e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

The Evolution of Complexity in Apple Darwin: A Common Coupling Point of View

Liguo Yu*
*Computer Science and Informatics, Indiana University South Bend
ligyu@iusb.edu

Abstract

Common coupling increases the interdependencies between software modules. It should be avoided if possible. In previous work, we presented two types of categorization of common coupling, one is for single-kernel-based software, one is for multi-kernel-based-software. In this paper, we analyze the relationships between these two types of categorization and apply them to study the evolution of the complexity of Apple Darwin. The same conclusion about Darwin's evolution is drawn based on the two types of categorization of common coupling: From version XNU-517 to version XNU-792, Darwin has restructured to reduce the number of difficulty-inducing high category (level) global variables in order to reduce the system complexity. However, due to the definition-use dependencies, the complexity of Darwin induced by global variables has increased from version XNU-517 to version XNU-792.

1. Introduction

Coupling measures the degree of dependencies between two software modules [9, 6, 5]. Strong coupling indicates a high degree of dependencies while loose coupling indicates a low degree of dependencies. High degree of dependency makes the software modules difficult to maintain and reuse. For example, to identify the origin of a fault, the best practice is to separate modules and test each of them individually. Loose coupling with low degree of dependencies can make the fault isolation process easier, while strong coupling with high degree of dependencies will make this process tedious and time/effort consuming. Consider reuse, it is easier to reuse a module that has loose coupling and is weakly dependent on others than a module that has strong coupling and is tightly dependent on others. Therefore, from the viewpoint of maintenance and reuse, a good software system should have low coupling between modules.

The software couplings can be divided as data coupling (simple data are passed as parameters in a function call), stamp coupling (data structures are passed as parameters in a function call), external coupling (two modules access the same file/database), and common coupling (two modules access the same global variable), in which, common coupling is considered to be a strong form of coupling. That is, common coupling induces high degree of dependencies between software modules and accordingly makes software modules difficult to understand, maintain, and reuse [8, 7].

Software evolution is inevitable. On the one hand, software needs to continually satisfy customers' functional requirements and non-function requirements. On the other hand, software needs to promptly adapt to the changes of hardware and system environments. Therefore, with the evolution of a software system, new features and new modules to support new hardware, are continually added to the source code. Both the size of the product and the com-

plexity of the product are expected to increase as new versions are developed and released. At the same time, to make the system align with its original quality design, the code structure needs to be monitored and examined frequently, and restructuring should be taken as needed to reduce the system complexity in order to achieve high maintainability and reusability. One way to reduce the system complexity is to replace existing strong couplings or new strong couplings introduced in the evolution process with loose couplings.

In this paper, we use common coupling as a measure of the system complexity and study how it changes with the evolution of a software system. The study is performed on Apple Darwin, an open-source operating system. The objective of this study is to understand the changing patterns of software complexity under the dual effects of size increasing and code restructuring in the evolution process.

The remainder of the paper is organized as follows: Section 2 describes kernel-based software. Section 3 reviews the categorizations of common coupling. Section 4 presents the study of the evolution of Darwin. The conclusions and limitations appear in Section 5.

2. Kernel-Based Software

Many software products, such as operating systems and database systems, are called *kernel-based software* [2]. That is, the software system consists of architecture and/or platform independent kernel modules, together with specific architecture and/or platform dependent nonkernel modules [8, 1]. Software product line is another example of kernel-based system, in which, the core assets are considered as kernel modules, and custom assets are considered as nonkernel modules. Figure 1 depicts the production of kernel-based software: Each implementation/installation of kernel-based software involves the use of all kernel modules and optional nonkernel modules.

In previous work [17], we identified two types of kernel-based software, single-kernel-based

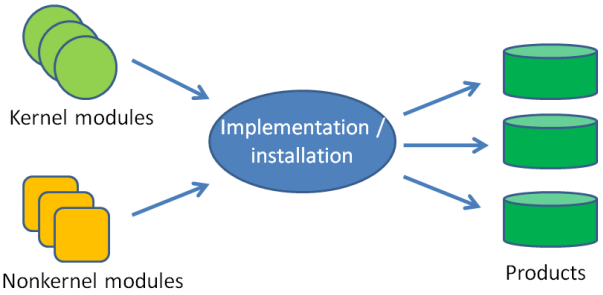

Figure 1. Depiction of the production of kernel-based software

software and multi-kernel-based software. In a kernel-based software system, if all the kernel modules are included in one component and the rest nonkernel modules are included in another component, we call it *single-kernel-based software*. Figure 2 shows the structure of a single-kernel-based software system, in which circles represent kernel modules, squares represent nonkernel modules, and rectangles represent components. In other words, in single-kernel-based systems, kernel modules and nonkernel modules are clearly separated into two components, kernel component and nonkernel component [17]. Examples of single-kernel-based systems are Linux and BSDs.

