Common coupling and pointer variables, with application to a Linux case study

**Stephen R. Schach · Tokunbo O. S. Adeshiyan ·
Daniel Balasubramanian · Gabor Madl ·
Esteban P. Osses · Sameer Singh ·
Karlkim Suwanmongkol · Minhui Xie ·
Dror G. Feitelson**

Abstract   Both common coupling and pointer variables can exert a deleterious effect on the quality of software. The situation is exacerbated when global variables are assigned to pointer variables, that is, when an alias to a global variable is created. When this occurs, the number of global variables increases, and it becomes considerably harder to compute quality metrics correctly. However, unless aliasing is taken into account, variables may incorrectly appear to be unreferenced (neither defined nor used), or to be used without being defined. These ideas are illustrated by means of a case study of common coupling in the Linux kernel.

1 Introduction

The goal of software engineering is to produce high-quality maintainable software. But there is little agreement regarding how quality and maintainability should be measured, and whether they can be measured directly. Over the years, various indirect measures have therefore been proposed. The degree of common coupling is one of them: Significant common coupling should be avoided, so low levels of common coupling are taken to indicate high quality and maintainability (Stevens et al., 1974). Such metrics are especially useful for the comparison of contending software development practices, such as open source vs. closed source.

   Common coupling refers to the use of global variables. Using global variables is bad practice because it violates the principles of encapsulation, information hiding, and abstraction

---

D. G. Feitelson: On sabbatical leave from Hebrew University.

S. R. Schach (✉) · T. O. S. Adeshiyan · D. Balasubramanian · G. Madl · E. P. Osses · S. Singh ·
K. Suwanmongkol · M. Xie · D. G. Feitelson
Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN
37235
e-mail: srs@vuse.vanderbilt.edu

(Myers, 1974; Yourdon and Constantine, 1979). A global variable is volatile, in the sense that its value may be changed in unpredictable ways, due to side effects of called functions.

Using global variables is bad practice; allowing pointer variables to point to global variables is even worse. When global variables are used directly, it is relatively straightforward to find all instances of these global variables and check their effect. But with pointer variables, a global variable may have several aliases. This makes it impractical to track the possible interactions among different modules, and increases the risk of undesirable effects.

We demonstrate the problems that stem from using pointers to global variables by means of a case study of common coupling in the Linux kernel.

The remainder of this paper is organized as follows: In Section 2 we discuss coupling issues, especially global variables and common coupling. The effects of pointer variables on common coupling are described in Section 3. The Linux case study is presented in Section 4, and the results of the case study in Section 5. Our conclusions appear in Section 6.

2 Coupling of software modules

A successful software project is one that meets its specifications within predefined budget and time constraints. This criterion for success is applicable to traditional closed-source software development, and has been measured for many thousands of projects. In fact, such measurements are the basis for the claim that the software industry is in a crisis; studies routinely show that the majority of projects fail to meet their targets (Jones, 1995; Johnson et al., 2001).

Regrettably, this straightforward metric cannot be applied to open-source software projects, because they typically have no detailed specifications, no budget, and no deadlines. Therefore, indirect metrics have to be found. Given the availability of the source code, it is natural to consider metrics that are based on the code itself, that is, metrics for code quality. These have the additional appeal of being quantitative, objective, and amenable to mechanized evaluation.

One such metric is the degree of coupling found in the code. Coupling between software modules measures the degree to which they are dependent on each other. One of the basic tenets of software engineering is that modules should be only weakly coupled together, because this promotes easier maintenance and reuse (Stevens et al., 1974; Myers, 1974; Offutt et al., 1993); contrariwise, strong coupling makes modules harder to understand and increases the propensity for errors (Binkley and Schach, 1998; Rilling and Klemola, 2003).

There are many different types of coupling that can occur between software modules (Offutt et al., 1993). Coupling means that one module depends on the other, typically in the form of using data that are produced by the other module. The distinctions are based on whether passed data are also used for control or not, and whether they are passed uni-directionally or bi-directionally.

