Categorization of Common Coupling and its Application to the Maintainability of the Linux Kernel

Liguo Yu, Stephen R. Schach, Kai Chen
Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN, USA
liguo.yu@vanderbilt.edu, srs@vuse.vanderbilt.edu, kai.chen@vanderbilt.edu

and Jeff Offutt
Department of Information and Software Engineering, George Mason University, Fairfax, VA, USA
ofut@ise.gmu.edu

Please send all correspondence to:

Stephen R. Schach

Department of Electrical Engineering and Computer Science
Box 351679, Station B
Vanderbilt University
Nashville TN 37235-1679, USA

Tel: (615) 322–2924
Fax: (615) 343–5459
E-mail:srs@vuse.vanderbilt.edu

Categorization of Common Coupling and its Application to the Maintainability of the Linux Kernel

ABSTRACT

Software product lines are typically comprised of kernel modules, which are common to all installations, and optional modules, which are used in some installations. Data coupling between modules, especially common coupling, has long been considered a source of concern in software design, but the issue is somewhat more complicated with software product lines. This paper presents a refined categorization of common (global) coupling based on definitions and uses between kernel and nonkernel modules and applies the categorization to a case study. Common coupling is usually avoided when possible because of the potential for introducing risky dependencies among software modules. The relative risk of these dependencies partially depends on the specific definition-use relationships. In a previous paper, we presented results from a longitudinal analysis of multiple versions of the open-source operating system Linux. This paper applies the new common coupling categorization to version 2.4.20 of Linux, counting the number of instances of common coupling between each of the 26 kernel modules and all the other nonkernel modules. We also categorized each coupling in terms of the definition-use relationships. Results show that the Linux kernel contains a large number of common couplings of all types, raising a concern about the long-term maintainability of Linux.

KEY WORDS: D.2.2.d Modules and interfaces, D.2.7.g Maintainability, D.2.8 Metrics/ Measurement, D.2.10.h Quality analysis and evaluations.

1. INTRODUCTION

Software product lines [1, 2] are typically comprised of a *kernel* collection of modules plus *optional* and *variant* modules or features [3]. Couplings among modules have great impact on the quality of the software. Coupling, particularly by way of global and non-local data members (called *common coupling*), has long been considered to present risks for software development, particularly for reuse and maintenance [4, 5]. As explained at the end of Section 2, this is a particular problem when the coupling is *clandestine* [6], that is, when existing modules are coupled to new modules by way of the new modules making use of the same common variables that the existing modules use.

However, we have noted that not all common coupling is the same. Most particularly, for software product lines, couplings within the kernel are not the same as couplings between kernel and nonkernel modules. Further, the risks of common coupling are strongly related to the pattern of definitions and uses of the variables involved in the common coupling. This paper presents a method for analyzing common coupling based on the definition-use patterns of common coupled variables, a method that is developed into a metric for software product lines.

This current paper follows a longitudinal study of the maintainability of the Linux kernel [7–9] that examined the kernel modules of nearly 400 successive versions of Linux. Our major results were that the number of lines of code in each kernel module increases *linearly* with version number, but that the number of instances of common coupling between each kernel module and all the other Linux modules grows *exponentially*. Both results were significant at the 99.99% level.

As explained in Section 2, common coupling is connected to fault-proneness. Consequently, we raised the concern that the dependencies between modules induced by

common coupling have the potential to render Linux hard to maintain at some future date, ending reference [7] by stating: "In conclusion, our analysis of the growth of common coupling within successive versions of Linux tends to support Ken Thompson's remark [10]: 'I don't think [Linux] will be very successful in the long run'."

Needless to say, these conclusions are not popular with Linux users when presented at conferences [8, 9]. An audience member at an ISSRE 2002 panel raised an intriguing question. He wondered how widely the values of global variables could be changed within Linux. For example, if global variables can be changed in just a few places, Linux would be considerably more maintainable than if global variables can be changed in many places. The answer to this question requires a definition-use analysis of Linux.

Our first step was to create a refinement of the traditional notion of common coupling. This resulted in an analysis technique and metric for evaluating software product lines. The analysis technique is based on the definition-use patterns of the coupled variables. It has applications to evaluation of maintenance and reuse, and can also serve as a tool to help designers avoid potential problems. The analysis technique and metric are presented in Sections 3 and 5. The technique has been applied as a case study to the widely used open-source operating system Linux, as described in Sections 4 and 6 with results in Section 7. The paper begins by summarizing traditional common coupling in Section 2.

2. MODULE DEPENDENCIES

The *coupling* between two units of a software product is a measure of the degree of interaction between those units and, hence, of the dependency between the units. Stevens, Myers, and Constantine [11] presented six levels of coupling between pairs of modules. 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 modularization. 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*).

Too much coupling will make a software product fault prone, difficult to reuse, and difficult to maintain. Page-Jones provided several reasons for minimizing the number of instances of coupling between modules [12]: (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 will cause problems in other modules; and (3) fewer interconnections between modules reduce programmer time in understanding the details of other modules. So, coupling should be used with caution.

In this paper, we consider the classical coupling category *common coupling,* called *global coupling* in the categorization of Offutt et al. [13]. Two modules P and Q are common coupled if P and Q share references to the same global variable.

It has been shown that coupling is related to fault-proneness [4, 5], although it has not yet been explicitly shown to be related to maintainability. On the other hand, there is as yet no 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.

There are three reasons why we consider *common coupling* in this paper. First, 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 [14]. Second, there is considerable controversy regarding what precisely constitutes weak or strong coupling, let alone which categorization of coupling should be followed. However, all categorizations we have seen include a form of coupling that corresponds to classical common coupling, and there seems to be unanimity that common coupling is risky.

The third reason we concentrated on common coupling is that common coupling possesses the unfortunate property that the number of instances of common coupling between a module M and the other modules can change drastically, even if module M itself never changes, an effect that has been called *clandestine common coupling* [6]. 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 though module M itself is unchanged. Bearing in mind that the size of Linux has increased nearly 1000% since version 1.0, it should come as no surprise that clandestine common coupling is widespread in Linux and continues to increase [6]. The resulting widespread presence of strong coupling is a major reason for examining the maintainability of Linux.