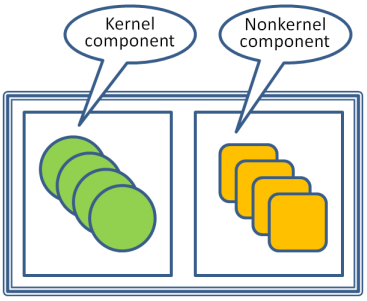

Figure 2. Depiction of single-kernel-based software

In a kernel-based software system, if the kernel modules are included in more than one component, we refer to that system as *multi-kernel-based software*. Figure 3 shows a multi-kernel-based software system, in which, kernel modules (represented with circles) and nonkernek modules (represented with squares) coexist in multiple components. These components that consist of both kernel modules and nonkernel modules are called *kernel-based components* and are represented with triple line rect-

angles. We use the term *outer component* to refer to the software component external to the kernel-based components. The outer component consists of no kernel modules and is represented with a dashed triple line rectangle. Examples of multi-kernel-based software are Apple Darwin and TrustedBSD SED Darwin [10].

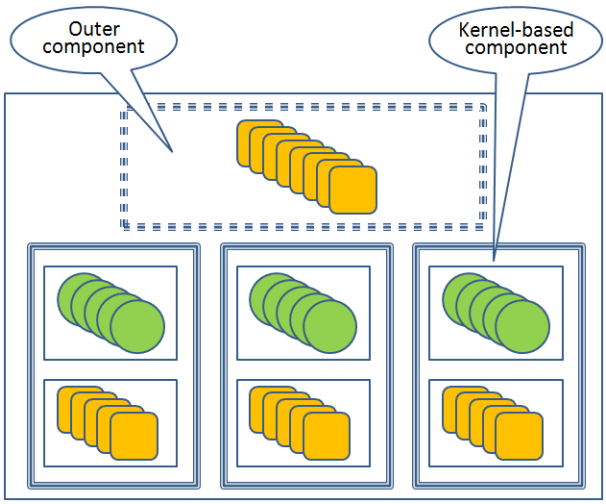

Figure 3. Depiction of multiple-kernel-based software

3. Categorizations of Common Coupling in Kernel-Based Software

In previous work, we presented two types of categorization of common coupling, one is within a single-kernel-based software system [15, 13, 16], and one is within a multiple-kernel-based software system [17]. These categorizations provide two approaches to measuring the maintenance effort and reuse effort of a kernel-based software system [14, 12]. These categorizations are reviewed here to provide the background knowledge about this research. Both of these categorizations are related with the definition-use analysis of global variables [15, 17].

3.1. Definition-use Analysis

The occurrence of a variable in a source code statement is related with one of the two tasks: reading the value of the variable or writing a value to the variable. Writing a value to a variable is called *definition* of a variable. The most common form of variable definition is an assignment statement, such as $\mathbf{x = 10}$. Reading the value of a variable is called *use* of a variable. The use of a variable is a statement that utilizes the value of the variable, such as **print(x)**. From the declaration of a variable to the destruction of that variable, each time the variable is invoked, it is either assigned a new value (a definition) or its present value is used (a use).

Common coupling induces dependencies between software modules through the definition-use of a global variable. For example, if module **M1** defines a global variable and module **M2** uses that global variable, we say that module **M2** is dependent on **M1** via common coupling. This dependency induced by definition-use of a global variable has effects on both software maintenance and reuse. Considering maintenance, if changes are made to module **M1**, attentions must be given to module **M2** to examine the effects of such changes and if necessary, corresponding changes should be made on **M2**. For reuse, if we want to reuse module **M2**, we must consider either reusing **M1** together with **M2** (because **M2** is dependent on **M1**), or modifying **M2** to remove its dependence on **M1**.