Some form of coupling is obviously needed in order to allow the modules to work together as parts of a single application. But not all forms of coupling are equal. In 1974, Stevens, Myers, and Constantine published an ordinal scale of coupling (Stevens et al., 1974). The second worst form of coupling was what they called "common coupling"; the term refers to the use of global (shared) variables, harking back to the COMMON keyword from FORTRAN. It is widely agreed that common coupling should be avoided wherever possible.

Using global variables is bad practice mainly because global variables allow for side effects. Consider a module that uses a global variable g, and calls a function $f()$ from another module. When the function returns, the value of g may have changed. Moreover,

future changes to `f()` may cause new and unexpected behavior of `g`. Consequently, the programmer cannot rely on `g` remaining consistent, and needs to handle it with extreme care. Another reason that common coupling is considered bad practice is that it is susceptible to clandestine increase, where the coupling of a given module increases without the module itself being modified in any way, just because other modules have been modified (Schach et al., 2003).

3 The effect of pointer variables on common coupling

Programming languages such as C allow for pointer variables that point to other variables. This mechanism can be used to create aliases; a given variable can be accessed using its original name, and also through pointer variables that point to it.

Although pointer variables have some important applications in dynamic data structures, their indiscriminate use causes many problems. Because these pointers are variables, they can be assigned at runtime. Accordingly, a pointer may point to different variables at different times in the execution of the program. This makes it extremely hard, or even impossible, to perform a static analysis of the behavior of the program.

One alternative is to utilize conservative approaches. For example, all modules that access a global data structure in some way will be considered to be common coupled, even if there are actually subsets of modules that do not really depend on each other. For example, a global data structure may consist of two fields, `x1` and `x2`. One set of modules may access only `x1` and another set may access only `x2`. In such a case, claiming that the modules of the two sets are common coupled is inaccurate.

In other words, in order to determine the extent of common coupling (and, hence, measure program quality indirectly), we need to be able to identify every instance of a global variable. The presence of pointer variables can make such a determination difficult or even impossible.

We now illustrate these ideas with a case study.

4 Case study: The Linux kernel

Tabulating the degree of common coupling has been used to assess the quality of the code in the Linux operating system kernel, and how it changes with time. The original study by Schach et al. found that the number of instances of common coupling grows exponentially with version number (Schach et al., 2002). A follow-up by Yu et al. (2004) found that much of the common coupling was of an especially bad type that coupled kernel modules to non-kernel modules; this "category-5 global coupling" is described in Section 5.4.

A deficiency of these studies is that they were based on a lexical analysis of the Linux source code. In other words, they identified references to global variables only if the same name was used. However, the Linux kernel is rife with instances of references to global variables using pointer variables. To see how presence of these aliases affected the accuracy of these studies, consider the process descriptors in Linux.

The most heavily used global variables in Linux are the process descriptors. There is a process descriptor for each process in the system. It resides in an 8-KB block of memory that also holds the kernel stack of the process. This memory area is allocated dynamically when the process is created. Accordingly, there is no pre-defined array of process descriptors, as there was in early versions of Unix.
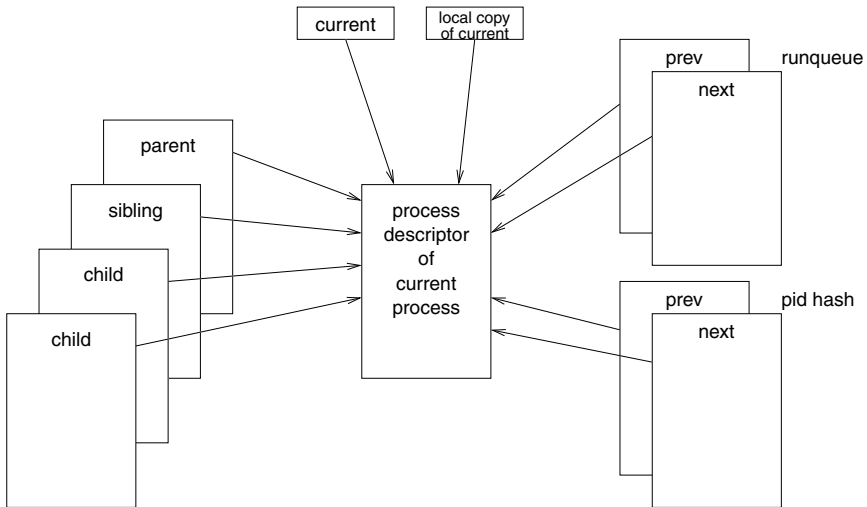
Fig. 1 Pointers that can be used to access a process descriptor in Linux