3. DEFINITION-USE ANALYSIS OF SOFTWARE PRODUCT LINES

Evaluating the potential affects of common coupling requires an analysis of the definition-use patterns. Every occurrence of a variable vvv in a program represents either a *definition* (or *def*) of that variable (for example, read (vvv) or vvv = 3) or a *use* of that variable (for example, x = vvv + 3 or if (vvv > 7) print (y)) [15]. Suppose that modules M_1 and M_2 are common coupled because they both reference global variable gv. There are three possible situations:

(1) Only M_1 can change the value of gv. That is, gv is defined in M_1 but only used in M_2.

(2) Only M_2 can change the value of gv. That is, gv is defined in M_2 but only used in M_1.

(3) Both M_1 and M_2 can change the value of gv. That is, gv is defined in both M_1 and M_2.

Situations (1) and (2) pose less risk for maintenance than (3) because there are fewer dependencies between the two modules when only one of them can change the value of gv. The dependencies are localized, thus effects of future changes can be easily determined. When only one module can define gv, changes to the other module cannot affect the defining module. Furthermore, within a given module that can change global variable gv, fewer places that can change gv is better.

In def-use analysis, each instance of a variable is labeled as either a definition or a use of that variable. The next section describes how def-use analysis is applied to a well-known and widely used software product line example, Linux.

4. METHOD AND RESULTS

We examined the latest stable version of Linux for an Intel-based computer, version 2.4.20. Linux contains a kernel component (27 modules, or files) and nonkernel components (4,061 modules) that are chosen for individual installations or for specific hardware platforms. Together, Linux comprises 3,332,168 lines of code (as computed with the Linux cross-referencing tool lxr). We used lxr to examine every instance of every global variable in every module, and determined whether that instance was a definition or a use. This examination and determination was performed independently by two of the authors (Yu and Chen), and the differences were reconciled.

Table I: Overview of Linux global variables.

Total number of global variables	Number of unique instances of a global variable in kernel modules	Number of unique instances of a global variable in nonkernel modules	Total number of instances of global variables in kernel modules	Total number of instances of global variables in nonkernel modules
99	193	2,814	1,022	14,688
Totals	3,007		15,710	

As shown in Table I, there are 15,710 instances of the 99 global variables. Of these instances, 1,022 are in kernel modules, and the remaining 14,688 instances in nonkernel modules. If multiple occurrences of a global variable within a given module are ignored, there are 193 unique instances of a global variable in a kernel module, and 2,814 unique instances of a global variable in a nonkernel module, or 3,007 unique instances in all.

Table II shows the number of global variables referenced by each pair of Linux kernel variables. For example, acct.c (A) and capability.c (B) both reference the same single global variable, whereas there are 15 global variables that are referenced by both time.c (W) and timer.c (X). Most kernel modules reference at least one global variable; only three modules,

module.c (L), pm.c (N), and user.c (Z), have no common coupling with any other kernel module.

Table II. The number of global variables referenced by each pair of Linux kernel modules. Key to modules: A: acct.c; B: capability.c; C: context.c; D: dma.c; E: exec_domain.c; F: exit.c; G: fork.c; H: info.c; I: itimer.c; J: kmod.c; K: ksyms.c; L: module.c; M: panic.c; N: pm.c; O: printk.c; P: ptrace.c; Q: resource.c; R: sched.c; S: signal.c; T: softirq.c; U: sys.c; V: sysctl.c; W: time.c; X: timer.c; Y: uid16.c; Z: user.c.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A		1	1		1	1	1	1	2	1	2		1		1	1		2	1	2	1	2	1	2	1	
B	1		1		1	2	2		1	1	2		1		1	2		2	2	1	2	2		2	1	
C	1	1			1	1			1	1			1		1	1		1	1	1	1	1		1	1	
D											1															
E	1	1	1			1	1		1	1			1		1	1		1	1	1	1	1		1	1	
F	1	2	1		1		2		1	1	1		1		1	2		2	2	1	2	1		2	1	
G	1	2			1	2		1	2	3	3		1		1	2		5	2	1	3	2		2	1	
H	1						1		1		1							1			1			1		
I	2	1	1		1	1	2	1		1	1		1		1	1		2	1	1	2	1		2	1	
J	1	1	1		1	1	3		1				1		1	1		2	1	1	1	4		1	1	
K	2	2		1		1	3	1	1							1	2	5	1	5	7	6	2	5		
L																										
M	1	1	1		1	1	1		1	1					1	1		1	1	1	1	4		1	1	
N																										
O	1	1	1		1	1	1		1	1			1			1		1	1	1	1	1		1	1	
P	1	2	1		1	2	2		1	1	1		1		1			1	2	1	2	1		2	1	
Q											2															
R	2	2	1		1	2	5	1	2	2	5		1		1	1			2	1	3	1		4	1	
S	1	2	1		1	2	2		1	1	1		1		1	2		2		1	2	3		2	1	
T	2	1	1		1	1	1		1	1	5		1		1	1		1	1		1	1		3	1	
U	1	2	1		1	2	3	1	2	1	7		1		1	2		3	2	1		8		3	1	
V	2	2	1		1	1	2		1	4	6		4		1	1		1	3	1	8			1	1	
W	1										2													15		
X	2	2	1		1	2	2	1	2	1	5		1		1	2		4	2	3	3	1	15		1	
Y	1	1	1		1	1	1		1	1			1		1	1		1	1	1	1	1		1		
Z																										

There is clearly a widespread incidence of common coupling between kernel modules. In order to investigate the maintainability of the Linux kernel, it is necessary to study in detail all these instances of common coupling between kernel modules, as well as all instances of coupling between kernel and nonkernel modules.

5. DEF-USE ANALYSIS DEFINITIONS AND CATEGORIES

Common coupling is normally considered to be a bi-directional relationship. However, it seems clear that definitions and uses have decidedly different affects on maintenance, thus the detailed

analysis that we need to perform in order to consider the effects of maintenance requires a directional relationship. We choose to represent def-use relationships graphically as illustrated in Figures 1 through 3. In Figure 1 the outer rectangle denotes the kernel, so M_1 and M_2 are both kernel modules. The arrow from M_1 to M_2 denotes that M_1 defines gv (at least once) and M_2 uses gv (at least once).