3.2. Categorization of Common Coupling in Single-Kernel-Based Software

In previous work, we divided global variables in single-kernel-based software into 5 categories, as shown below [15].

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

defined and used in one or more kernel modules.

In these five categories, high categories (Categories 4 and 5) global variables are considered worst for kernel maintenance and reuse, because they induce dependencies of kernel modules on nonkernel modules; Categories 2 and 3 global variables induce dependencies locally within the kernel and are accordingly considered better than Categories 4 and 5; Category-1 global variables do not affect kernel dependencies and are considered better than all others. For more discussions about these five-category global variables, the readers are referred to [15].

3.3. Categorization of Common Coupling in Multiple-Kernel-Based Software

In previous work [17], we divided global variables in multiple-kernel-based software into 6 levels. They are listed below.

- Level 0: A global variable is defined in kernel modules but not used in kernel modules.
- Level 1: A global variable is defined and used within the same kernel module but not defined in any other modules.
- Level 2: A global variable is used in kernel modules and is defined in nonkernel modules of the same kernel-based component.
- Level 3: A global variable is used in kernel modules of one kernel-based component and is defined in nonkernel modules of an outer component.
- Level 4: A global variable is used in kernel modules of one kernel-based component and is defined in nonkernel modules of another kernel-based component.
- Level 5: A global variable is used in kernel modules of one kernel-based component and is defined in kernel modules of another kernel-based component.

In these levels, a Level-0 global variable cannot affect the dependencies of kernel modules, because there is no use in kernel modules. Therefore, the presence of a Level-0 global variable will not cause difficulties in the maintenance and reuse of kernel modules. The definition and use of a Level-1 global variable are all within one kernel module and accordingly, this definition does not affect the dependency of this kernel module as well as other kernel modules.

In contrast, a kernel module that uses a Level-2 global variable depends on the nonkernel modules that define the global variable. This dependency is within the same kernel-based component and it does not affect other kernel-based components. High level (Levels 3 to 5) global variables might affect kernel maintenance and reuse. A Level-3 global variable induces dependencies of kernel modules on outer modules. Level-4 and Level-5 global variables are worst. They induce dependencies of kernel modules on modules of other kernel-based components. For more discussions about these six-level global variables, the readers are referred to [17].

3.4. Relations between the Two Types of Categorizations

In multiple-kernel-based systems, there are more than one kernel-based components. However, in some cases, we are interested in only one specific kernel-based component and consider the kernel modules within this component as kernels and all the rest modules (within or outside of this component) as nonkernels. Therefore, we can also consider this multi-kernel-based software system as a single-kernel-based software system. For example, in Figure 3, if we only consider the kernel modules within the first kernel-based component as kernels, the resulting system is single-kernel based, as shown in Figure 4.

Accordingly, the global variables categorized into six levels in multi-kernel-based software systems can be recategorized and mapped to the five categories in single-kernel-based software systems. The mapping and the relationships are shown in Table 1. Therefore, a global variable in multi-kernel-based software could be categorized using two schemes. In the remainder of this paper, we call the 5-category categorization *Categorization 1* and the 6-level categorization *Categorization 2*.

To study the effects of definition-use of a global variable on kernel dependencies, we utilize the following terminologies.

- **Dependency-inducing definition**: a definition of a global variable that can induce the dependency of kernel modules on other modules.
- **Non-dependency-inducing definition**: a definition of a global variable that cannot induce the dependency of kernel modules on other modules.
- **Safe dependency-inducing definition**: a definition of a global variable that induces the dependency of kernel modules on other kernel modules.
- **Unsafe dependency-inducing definition**: a definition of a global variable that induces the dependency of kernel modules on other nonkernel modules.

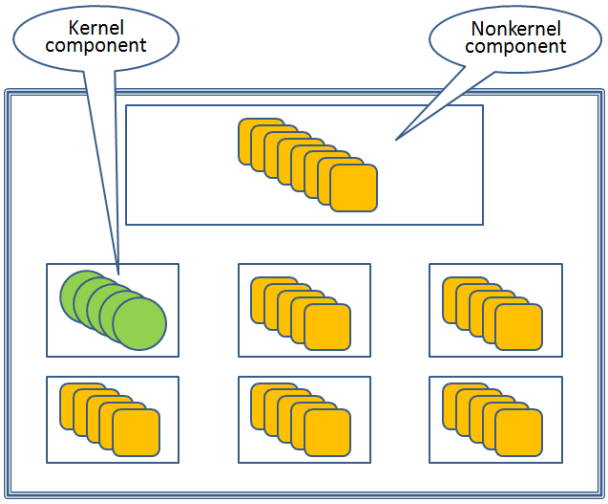

Figure 4. The single-kernel point of view of the system shown in Figure 3

Table 1. The mapping of 6-level categorization to 5-category categorization of global variables in multi-kernel based software