In Linux, a process descriptor is a structure of type `task_struct`, which has 105 fields.[1] Process descriptors are most commonly accessed via global pointer variable `current`, which points to the currently running process. (In reality, `current` is implemented as a mask on the stack pointer register, based on the fact that the kernel stack and process descriptor are co-located in the same memory block.) Of the 105 fields, 25 are pointers to various data structures; of these, 9 are pointers to other instances of type `task_struct`. They are used to link the process descriptors into three separate data structures: the run-queue (or some other list of processes), the pid (process identifier) hash table, and the tree of process family relationships (connecting processes to their parent, siblings, and children). The run-queue links, in particular, are used by the scheduler to traverse all the runnable processes in the system and select one for execution. To complicate matters further, there are at least 117 places in the code where `current` is copied to a local pointer that is then used to access the current process descriptor. In short, there are many different ways to access a process descriptor, all using pointers (Fig. 1). The earlier works on common coupling in Linux (e.g., Yu et al., 2004) considered the use of only `current` itself.

A similar situation occurs for data structures that are pointed to by the process descriptor. Sixteen of the fields of `task_struct` are pointers to other structures, including ones that describe the process's memory layout and open files; some subfields are also pointers to other structures. All these are often accessed via two or more levels of indirection using `current`. But there are at least 148 cases where a local copy of a pointer is made, affecting the access to 28 different subfields (of a total of 249 subfields that are defined). There are also at least 24 cases of aliases to aliases (that is, a local pointer to `current` or to a field is copied to another local pointer). Again, previous studies counted only the accesses using `current`, and missed those that use a local copy.

---

[1] This and other references to the Linux source code refer to kernel version 2.4.20, in order to be able to draw comparisons with previous studies, especially (Yu et al., 2004).

Table 1 Fields of
`task_struct` referenced via
`current`

Referenced via `current`	89 Fields
Referenced only via aliasing	1 Fields
Never referenced	15 Fields
Total fields	105 Fields

5 Results

We investigated the common coupling induced by `current`. First, we looked solely at the fields of `task_struct`.[2] Then we also considered the fields of the other types of structures pointed to by fields of `task_struct`. In each case, we considered the situation with and without aliasing.

Our results were obtained as follows: First, we identified all 8384 instances of `current` in the Linux source code. Second, each instance of `current` was examined independently by two researchers and analyzed as described in the following subsections. Third, the few discrepancies were easily resolved by the researchers concerned.

The reason why we decided to analyze the code manually, rather than use an automated tool, was that we could not find a tool with the required fidelity. Some of the uses of `current` in Linux are rather subtle, because it is used both as a pointer and as an identifier of the current process. The problem of determining what each reference means is of course exacerbated when aliases are concerned. In addition, we determined that nine researchers, each working for just two hours a week, could complete the task in several months; building a tool would take considerably longer.

5.1 Analysis of fields of `task_struct`

In this subsection, we consider the fields of `task_struct` referenced using `current`. As shown in Table 1, when we consider just `current` itself and the fields to which it points, 89 of the 105 fields of `task_struct` are referenced. An example of such a reference (to field `processor`) is

```
current−>processor = 0;
```

If we also consider all aliases for `current`, a single additional field is referenced. These are the statements in question:

```
struct task_struct *tsk = current;
tsk−>vfork_done = NULL;
```

Clearly, aliasing has a negligible impact on the fields of `task_struct` accessed via `current`.