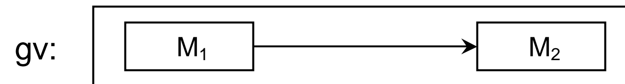

Figure 1: An example of our graphical notation for representing common coupling. Modules M_1 and M_2 both reference global variable gv; the arrow means that M_1 defines gv and M_2 uses it.

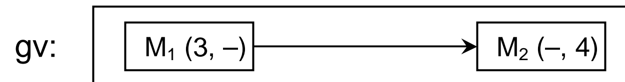

Figure 2: Figure 1 with explicit def-use multiplicities added; M_1 defines gv three times and M_2 uses gv four times.

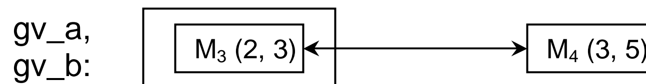

Figure 3: A further example of our graphical notation for representing common coupling. Modules M_3 and M_4 both define and use global variables gv_a and gv_b.

Figure 2 adds "multiplicity" information to Figure 1, showing that gv is defined three times in M_1 and used four times in M_2. In general, a pair (p, q) denotes that the global variable is defined p times and used q times in the module in which the pair appears.

When a global variable is defined in two modules, the arrow connecting the boxes representing the modules is double-headed. This is shown in Figure 3, which shows that global variables gv_a and gv_b are both defined and used in modules M_3 and M_4. The outer rectangle denotes that M_3 is a kernel module, whereas M_4 is a nonkernel module. Global variables gv_a and gv_b are together defined twice in M_3 and three times in M_4, and together are used three

times in M_3 and five times in M_4. Additional features of our notation will be explained when they are introduced.

As shown in Table III and described below, we classify global variable into five categories. All forms of common coupling are considered to be undesirable (although sometimes necessary), but the ways global variables are used can be classified as safe or unsafe with respect to maintainability. Linux is composed of a kernel set of modules, which are included in all installations of Linux, and nonkernel modules that are included in specific installations. This configuration is found in many software product lines [3], thus the analysis can be applied to software product line systems. Because kernel modules are included in all installations of Linux, we are most concerned with effects of common coupling on the kernel. We consider global variables to be either safe or non-safe with respect to the kernel. Consequently, we are concerned with defs in both kernel and nonkernel modules and uses in kernel modules, but uses in nonkernel modules are not relevant.

Table III. Categorization of global variables.

Category number	Description
1	A global variable defined in kernel modules but not used in any kernel modules.
2	A global variable defined in one kernel module and used in one or more kernel modules.
3	A global variable defined in more than one kernel module, and used in one or more kernel modules
4	A global variable defined in one or more nonkernel modules and used in one or more kernel modules.
5	A global variable defined in one or more nonkernel modules and defined and used in one or more kernel modules.

- <u>k→k safe</u>: A global variable is defined to be *kernel-to-kernel safe* (or k→k safe for brevity) if there are no defs in the kernel that can reach uses in the kernel.

Thus, a change to a k→k safe variable in a kernel module cannot affect the kernel. So, if there is no def of a specific global variable in a kernel module or there is no use of that global variable in a kernel module, the global variable is k→k safe.

- k→k unsafe: A global variable is defined to be *kernel-to-kernel unsafe* (k→k unsafe) if it is not k→k safe.

In the various operating systems we have examined, we have noticed that there are a number of k→k unsafe global variables that are defined in only one kernel module. (In the case of Linux, this is reported in Section 6.) Accordingly, we need a further definition:

- minimally k→k unsafe: A k→k unsafe global variable is defined to be *minimally kernel-to-kernel unsafe* (minimally k→k unsafe) if it is defined in only one kernel module and used in one or more kernel modules.

We require two further definitions to describe the effect of a change in a nonkernel module to a kernel module.

- nonk→k safe: A global variable is defined to be *nonkernel-to-kernel safe* (nonk→k safe) there are no defs in nonkernel modules that can reach uses in kernel modules.

Thus, a change to a nonk→k safe global variable in a nonkernel module cannot affect the kernel.

- <u>nonk→k unsafe</u>: A global variable is defined to be *nonkernel-to-kernel unsafe* (nonk→k unsafe) if it is not nonk→k safe.

We next use these definitions to divide common variables into five categories. The categories have increasing levels of effect on the maintainability of the kernel, and thus can be considered be increasingly objectionable. Later in the paper we apply these categories to the Linux kernel and count the number of common variables of each type. As previously explained, we consider uses only inside the kernel, but we consider definitions both inside and outside the kernel.

5.1 Category-1 Global Variables

A category-1 global variable is defined in a kernel module but has no kernel uses. A category-1 global variable is used in one or more nonkernel modules, but this is not important from the viewpoint of the maintainability of the kernel. Figure 4 depicts a category-1 global variable.

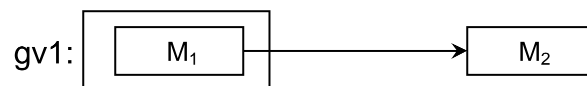

Figure 4: A category-1 global variable gv1.

Category-1 global variables can probably be considered the least objectionable with respect to maintainability. First, there is no common coupling between kernel modules that involves category-1 variables, so a change to a category-1 variable cannot affect another kernel module. Thus, category-1 global variables are *k→k safe* as reflected in Table IV. Second, there are no uses of category-1 variables in kernel modules, so a modification to a nonkernel module involving a category-1 global variable cannot induce a regression fault in a kernel module.

Accordingly, category-1 global variables are also *nonk→k safe*. The fact that category-1 global variables are both k→k safe and nonk→k safe means that category-1 global variables have minimal impact on the maintainability of kernels.

Table IV: The safety of global variables in each category.

	k→k safe	Minimally k→k unsafe	k→k unsafe
nonk→k safe	Category 1	Category 2	Category 3
nonk→k unsafe	Category 4	—	Category 5

5.2 Category-2 Global Variables

A category-2 global variable is defined in one kernel module, and is also used in one or more kernel modules. Category-2 global variables may be used in nonkernel modules, but that use is not important. Figure 5 depicts a category-2 global variable gv2.