Level number (6-level categorization)	Category number (5-category categorization)
0	1
1	2, 3
2, 3, 4, 5	4, 5

In other words, if a definition of a global variable induces the dependency of kernel modules on other modules (either kernel or nonkernel), it is a dependency-inducing-definition. Otherwise, the definition is called a non-dependency-inducing definition. The definitions of global variables in Categories 2 to 5 (Levels 1 to 5) are dependency-inducing definitions. Definitions of global variables in Category 1 (Level 0) induce no dependencies of kernel modules and are accordingly non-dependency-inducing.

There are two types of dependency-inducing definitions, safe dependency-inducing definition and unsafe dependency-inducing definition. Safe dependency-inducing definitions are related with Categories 2, 3 and 5 (or Levels 1 to 5) global variables and these definitions occur in kernel modules. Unsafe dependency-inducing definitions are related with Categories 4 and 5 (or Levels 2 to 5) global variables and these definitions occur in nonkernel modules. Table 2 and Table 3 classify the definitions into different types. For example, a definition of Category-1 global variable in a nonkernel module is a non-dependency-inducing definition; a definition of a Level-2 global variable in a kernel module is a safe dependency-inducing definition; a definition of a Level-3 global variable in a nonkernel module is a unsafe dependency-inducing definition. The symbol "—" indicates there is no such definitions of a global variable in the corresponding category (level).

We remark here that the terminologies of safe/unsafe dependency-inducing definitions presented in this paper are different from the terminologies of safe/unsafe definitions cited in [15]. The unsafe definitions cited in [15] are actually the dependency-inducing definitions presented here and the safe definitions cited in [15] map to the non dependency-inducing definitions presented here.

4. The Evolution of Darwin

4.1. Overview

Darwin is Apple's open-source operating system for Macintosh computers. Figure 5 shows the architecture of Apple Darwin. It conceptually consists of three components: One kernel-based component (denoted as **osfmk**) that is reused (with modifications) from Mach [4], another kernel-based component (denoted as **bsd**) that

Table 2. The classification of definitions in single-kernel based software

Category	Definitions in kernel modules	Definitions in nonkernel modules
1	Non-dependency-inducing	Non-dependency-inducing
2	Safe dependency-inducing	–
3	Safe dependency-inducing	–
4	–	Unsafe dependency-inducing
5	Safe dependency-inducing	Unsafe dependency-inducing

Table 3. The classification of definitions in multiple-kernel based software

Level	Definitions in kernel modules	Definitions in nonkernel modules
0	Non-dependency-inducing	Non-dependency-inducing
1	Safe dependency-inducing	–
2	Safe dependency-inducing	Unsafe dependency-inducing
3	Safe dependency-inducing	Unsafe dependency-inducing
4	Safe dependency-inducing	Unsafe dependency-inducing
5	Safe dependency-inducing	Unsafe dependency-inducing

is reused (with modifications) from FreeBSD [3, 11], and the third component (denoted as outer) is a new written regular component. The structure of Darwin shown in Figure 5 indicates that it is a dual-kernel-based system. Darwin is written in C/C++, in the remainder of this paper, we use the term module to refer to a source code file written in C or C++ (**.c** file, **.cpp** file, or **.h** file).