5.2 Wider analysis

We now consider all fields referenced, directly or indirectly, by `current`. For example, consider the statement

```
current−>fs−>altrootmnt = mnt;
```

---

[2] In what follows, for brevity we use the informal terminology "fields of `task_struct`" rather than the more precise "fields of instances of type `task_struct`."

Table 2 Subfields of `current` referenced via `current` or via an alias

Referenced via `current`	280 Subfields
Referenced only via aliasing	58 Subfields
Total referenced fields	338 Subfields

Here, pointer variable `current` points to pointer field `fs` in `task_struct`. Pointer `fs` is a pointer to a structure of type `fs_struct`. Field `altrootmnt` is a field of a structure of type `fs_struct` and is set equal to `mnt`. In this case, the fields referenced are `fs` and `altrootmnt`; we refer to such fields collectively as *subfields* of `current`.

Another example is

```
sig = fpu_emulator_cop1Handler(0, regs, &current
    ->thread.fpu.soft);
```

In this example, `current` points to a struct of type `task_struct`, which contains a field `thread`, a struct of type `thread_struct`. The latter has a field `fpu` that is a struct of type `fp_status`; this struct has a field `soft`. Here the subfields of `current` are `thread`, `fpu`, and `soft`.

Table 2 shows the results when all subfields referenced using `current` and its aliases are considered. The results in Table 2 incorporate those of Table 1. Now there are 58 fields that are accessed only via aliases. In other words, over 17 percent of the 338 fields would not be taken into account if aliasing were ignored.

5.3 Unreferenced and undefined global variables

Every instance of a variable in a program is either a *definition* of that variable (that is, a change made to the value of that variable) or a *use* of that variable (that is, a utilization of the current value of that variable).

An *unreferenced* field is one that is neither defined nor used in the statements we examined. As shown in Tables 3 and 2, following aliases exposes 58 additional subfields that are not seen when references using only `current` are considered. They are therefore classified as unreferenced when aliasing is not considered. But even with aliasing there are six unreferenced fields. These six fields fall into the following categories:

1. "Fields" that were found by mistake, due to outdated comments that mention fields that no longer exist. There were four such fields.
2. A field (`thread.usp`) that exists, is mentioned in a comment, but is not referenced from `current`. However, it could be referenced via some other mechanism of which we are unaware.
3. A field (`thread.esp`) that occurs in only an assembler statement. We have set this instance aside until we have done the necessary research into the nature of common coupling between a second-generation language (assembler) and a third-generation language (C).

Table 3 The effect of following aliases

	Unreferenced	Undefined	Defined	Total fields referenced
Without aliasing	64	78	202	278
With aliasing	6	89	249	338

An *undefined* field is one that is used but not defined in the statements we examined. As previously mentioned, without aliasing there were 64 unreferenced fields. Aliasing caused 22 of them to become undefined, and 36 to become defined. Also, without aliasing there were 78 undefined fields. Aliasing caused 11 of them to become defined. So, as shown in Table 3, the number of unreferenced fields dropped from 64 to 6, and the number of undefined fields increased from 78 to 89 ($= 78 + 22 - 11$).

5.4 Categorization of common coupling

Yu et al. (2004) categorized common coupling in kernel-based software. They set up five categories of common coupling on the basis of the roles that the global variables play. As explained in Section 5.3, every occurrence of a variable in the code can be classified as either a *definition* or a *use* of that variable. Yu et al. (2004) applied this classification to occurrences of global variables in the code, and then categorized the global variables as follows:

Category 1: Global variables that are defined in kernel modules but not used in any kernel module. Global variables of this kind can be interpreted as "kernel outputs"; in object-oriented terminology, they serve as "get" methods (accessors) for certain internal kernel attribute. As such, their use is reasonable.

Category 2: Global variables that are defined in a single kernel module, and used in other kernel (and non-kernel) modules. Such a global variable can be interpreted as a "get" within the kernel in addition to being a "get" used by external modules. This is less desirable than category 1, but is still reasonable.

Category 3: Global variables that are defined in several different kernel modules. This causes the different kernel modules to depend on one other, and is therefore an undesirable usage mode.