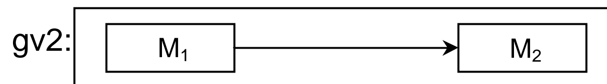

Figure 5: A category-2 global variable gv2.

As with category-1, a modification to a category-2 global variable in a nonkernel module cannot affect a kernel module because there are no definitions of category-2 global variables in nonkernel modules. That is, category-2 global variables are nonk→k safe. However, category-2 global variables are k→k unsafe because a change to the kernel module that defines the variable can affect the kernel module that uses it. By definition, however, a category-2 global variable is defined in only one kernel module, and thus it is *minimally k→k unsafe*.

5.3 Category-3 Global Variables

A category-3 global variable is defined in more than one kernel module, and is also used in one or more kernel modules. Category-3 global variables may be used in nonkernel modules, but that use is not important. Figure 6 depicts a category-3 global variable.

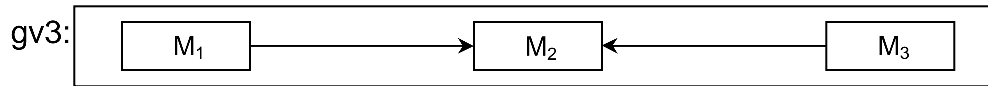

Figure 6: A category-3 variable gv3.

As with category 2, category-3 global variables are *nonk→k safe*. However, they are *k→k unsafe*. They are not minimally k→k unsafe, because a category-3 global variable is defined in more than one kernel module.

5.4 Category-4 Global Variables

A category-4 global variable is defined in one or more nonkernel modules, and used in one or more kernel modules. As with category-2 and -3 global variables, uses in nonkernel modules are not important. Figure 7 depicts a category-4 global variable gv4. Figure 7 is the same as Figure 4, but with the direction of the arrow reversed.

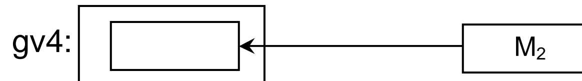

Figure 7: A category-4 variable gv4.

Category-4 global variables are undesirable. They are *k→k safe* but *nonk→k unsafe*. That is, a kernel module that uses a category-4 global variable is vulnerable to modifications to that global variable in a nonkernel module that defines the variable. The principle of "separation of concerns" tells us that changes to nonkernel modules should not be able to affect kernel modules.

5.5 Category-5 Global Variables

A category-5 global variable is defined in one or more nonkernel modules, defined in one or more kernel modules and used in one or more kernel modules. Figure 8 depicts a category-5 global variable.

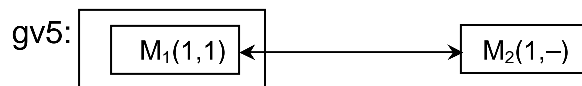

Figure 8: A category-5 variable gv5.

Category-5 global variables are both *k→k unsafe* and *nonk→k unsafe*. That is, a kernel module that contains a category-5 global variable is vulnerable to modifications to both a kernel module and a nonkernel module in which that global variable is defined. It is extremely difficult to minimize the impact of changes that involve category-5 global variables.

6. LINUX CASE STUDY

These categories have been applied as a case study to the Linux operating system. As explained in Section 4, global variables of all five categories were found and counted.

6.1 Category-1 Global Variables in Linux

A category-1 global variable is defined in one or more kernel modules (files) and used in one or more nonkernel modules. Figure 9 shows an example of a global variable, total_forks, that falls into this category. It is defined twice in kernel module fork.c, and is used twice in a nonkernel module. Global variable prof_buffer, shown in Figure 10, is also in category-1. It is defined once in a kernel module (timer.c) and is used a total of 49 times in 24 nonkernel modules.

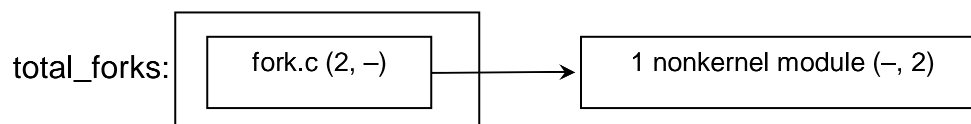

Figure 9: Common coupling of category-1 global variables total_forks.

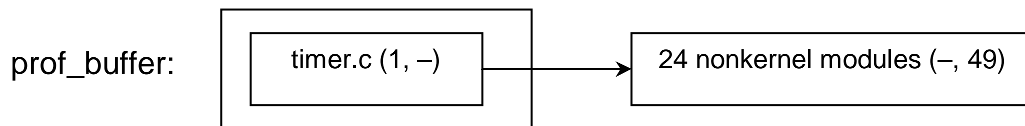

Figure 10: Common coupling of category-1 global variable prof_buffer.

Table V shows that 23 Linux global variables fall into category 1. Ignoring multiple instances of a global variable in a module, there are 25 unique instances of a category-1 global variable in a kernel module, and there are 35 such instances altogether. All these instances are definitions.

Table V: Summary of definitions and uses of global variables in Linux.

Category number	Number of global variables	Kernel modules			Nonkernel modules		
		Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses	Number of unique instances of a global variable	Number of instances of definitions	Number of instances of uses
1	23	25	35	–	220	–	389
2	28	76	36	208	1,041	–	4,437
3	4	10	25	15	91	–	302
4	24	27	–	65	66	40	171
5	20	55	180	458	1,396	1,732	7,617
Overall	99	193	276	746	2,814	1,772	12,916

6.2 Category-2 Global Variables in Linux

A category-2 global variable is defined in one kernel module and used in one or more kernel modules. (The global variable may be used in a nonkernel module, but this is not important.) Figure 11 shows four category-2 global variables: overflowuid, overflowgid, fs_overflowuid, and fs_overflowgid. They are each defined once (a total of 4 definitions) in sys.c, and are each used once in ksyms.c and sysctl.c (a total of 4 times in each module) and a total of 22 times in 16 nonkernel modules (files).