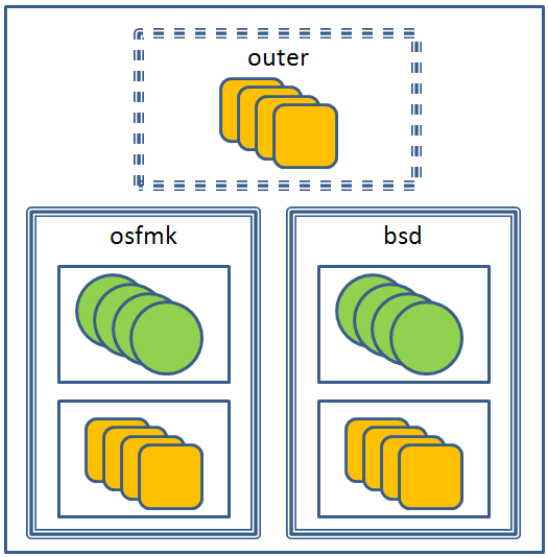

Figure 5. Architecture of Apple Darwin, a dual-kernel-based software system

To study the changes of kernel dependencies of Darwin with the evolution of the system complexity, we compared common coupling in two versions of Darwin, XNU-517 and XNU-792, which were released in November 2003 and April 2005, respectively. Figure 6 illustrates the evolution of Darwin from XNU-517 to XNU-792. It shows that the total number of modules increased about 5 percent and the total size measured in KLOC (Thousand Lines of Code) increased about 11 percent.

4.2. The Evolution of Complexity Induced by Common Coupling

In order to study common coupling in Darwin kernels, for each kernel module, we determined all the global variables and characterized them using two different schemes described in Section 3, *Categorization 1* and *Categorization 2*. Figure 7 and Figure 8 illustrate the evolution of the global variables in Darwin from the viewpoint of single-kernel-based software and multiple-kernel-based software respectively.

In both categorizations, we can see that the total number of global variables increased from version XNU-517 to version XNU-792. However, if we look at the most unfavorite global variables (Categories 4 and 5 in *Categorization 1* and Levels 2, 3, 4, and 5 in *Categorization 2*), the number decreased from 17 to 14 and from 23 to 19 for **osfmk** kernel and **bsd** kernel respectively. Therefore, we can conclude that, from version XNU-517 to version XNU-792, Darwin has restructured to reduce the number of high-level

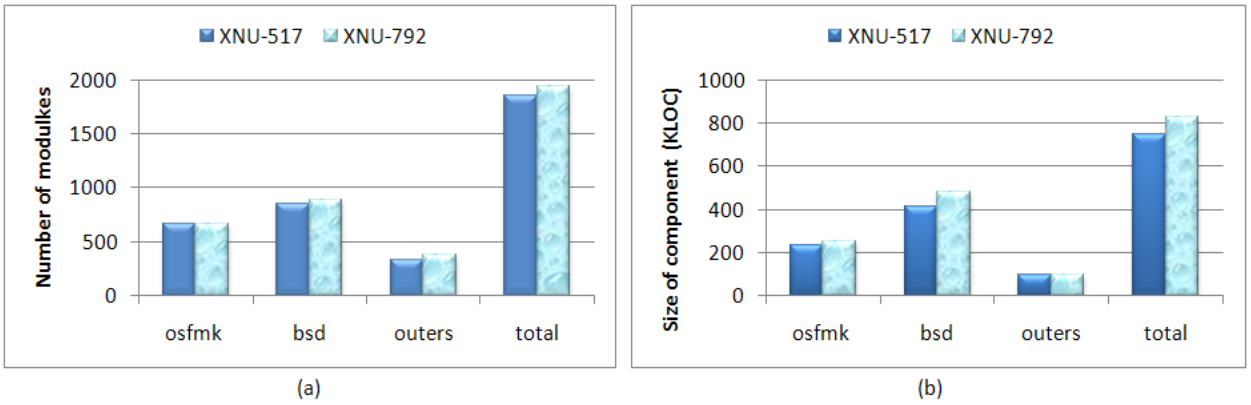

Figure 6. The evolution of Darwin from XNU-517 to XNU-791: (a) the number of modules; and (b) the size (KLOC) of the system