Category 4: Global variables that are defined in non-kernel modules and used in kernel modules. Although this creates a dependency of the kernel on non-kernel code, it may be necessary as an input mode; in other words, this is similar to a "set" method (mutator) of a kernel attribute. Although this is distinctly undesirable, it may be hard to avoid.

Category 5: Global variables that are defined in both kernel and non-kernel modules, and used in kernel modules. This is an extreme form of coupling between kernel and non-kernel code, and is highly undesirable.

Subsequently, Feitelson et al. observed that many of the subfields of `current` are global variables that are neither defined nor used in the kernel. Accordingly, they added a sixth category (Feitelson et al., 2007):

Category 0: Global variables that are neither defined nor used in the kernel.

In other words, there exists a six-way categorization of common coupling in kernel-based software on the basis of definition–use analysis. Furthermore, the higher the category number, the more undesirable is the resultant common coupling.

Without aliasing, 202 fields of `current` are defined, as shown in Table 3. The categorization of these global variables is shown in Table 4. When aliasing is taken into account, the number of defined fields increases to 249; their distribution is shown in Table 5.

We observe first that the number of fields increases by nearly 25 percent when aliasing is taken into account, and second that the distribution of global variables among the categories changes. In particular, some fields moved "up" to higher categories, which indicates a stronger form of coupling, which is undesirable. Notably, in the initial categorization there were no

Table 4 Results of categorizing subfields of `current` without any aliasing

Category 0	118 subfields	(58.4%)
Category 1	5 subfields	(2.5%)
Category 2	27 subfields	(13.4%)
Category 3	0 subfields	(0.0%)
Category 4	7 subfields	(3.5%)
Category 5	45 subfields	(22.3%)
Total	202 subfields	(100.0%)

Table 5 Results of categorizing subfields of `current` incorporating aliasing

Category 0	154 subfields	(61.8%)
Category 1	5 subfields	(2.0%)
Category 2	27 subfields	(10.8%)
Category 3	3 subfields	(1.2%)
Category 4	7 subfields	(2.8%)
Category 5	53 subfields	(21.3%)
Total	249 subfields	(100.0%)

Table 6 Re-categorization of fields of `current` as a result of considering references made via aliases

From	UR	UD	0	1	2	3	4	5
Unreferenced	6	22	33	0	3	0	0	0
Undefined	0	67	9	0	0	1	0	1
Category 0	0	0	112	0	0	0	2	4
Category 1	0	0	0	5	0	0	0	0
Category 2	0	0	0	0	24	2	0	1
Category 3	0	0	0	0	0	0	0	0
Category 4	0	0	0	0	0	0	5	2
Category 5	0	0	0	0	0	0	0	45

(The header "To" spans columns UR through 5.)

fields in category 3, but with aliasing three such fields were found. More significantly, the number of fields in highly undesirable category 5 increased from 45 to 53, an increase of over 17 percent. These changes are shown in more detail in Table 6.

5.5 References to global variables

We also counted the number of references (that is, individual occurrences in the code) to each of the fields and subfields accessible from `current`. Comparing Tables 7 and 8, we see that not only does the number of references increase when aliasing is taken into account, the percentage of references in the different categories changes as well. And again, at least for the kernel modules, the percentage of references made to fields belonging to undesirable categories increased. In non-kernel modules, the percentage of references made to such fields decreased slightly, whereas their absolute numbers increased significantly. This was due to the introduction of many new subfields in category 0, which were found only with aliasing.

Table 7 Results of analyzing individual occurrences of fields of `current` without aliasing

	Kernel	Non-kernel
Category 0	0 (0.0%)	1410 (23.3%)
Category 1	6 (1.4%)	18 (0.3%)
Category 2	96 (21.8%)	191 (3.2%)
Category 3	0 (0.0%)	0 (0.0%)
Category 4	18 (4.1%)	131 (2.2%)
Category 5	320 (72.7%)	4310 (71.1%)
Total	440 (100.0%)	6060 (100.0%)

Table 8 Results of analyzing individual occurrences of fields of `current` with aliasing