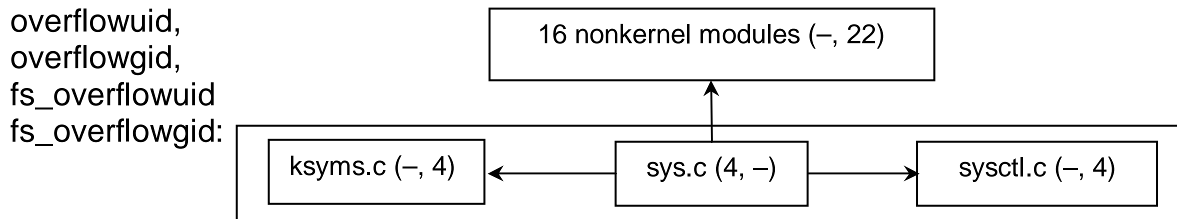

Figure 11: Category-2 global variables overflowuid, overflowgid, fs_overflowuid, and fs_overflowgid

Table VI: Details of category-2 global variables in Linux nonkernel modules.

Variable	Nonkernel modules		
	Number of modules containing a unique instance of a global variable	Number of definitions	Number of uses
jiffies	887	–	3,977
All others	154	–	460
Total	1,041	–	4,437

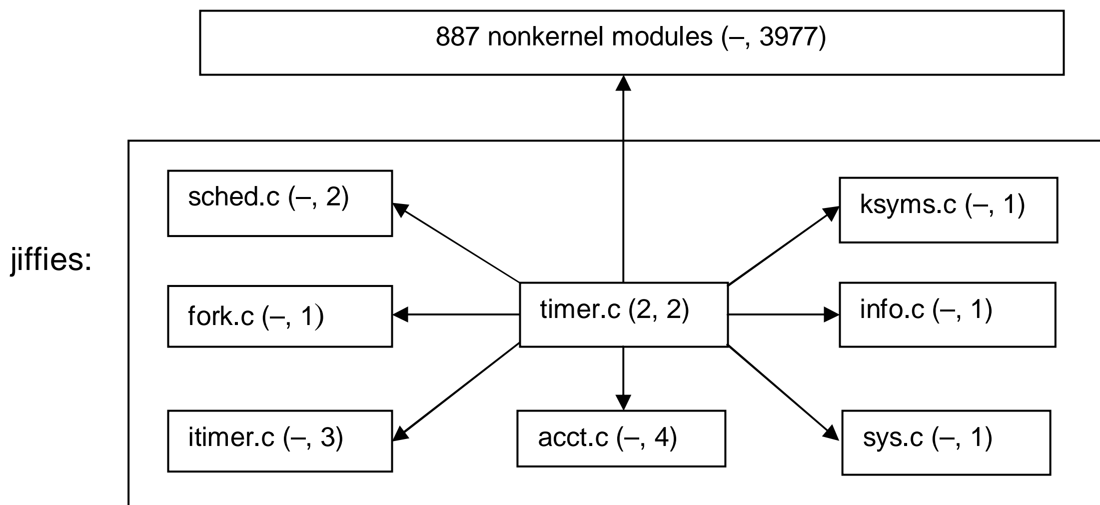

Figure 12: Category-2 global variable jiffies.

Table V shows that there are 28 category-2 global variables in Linux. Ignoring multiple instances of a global variable in a kernel module, there are 76 instances of a category-2 global variable in a kernel module. Considering all instances of category-2 global variables in kernel modules, there are 36 definitions and 208 uses. The number of unique instances of a category-2 variable in a nonkernel module and the number of instances of uses are 1,041 and 4,437,

respectively. This is mainly due to global variable jiffies, details of which are shown in Table VI. The global variable jiffies is defined in timer.c only twice, but it is used 3,977 times in 887 nonkernel modules, as shown in Figure 12. Excluding jiffies, the other 27 global variables in category-2 are found in only 154 nonkernel files; there are only 460 instances of their uses.

Detailed analysis of jiffies shows that 716 out of the 887 nonkernel modules are related to drivers and structures. In fact, 3,381 out of the 3,977 instances of uses relate to drivers and arch (platform-specific) files. That is, in any one implementation of Linux, there are only 596 uses of category-2 global variables, plus a few more in driver modules and in platform-specific modules. Unfortunately, this does not change the maintainability of the Linux kernel. When modifications are made to Linux, they must be made to the entire system, not just to a specific small set of drivers and platform-specific modules.

6.3 Category-3 Global Variables in Linux

A category-3 global variable is defined in more than one kernel module, and used in one or more kernel modules (and possibly in one or more nonkernel modules, but this is not important). Two of the global variables that fall into this category are time_constant and xtime, depicted in Figures 13 and 14, respectively. The dotted lines in Figure 14 separate the modules that have instances of both definitions and uses of the global variable (timer.c, time.c) from the modules that have instances of only uses of the global variable (acct.c, ksyms.c).

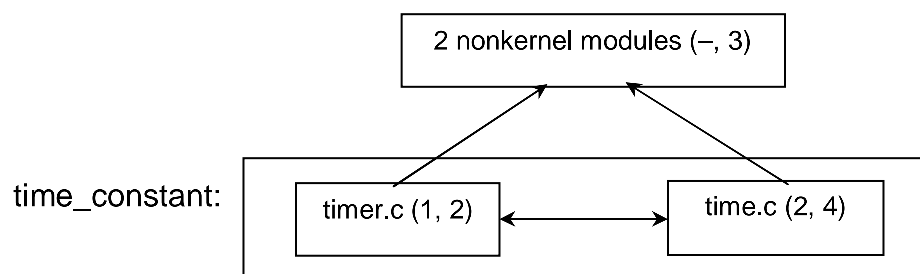

Figure 13: Category-3 global variable time_constant.