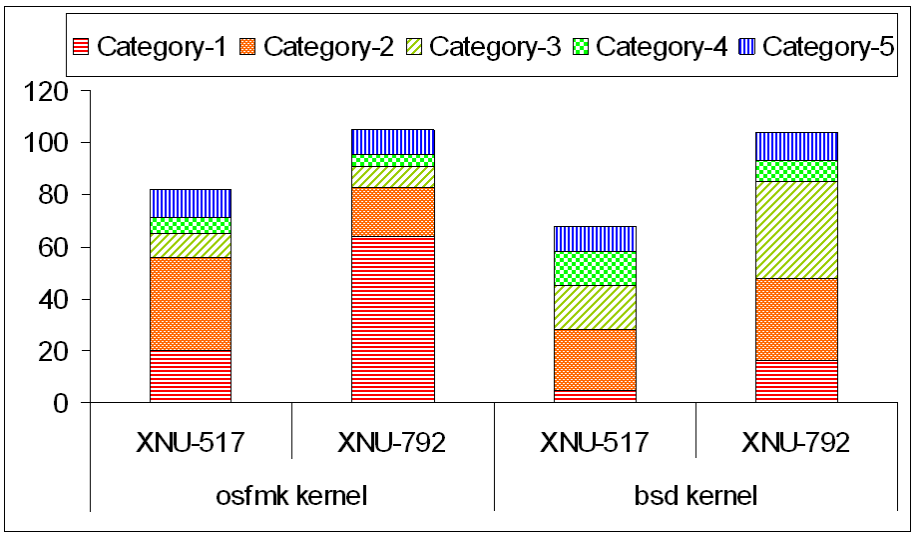

Figure 7. The number of global variables in Darwin kernels – *Categorization 1*

global variables, which are potential obstacles to kernel maintenance and reuse.

However, as we discussed before, the complexity induced by common coupling is related with the definition-use of global variables. To understand the evolution of the complexity in detail, we need to study the evolution of definition-use of global variables in Darwin from version XNU-517 to version XNU-792.

As described in Section 3, only the dependency-inducing definitions can affect kernel dependencies, we therefore studied the evolution of dependency-inducing definitions of global variables in **osfmk** kernel and **bsd** kernel. The results are shown in Table 4 and Table 5. The definitions are classified using the *Categorization 1* scheme, which applies to single-kernel-based software. It is worth noting that Category-1 global variables have non-dependency-inducing definitions and are accordingly not included in Table 4 and Table 5. Fig. 9 summarizes **osfmk** kernel (Table 4) and **bsd** kernel (Table 5) and shows the overall evolution of dependency-inducing definitions in Darwin. It can be seen that from version XNU-517 to version XNU-792, both the number of safe dependency-inducing definitions and the number of unsafe dependency-inducing definitions increased.

Table 6 shows the detail about the evolution of unsafe dependency-inducing definitions in Apple Darwin from the viewpoint