	Kernel	Non-kernel
Category 0	0 (0.0%)	1894 (25.3%)
Category 1	6 (1.0%)	18 (0.2%)
Category 2	96 (15.2%)	197 (2.6%)
Category 3	13 (2.1%)	10 (0.1%)
Category 4	16 (2.5%)	166 (2.2%)
Category 5	500 (79.2%)	5214 (69.5%)
Total	631 (100.0%)	7499 (100.0%)

5.6 Correlation with and without aliasing

Finally, we considered the effect of aliasing on each of the 839 modules. First, we looked at the extent of aliasing. This is shown in Fig. 2, a scatter plot of the total number of direct or indirect references to fields of `current` with and without considering aliasing.

Each dot corresponds to a single Linux file (module). The 717 files that lie on the diagonal are not affected by aliasing, whereas the 122 files above the diagonal use aliasing. In 47 of those files, the total number of fields referenced directly or indirectly increases when aliasing

Fig. 2 References to fields of `current` in 839 Linux files with aliasing and without aliasing
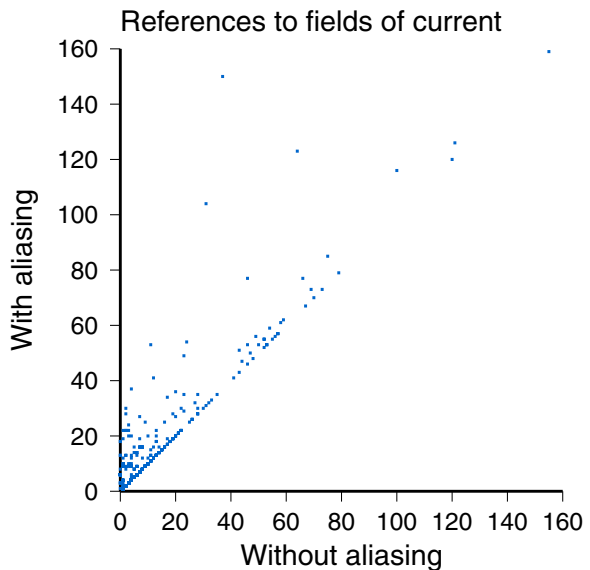
Table 9 Descriptive statistics for all 839 files

	Fields referenced		References to fields	
	Without aliasing	With aliasing	Without aliasing	With aliasing
Maximum	27	30	298	376
Mean	2.80	3.08	8.07	9.83
Median	1	2	3	3
Minimum	0	1	0	1

Table 10 Descriptive statistics for the 122 files that use aliasing

	Fields referenced		References to fields	
	Without aliasing	With aliasing	Without aliasing	With aliasing
Maximum	25	30	298	376
Mean	5.56	7.44	18.64	30.13
Median	3	6	6	16
Minimum	0	1	0	1

is taken into account; in 67 files, the total number of direct or indirect references to fields increases; in 20 files, fields of `current` are referenced only as a consequence of aliasing.

Most files have only a few references, but a few files have many. In some cases, most or even all the references are due to aliases. These results are reflected in Table 9, which shows descriptive statistics for all 839 files, and in Table 10, which shows the corresponding statistics for the 122 files that use aliasing. In each case, the range is large, the mean is small, and the median is smaller than the mean, sometimes by a factor of 2 or 3. In other words, the use of aliases is unevenly distributed.

Not only is aliasing unevenly distributed among files (modules), it is highly unevenly distributed among directories. There is a heavy use of aliases in the `kernel` subdirectory (the "kernel" in Schach et al. (2002) and Yu et al. (2004)), some use in `mm` (memory management), and some use in `arch` (architecture-specific code). In particular, many of the files that reference fields of current only via an alias are of the form `arch/*/kernel/semaphore.c`. There are exceedingly few uses of aliases in drivers, `#include` files, the file system (`fs`), and the networking code.

Then we computed the correlation between common coupling in the absence of aliasing and common coupling in the presence of aliasing. More precisely, because the data are so unevenly distributed and because of the large number of identical values, we used the nonparametric Spearman rank correlation coefficient, corrected for ties (Hollander and Wolfe, 1973).

