestine common coupling in a way that was almost impossible to detect. This kind of mistake is particularly hard to debug because the failure is so very far from the cause, and the source code for the library was not easily available.

Both maintenance and reuse are used to increase the quality of the resulting software, but they both mean the initial designers cannot foresee all consequences of their design decisions. Software developers now need to consider *potential faults*, that is, aspects of the program that do not affect the quality initially, but could have deleterious consequences when the software goes through some maintenance or reuse. Common coupling is one type of potential fault, because if a component has a global variable, it is automatically available in the scope of other software components in the new system, or new parts of the modified system. This availability can easily lead to clandestine common coupling, sometimes even accidentally.

Acknowledgements

This research was supported in part by the National Science Foundation under grants CCR-9900662 and CCR-0097056.

References

- Binkley, A.B. and Schach, S.R. 1998. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures, *Proceedings of the 20th International Conference on Software Engineering*, pp. 452–455.
- Briand, L.C., Daly, J., Porter, V. and Wüst J. 1998. A comprehensive empirical validation of design measures for object-oriented systems, *Proceedings of the 5th International Software Metrics Symposium*, pp. 246–257.
- Kafura, D. and Henry, S. 1981. Software quality metrics based on interconnectivity, *Journal of Systems and Software* 2(2): 121–131.
- Offutt, J., Harrold, M.J. and Kolte, P. 1993. A software metric system for module coupling, *Journal of Systems and Software* 20(3): 295–308.
- Page-Jones, M. 1980. *The Practical Guide to Structured Systems Design*. New York, Yourdon Press, 1980.
- Schach, S.R. 2002. *Object-Oriented and Classical and Software Engineering*, 5th ed. Boston, WCB/McGraw-Hill.
- Selby, R.W. and Basili, V.R. 1991. Analyzing error-prone system structure, *IEEE Transactions on Software Engineering* 17(2): 141–152.
- Stevens, W.P., Myers, G.J. and Constantine, L.L. 1974. Structured design, *IBM Systems Journal* 13(2): 115–139.
- Troy D.A. and Zweben, S.H. 1981. Measuring the quality of structured designs, *Journal of Systems and Software* 2: 112–120.
- What is Linux? 2000. Linux online, <http://www.linux.org/info/index.html>, 6 March.
- Wulf, W. and Shaw, M. 1973. Global variables considered harmful, *SIGPLAN Notices* 8(2): 28–34.

Stephen R. Schach is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University in Nashville, TN. Steve is the author of over 100 refereed publications. He has written eight software engineering textbooks, including *Object-Oriented and Classical Software Engineering*, 5th ed. (WCB/McGraw-Hill, 2002). He consults internationally on software engineering topics. Steve's research interests are in software maintenance. He obtained his Ph.D. from the University of Cape Town in South Africa.
E-mail: srs@vuse.vanderbilt.edu.

Bo Jin obtained his M.S. in computer science at Vanderbilt University in Nashville, TN in 2002. His main research interest is in software maintenance.
E-mail: bo.jin@vanderbilt.edu.

David R. Wright obtained his B.S. in computer science and English at Vanderbilt University in Nashville, TN in 2001.
E-mail: davidwrightnyc@hotmail.com.

Gillian Z. Heller is a Senior Lecturer in the Department of Statistics at Macquarie University, Sydney, Australia, where she has been for the last 10 years. Her B.Sc. (Hons) and Ph.D. degrees are in mathematical statistics, from the University of Cape Town, South Africa, and her M.Sc. in operations research is from the University of South Africa. Gillian's research interests are in discrete distribution theory, with applications in biostatistics.
E-mail: gheller@efs.mq.edu.au.

Jeff Offutt is an Associate Professor of Information and Software Engineering at George Mason University and holds part-time visiting positions at NIST and Skövde University. Jeff's current research interests include software testing, analysis and testing of Web applications, software maintenance and object-oriented program analysis. He has published over seventy-five refereed papers. Jeff was program chair for ICECCS 2001 and is on the editorial boards for the IEEE Transactions on Software Engineering, the Journal of Software Testing, Verification and Reliability, the Journal of Software and Systems Modeling and the Software Quality Journal. Jeff's Ph.D. is from the Georgia Institute of Technology. He previously held a faculty position in the Department of Computer Science at Clemson University.
E-mail: ofut@ise.gmu.edu.