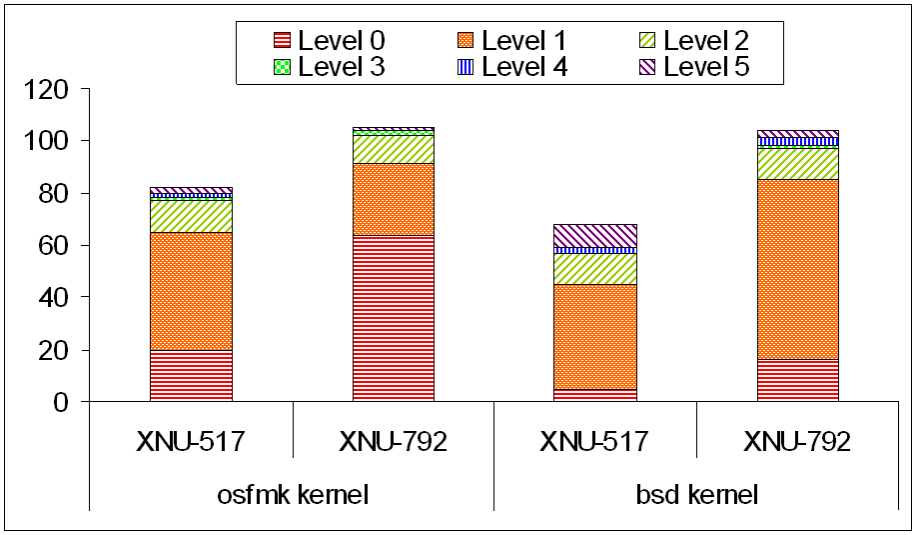

Figure 8. The number of global variables in Darwin kernels – *Categorization 2*

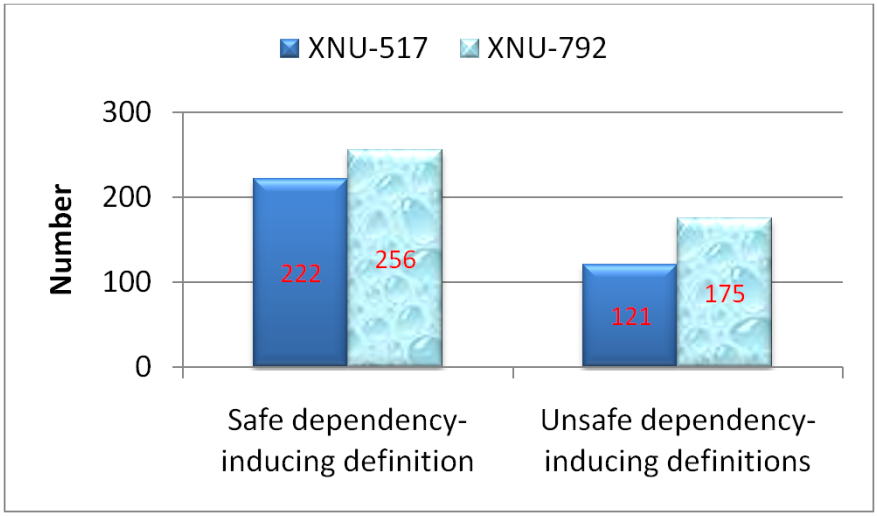

Figure 9. The evolution of the dependency-inducing definitions in Darwin

of multi-kernel-based system and the global variables are classified using *Categorization 2* scheme. It is worth noting that Level-0 and Level-1 global variables have no unsafe dependency-inducing definitions and are according not listed in Table 6.

As discussed in Section 3.3, high level (Level-4 and Level-5) global variables can bring more difficulties for kernel maintenance and reuse than low level (such as Level 2 and Level 3) global variables. Because the number of high level (Level-4 and Level-5) global variables is reduced from version XNU-517 to version XNU-792 (Figure 8), the number of unsafe dependency-inducing definitions induced by high level (Level-4 and Level-5) global variables decreased (Table 6). However, the number of unsafe dependency-inducing definitions induced by low level (Level 2 and Level 3) global variables increased tremendously (Table 6). It can be seen in Table 6, overall the number of unsafe dependency-inducing definitions increased from 121 in version XNU-517 to 175 in version XNU-792, matching the evolution of unsafe dependency-inducing definitions shown in Figure 9, which is derived from Table 4 and Table 5. Therefore, using two different categorization schemes, the same result is obtained: the complexity of Ap-

Table 4. The evolution of dependency-inducing definitions for **osfmk** kernel-*Categorization 1.*

Category number	Number of definitions			
	Definitions in kernel (Safe dependency-inducing)		Definitions in nonkernel (Unsafe dependency-inducing)	
	XNU-517	XNU-792	XNU-517	XNU-792
2	59	19	–	–
3	28	21	–	–
4	–	–	25	15
5	27	23	33	74
Sum	**114**	**63**	**58**	**89**

Table 5. The evolution of dependency-inducing definitions for **bsd** kernel – *Categorization 1.*

Category number	Number of definitions			
	Definitions in kernel (Safe dependency-inducing)		Definitions in nonkernel (Unsafe dependency-inducing)	
	XNU-517	XNU-792	XNU-517	XNU-792
2	34	32	–	–
3	52	134	–	–
4	–	–	48	15
5	22	27	15	71
Sum	**108**	**193**	**63**	**86**

Table 6. The evolution of unsafe dependency-inducing definitions in Darwin kernel – *Categorization 2*

Level number	**osfmk** kernel		**bsd** kernel	
	XNU-517	XNU-792	XNU-517	XNU-792
2	53	77	22	18
3	2	11	0	44
4	2	0	17	12
5	1	1	24	12
Sum	**58**	**89**	**63**	**86**

ple Darwin induced by global variables (common coupling) increased from version XNU-517 to version XNU-792.

4.3. Discussions

Software evolution is inevitable, because new features need to be frequently added and new hardware and platforms need to be continually supported. This is demonstrated in the evolution of Apple Darwin: From version XNU-517 to version XNU-792, both the size of kernel and the size of the entire system increased.

As new modules are added to the system, new dependencies need to be created between these new modules and existing modules, which will increase the complexity of the system. The maintenance activity performed on existing modules might also alter its original quality and increase the complexity of the system. Therefore, an evolving software system needs to be restructured regularly to retain its high quality design. This is also demonstrated in the evolution of Apple Darwin, in which, we found, from version XNU-517 to version XNU-792, Darwin has restructured through reducing the number of high category (level) global variables. This restructuring effort decreases the effect of the complexity increasing and dependency increasing due to the growth of the kernel size and the product size.