First we considered all 839 files, and then we looked at the 122 files that use aliasing. Our results appear Table 11. We conclude that in general those files that are highly common coupled when aliasing is not considered are also highly common coupled when aliasing is

Table 11 Correlation between common coupling in the absence of aliasing and in the presence of aliasing

	Fields referenced		References to fields	
	Rank correlation	P-value	Rank correlation	P-value
All 839 files	0.9172	< 0.0001	0.8785	< 0.0001
122 files that use aliasing	0.8409	< 0.0001	0.8339	< 0.0001

considered. We deduce that the increases in common coupling reflected in Table 6 when aliasing is considered are generally associated with those files that have high common coupling when aliasing is not considered.

5.7 Implications for earlier research

As outlined at the start of Section 4, the case study in this paper extends and corrects the results of the original case study by Schach et al. (2002) and the follow-up case study by Yu et al. (2004). However, the results of this paper do not change the conclusions of Schach et al. (2002) and Yu et al. (2004); on the contrary, they strengthen them.

Both Schach et al. and Yu et al. counted only common coupling induced by the global variables themselves; they overlooked common coupling induced by aliases of those global variables. Had they included this additional common coupling, the number of instances of common coupling would have been even larger than what they reported; both papers focused on directory `kernel`, where the use of aliases is the heaviest. Accordingly, the results of this paper do not invalidate (Schach et al., 2002; Yu et al., 2004), but rather reinforce their conclusion that, in the long term, Linux will become nonmaintainable unless Linux is refactored to greatly reduce the amount of common coupling.

5.8 Threats to the validity of the Linux case study

In this case study, we have considered all fields referenced directly or indirectly by pointer variable `current`. We have also considered aliases of `current` and of pointer variables referenced directly or indirectly by `current`. However, we have *not* considered all of the many aliases in Linux. A consequence is that there may be many more global variables than those that we have identified.

We have identified a number of fields that apparently either are never referenced, or are never defined. Some fields are initialized by copying. That is, sometimes a structure is created as a copy of a preinitialized "standard" version of the structure. The structure as a whole is copied, thereby defining the relevant fields. However, the only definition mechanism we have considered is assignment. Accordingly, we may have miscategorized some global variables as unreferenced.

6 Conclusions

It is widely agreed that common coupling, that is, the use of global variables, should be minimized, and that pointer variables need to be handled with care. In this paper we have demonstrated four examples of what can happen when pointer variables and common coupling interact.

First, the creation of an alias for a global variable means that another global variable has been created.[3] That is, aliasing of global variables is antithetical to the goal of minimizing the number of global variables in a program. This holds irrespective of whether the global variable in question is a pointer variable (as is the case in this paper). However, the severity of the situation is aggravated when the global variable in question is a pointer. When a pointer to a structure is a global variable, then all the fields of that structure become global variables,

---

[3] More precisely, what has been created is a reference to an existing global variable. From the viewpoint of a programmer, however, the effect is as if a new global variable has been created.

too. Creating an alias to the pointer results in even more global variables. That is, aliasing can create global variables that do not exist in the absence of aliasing.

Second, if fields of the structure in question are themselves pointer variables, then the items to which they point are also global variables, and this may be taken to any level. Again, if there is aliasing as well, then the number of global variables can be further increased.

Third, the presence of aliasing means that considerable additional work may have to be done in order to compute quality metrics correctly. These metrics include the number of global variables and their categorization in terms of definition-use analysis.

Fourth, without taking aliasing in account, variables may incorrectly appear to be unreferenced (neither defined nor used), or to be used without being defined.

In conclusion, the combination of global variables and pointer variables is highly undesirable. In those situations where global variables are essentially unavoidable, pointer variables should be eschewed.

References

Binkley, A.B., Schach, S.R. 1998. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In: 20th Intl Conf. Softw. Eng. pp. 452–455.

Feitelson, D.G., Adeshiyan, T.O.S., Balasubramanian, D., Etsion, Y., Madl, G., Osses, E.P., Singh, S., Suwanmongkol, K., Xie, M., Schach, S.R. 2007. Fine-grain analysis of common coupling and its application to a Linux case study. J Syst. Softw., to appear.