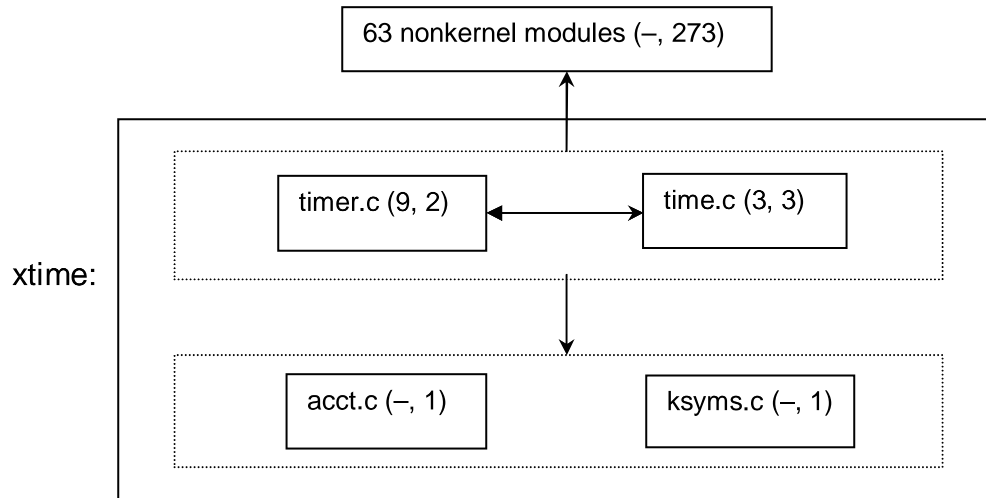

Figure 14: Category-3 global variable xtime.

A summary of the definitions and uses of the category-3 global variables appears in Table V. In addition, details of those variables are given in Table VII; one global variable, xtime, is responsible for nearly ninety percent of the instances of uses in nonkernel modules.

Table VII: Details of category-3 global variables in Linux nonkernel modules.

Variable	Nonkernel modules		
	Number of modules containing a global variable	Number of definitions	Number of uses
xtime	63	–	273
Others	28	–	29
Total	91	–	302

6.4 Category-4 Global Variables in Linux

A category-4 global variable is defined in one or more nonkernel modules and used in one or more kernel modules. Two of the global variables belonging to this category, child_reaper and system_utsname, are shown in Figures 15 and 16, respectively. A summary of definitions and uses appears in Table V.

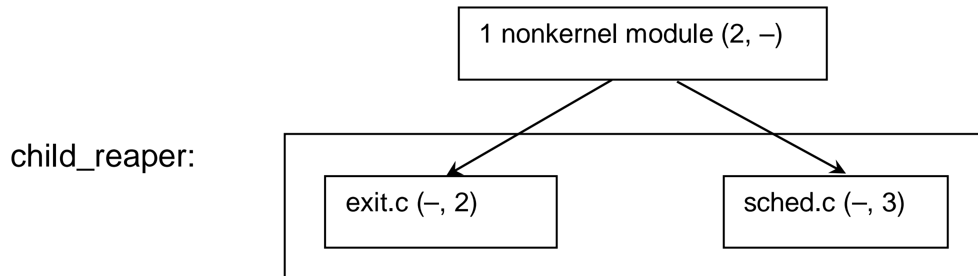

Figure 15: Category-4 global variable child_reaper.

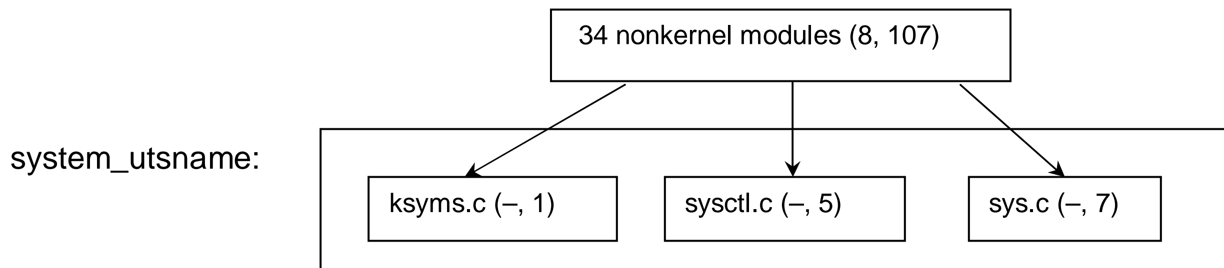

Figure 16: Category-4 global variable system_utsname.

6.5 Category-5 Global Variables in Linux

A category-5 global variable is defined and used in one or more nonkernel modules and one or more kernel modules. Linux contains 20 category-5 global variables. Figure 17 shows global variables panic_timeout and stop_a_enabled and Figure 18 depicts the definitions and uses of global variable init_task. Data from Figures 17 and 18, as well as from the other category-5 global variables, appear in Tables VIII and IX. In particular, there are 1,732 nonkernel definitions of category-5 global variables, 1508 just for current; details of the definitions and uses of current are shown in Figure 19.

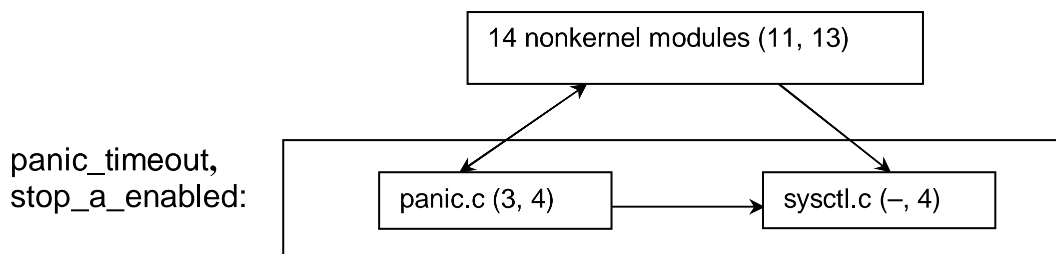

Figure 17: Category-5 global variables panic_timeout and stop_a_enabled.