However, the definition-use analysis of the evolution of kernel dependencies shows that the number of dependency-inducing definitions, especially the number of the unsafe dependency-inducing definitions, increased from version XNU-517 to version XNU-792. Therefore,

the overall complexity of the system in the viewpoint of common coupling increased despite the effort of restructuring in reducing the number of unfavorite high category (level) global variables.

With the growth of the size of Darwin, both the module complexity and the module dependency are expected to continually increase. To reduce the effects of common coupling on kernel maintenance and kernel reuse, we suggest that major restructuring should be taken on Apple Darwin to reduce the number of dependency-inducing definitions, especially the number of unsafe dependency-inducing definitions.

5. Conclusions and Limitations

In this paper, we studied the evolution of the complexity of Apple Darwin from version XNU-517 to version XNU-792. We applied two schemes of categorization of common coupling, in which Apple Darwin is considered as both a single-kernel-based software system and a multi-kernel-based software system. Analysis of the two categorizations gives the same result. Specifically, the study found that from version XNU-517 to version XNU-792, the complexity of Apple Darwin increased. To reduced this increase of complexity, restructuring is necessary and it has been taken. Although the number of high category (level) unfavorite global variables is reduced in this restructuring process, the number of unsafe-dependency-inducing definitions increases. Therefore, the complexity of Darwin increased from the viewpoint of common coupling in spite of the effort of restructuring. Suggestions are that major restructurings should be taken to reduce or remove the unsafe dependency-inducing definitions in order to reduce the system complexity.

There are several limitations to this research. One limitation is that this research only focus on one type of coupling: common coupling; other types of component dependencies are not considered. The result could be improved if the evolution of more types of couplings are studied. Another limit is that only two versions of Apple Darwin are studied in this research. If more versions of Apple Darwin together with more versions of other operating systems are studied using the technique proposed in this paper, the result could be more interesting and convincing.

References

[1] P. B. Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 4(4):238–241, 1970.

[2] T. Härden. New approaches to object processing in engineering databases. In *Proceedings of International Workshop on Object-Oriented Database Systems*, pages 217–217, September 1986.

[3] Kernelthread. What is Mac OS X. http://www.kernelthread.com/mac/osx/, 2005.

[4] Mach. Mach 3.0 sources. http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/sources/, undated.

[5] J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *Journal of Systems and Software*, 20(3):295–808, 1993.

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

[7] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and J. Offutt. Maintainability of the Linux kernel. *IEE Proceedings–Software*, 149(1):18–23, 2002.

[8] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and J. Offutt. Quality impacts of clandestine common coupling. *Software Quality Journal*, 11(3):211–218, 2003.

[9] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(13):115–139, 1974.

[10] TrustedBSD. http://www.trustedbsd.org/sedarwin.html, 2008.

[11] J. West. How open is open enough? modeling proprietary and open source platform strategies. *Research Policy*, 32(7):1259–1285, 2003.

[12] L. Yu. Common coupling as a measure of reuse effort in kernel-based software with case studies on the creation of MkLinux and Darwin. *Journal of the Brazilian Computer Society*, 14(1):45–55, 2008.

[13] L. Yu and S. Ramaswamy. Categorization of common coupling in kernel-based software. In *Proceedings of the 43rd ACM Southeast Conference*, volume 2, pages 207–210, March 2005.

[14] L. Yu, S. R. Schach, and K. Chen. Common coupling as a measure of reuse effort in kernel-based software. In *Proceedings of 19th International Conference on Software Engineering and Knowledge Engineering*, pages 39–44, July 2007.

[15] L. Yu, S. R. Schach, K. Chen, and J. Offutt. Categorization of common coupling and its application to the maintainability of the Linux kernel. *IEEE Transactions on Software Engineering*, 30(10):694–706, 2004.

[16] L. Yu, S. R. Schach, K. Chen, J. Offutt, and G. Heller. Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD. *Journal of Systems and Software*, 79(6):807–815, 2006.

[17] L. Yu, S. R. Schach, K. Chen, and S. Ramaswamy. Coupling measurement in multi-kernel-based software with its application to Darwin. *The International Journal of Intelligent Control and Systems*, 13(2):109–118, 2008.