Hollander, M., Wolfe, D.A. 1973. Nonparametric Statistical Methods. John Wiley & Sons.

Johnson, J., Boucher, K.D., Connors, K., Robinson, J. 2001. Project management: the criteria for success. Softwaremag.com, URL www. softwaremag.com/archive/2001feb/CollaborativeMgt.html.

Jones, C. 1995. Patterns of Software System Failure and Success. Intl Thomson Computer Pr (Sd).

Myers, G. 1974. Reliable Software Through Composite Design. Mason and Lipscomb Publishers.

Offutt, A.J., Harrold, M.J., Kolte, P. 1993. A software metric system for module coupling. J Syst. Softw. **20**(3):295–308.

Rilling, J., Klemola, T. 2003. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In: 11th IEEE Intl. Workshop Program Comprehension, pp. 115–124.

Schach, S.R. 2007. Object-Oriented and Classical Software Engineering, 7th edition. McGraw-Hill.

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

Schach, S.R., Jin, B., Wright, D.R., Heller, G.Z., Offutt, J. 2003. Quality impacts of clandestine common coupling. Softw Quality J **11**:211–218.

Stevens, W.P., Myers, G.J., Constantine, L.L. 1974. Structured design. IBM Syst J **13**(2):115–139.

Yourdon, E., Constantine, L.L. 1979. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Englewood Cliffs, NJ: Prentice Hall.

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

Stephen R. Schach is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University, Nashville, Tennessee. Steve is the author of over 125 refereed research papers.

He has written eleven software engineering textbooks, including Object-Oriented and Classical Software Engineering, Seventh Edition (McGraw-Hill, 2007). He consults internationally on software engineering topics. Steve's research interests are in empirical software engineering and open-source software engineering. He obtained his Ph.D. from the University of Cape Town.

Tokunbo O. S. Adeshiyan is currently a software engineer at IBM in Research Triangle Park, NC. His technical interests include web-based applications, server virtualization, and software engineering. Prior to employment with IBM, Toks received his M.S. in Computer Science from Vanderbilt University in Nashville, TN.

Daniel Balasubramanian is a Ph.D. student at Vanderbilt University. His research interests include software tools for embedded system design and software maintainability.

Gabor Madl is a Ph.D. student and a graduate student researcher at the Center for Embedded Computer Systems at the University of California, Irvine. His research interests include the formal verification, optimization, component-based composition, and QoS management of distributed real-time embedded systems and system-on-chip designs. He received his M.S. in computer science from Vanderbilt University and in computer engineering from the Budapest University of Technology and Economics.

Esteban P. Osses is an Instructor in the Department of Informatics Engineering and Computer Science at University of Concepcion, Chile. He obtained his M.Sc. in Computer Science at Vanderbilt University in Nashville, Tennessee in 2005. His main research interest is in evolutionary computing, object-oriented software analysis and design, and collaborative environments.

Sameer Singh is a Graduate Student in the Department of Electrical Engineering and Computer Science at Vanderbilt University in Nashville, Tennessee. His current research interests lie in the application of conventional Artificial Intelligence and Machine Learning techniques to various Cognitive Architectures. He has also worked on various projects in the fields of open-source software engineering, robotics and bio-informatics.

Karlkim Suwanmongkol is a Ph.D. candidate in the Deaprtment of Electrical Engineering and Computer Science at Vanderbilt University in Nashville, Tennessee. His research interests include Model-based software development and maintenance, Model Driven Architecture (MDA), and software engineering.

Dror G. Feitelson received the B.Sc. degree in Mathematics, Physics, and Computer Science in 1985, and the M.Sc. and Ph.D. degrees in Computer Science in 1987 and 1991, all from The Hebrew University of Jerusalem, Israel, where he is now on the faculty of the School of Computer Science and Engineering. His main research interests are in experimental aspects of computer science, and in particular the scientific observation and study of complex computer-related artifacts, ranging from the workloads on parallel supercomputers to open-source software projects.