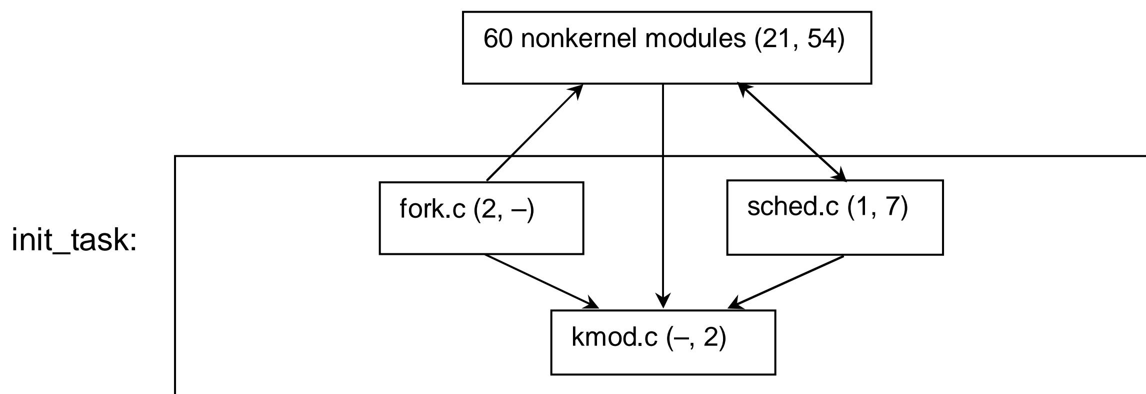

Figure 18 Category-5 global variable init_task.

Table VIII: Details of category-5 global variables in Linux kernel modules.

Global variable	Kernel modules		
	Number of modules containing a global variable	Number of definitions	Number of uses
current	18	114	382
Others	37	66	76
Total	55	180	458

Table IX: Details of category-5 global variables in Linux nonkernel modules.

Global variable	Nonkernel modules		
	Number of modules containing a global variable	Number of definitions	Number of uses
current	1077	1508	7290
Others	319	224	327
Total	1396	1732	7617

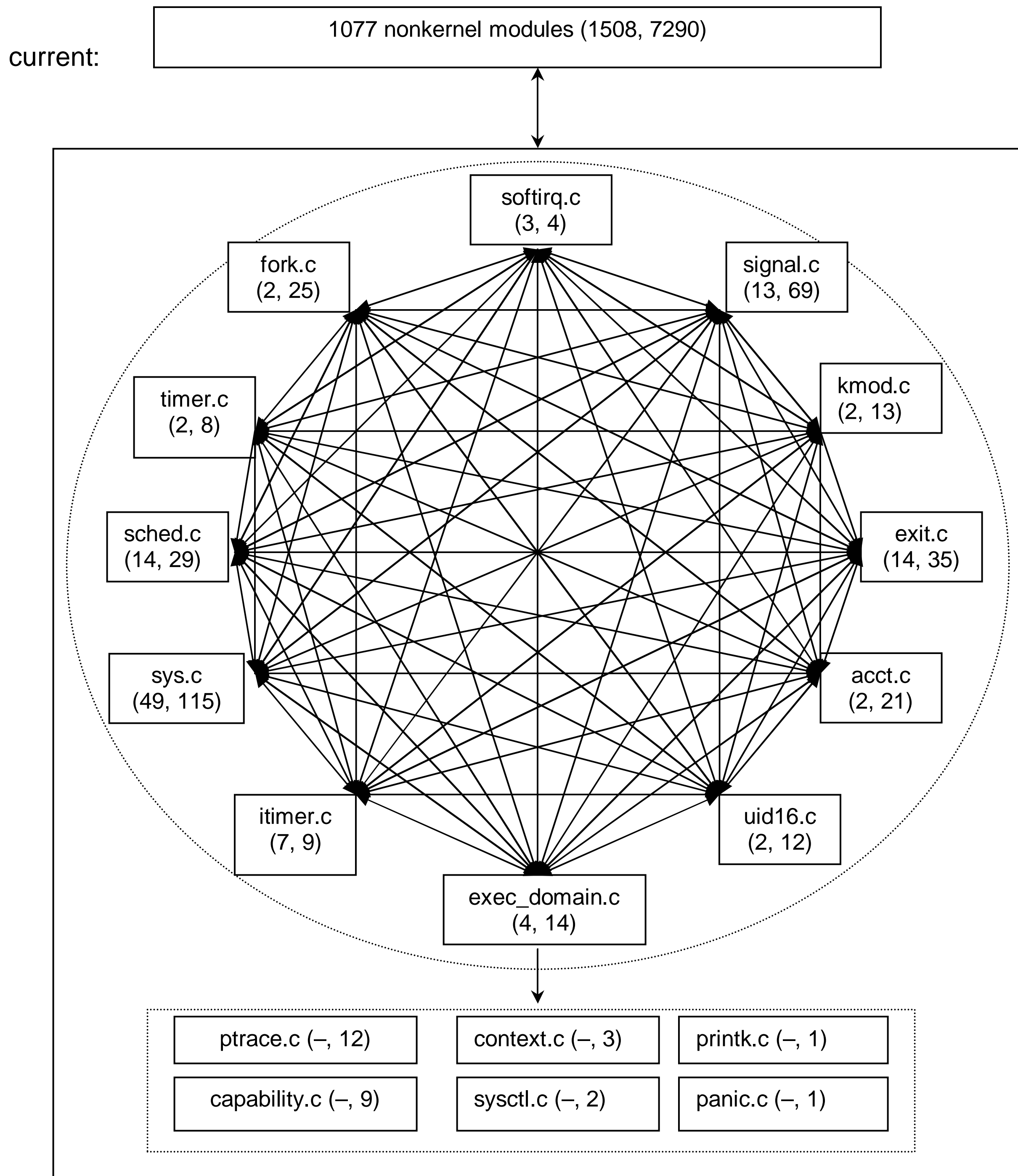

Figure 19 Category-5 global variable current.

In our opinion, the large number of definitions of category-5 global variables represents a risk to the long-term maintainability of the Linux kernel.

7. THE MAINTAINABILITY OF THE LINUX KERNEL

Table IV summarizes the safety characteristics for each of the five global variable categories, that is, it shows the potential effect that a modification to a global variable in each category can have on the Linux kernel. Categories 1 through 3 are all *nonk→k safe*. That is, a modification to a global variable in a nonkernel module cannot affect a kernel module containing an instance of that global variable (unless the modification introduces a definition of the global variable and thereby changes its category). In addition, a category-1 global variable is *k→k safe*; not even a change to a kernel module can affect a kernel module containing an instance of that global variable. A category-2 global variable is *minimally k→k safe*; there is just one kernel module that contains a definition of that global variable, and only a change to a definition in that kernel module can affect another kernel module containing an instance of the global variable. Finally, a category-3 global variable is *k→k unsafe*.

Category-4 and category-5 global variables are *nonk→k unsafe*. That is, a modification to a nonkernel module can affect a kernel module (unless the modification removes all definitions of that global variable in nonkernel modules and thereby changes its category). Category-4 global variables are *k→k safe*, whereas category-5 global variables are *k→k unsafe*.

The distribution of the definitions and uses of the Linux global variables by category appears in Table X. As stated in the previous paragraph, global variables in categories 1, 2, and 3 are all nonk→k safe. They constitute about 55% of all global variables.

Table X: Distribution of definitions and uses of Linux global variables by category.

Category	Percentage of all global variables	Percentage of instances of global variables in kernel modules	Percentage of instances of global variables in nonkernel modules
1	23%	13%	8%
2	28%	39%	37%
3	4%	5%	3%
4	24%	14%	2%
5	20%	29%	50%

However, from the viewpoint of maintenance, what is important is not the number of unsafe global variables, but rather the number of *instances* of global variables and the number of *instances* of unsafe definitions of global variables. First, if a change is made to a global variable, it has to be consistently made to every instance of that global variable. Thus, the total number of instances of global variables (15,710 in the version of Linux we examined here, as stated in Table I) is important. Second, every unsafe definition of a global variable constitutes a potential source of vulnerability from the viewpoint of maintenance of the Linux kernel. From Table V, we see that there are 36 instances of definitions of category-2 global variables, 25 instances of definitions of category-3 global variables, and 180 instances of definitions of category-5 global variables in kernel modules. Furthermore, there are 1,772 instances of definitions of category-4 and category-5 global variables in nonkernel modules. That is, there are 2,013 instances of definitions of global variables that could affect a kernel module if a modification were made to the module containing that global variable.

In addition to the large total number of instances of global variables, we believe the fact that that there are many instances of unsafe definitions of global variables represents a long-term risk factor for Linux.

8. CONCLUSIONS AND FUTURE WORK

This paper presents two ideas. First, it defines a way to classify instances of common (global) coupling within software product lines. The classification is based on the definition-use characteristics of the variable references between kernel and nonkernel modules. Second, the paper presents results from a case study of applying this classification to Linux. Instances of common coupling were classified into five categories and analyzed as being "safe" or "unsafe" for the kernel modules.

Version 2.4.20 of Linux for Intel-based computers contains 99 global variables. We determined whether each of the 15,710 instances of those global variables is a definition or a use of that global variable. We found 2,013 unsafe definitions of global variables, that is, definitions of global variables that could affect a kernel module if a modification involving that global variable were made to the module in which the global variable is defined.

In a previous paper [7], we concluded that the large number of instances of common coupling within Linux has the potential to lead to problems with maintaining the Linux kernel in the long term. The large number of unsafe definitions of global variables reported in this paper reinforces our concern that, unless Linux is modified to contain less global coupling, the future prognosis of the maintainability of the Linux kernel will be unfavorable.

We hope that this paper will contribute to the field in three ways. First, it is hoped that these results can influence developers of Linux to reduce the amount of common coupling, particularly of category-4 and category-5 global variables. Second, it is hoped that this method of measuring a kernel-based system can be applied to other software systems that are developed using software product lines. Finally, the categorization of common coupling should be used as

a guide by developers of product line software. Categories 4 and 5 are the riskiest types of common coupling and should always be avoided.

ACKNOWLEDGMENTS

This work was sponsored in part by the National Science Foundation under grant number CCR–0097056. We would like to thank Hassan Gomaa for helpful suggestions regarding software product lines.

REFERENCES

[1] P. Clements, "A Framework for Software Product Line Practice—Version 4.1," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 2003. Available at http://www.sei.cmu.edu/plp/framework.html.

[2] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns,* Addison-Wesley, Reading, MA, 2002.

[3] H. Gomaa, "Modeling Software Product Lines with UML," *Proc. 3rd Int'l Workshop on Software Product Lines*, pp. 27–31, May 2001.

[4] L. C. Briand, J. Daly, V. Porter, and J. Wüst, "A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems," *Proc. 5th Int'l Software Metrics Symposium*, pp. 246–257, Nov.1998.

[5] D. A. Troy and S. H. Zweben, "Measuring the Quality of Structured Designs," *J. Systems and Software*, vol. 2, pp. 112–120, 1981.

[6] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and J. Offutt, "Quality Impacts of Clandestine Common Coupling," *Software Quality Journal*, vol. 11, pp. 211–218, 2003 (to appear).

[7] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt, "Maintainability of the Linux Kernel," *IEE Proceedings—Software,* vol. 149, no. 2, pp. 18–23, 2002.

[8] S. R. Schach and J. Offutt, "On the Nonmaintainability of Open-Source Software," *Proc. 2nd Workshop on Open Source Software Eng.*, pp. 47–49, May 2002.

[9] J. Offutt, "Open-source Software: More or Less Secure and Reliable?" Panel at *Int'l Symposium on Software Reliability Eng. (ISSRE '02)*, Nov. 2002.

[10] D. Cooke, J. Urban, and S. Hamilton, "Unix and Beyond: An Interview with Ken Thompson," *IEEE Computer*, vol. 32, no. 5, pp. 58–64, 1999.

[11] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems J.,* vol. 13, no. 2, pp. 115–139, 1974.

[12] M. Page-Jones, *The Practical Guide to Structured Systems Design,* Yourdon Press, New York, 1980.

[13] J. Offutt, M. J. Harrold, and P. Kolte, "A Software Metric System for Module Coupling," *J. Syst. and Software*, vol 20, no. 3, pp. 295–308, 1993.

[14] S. Wang, S. R. Schach, and G. Z. Heller, "A Case Study in Repeated Maintenance," *J. Software Maintenance and Evolution: Research and Practice,* vol. 13, no. 2, pp. 127–141, 2001.

[15] F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM*, vol. 19, no. 3, pp. 137–146, 1976.