INTERNATIONAL JOURNAL OF INTELLIGENT CONTROL AND SYSTEMS
VOL. 13, NO. 2, JUNE 2008, 109-119

# Coupling Measurement in Multi-Kernel-Based Software with Its Application to Darwin

Liguo YU*, Stephen R. SCHACH**, Kai CHEN+ and Srini RAMASWAMY++

*Abstract*— **This paper addresses software dependency by presenting a multi-kernel-based software model and introducing the concept of directional coupling (d-coupling). A six-level d-coupling model to represent the dependencies of a kernel module on other modules in a multi-kernel-based software system is defined. This model can be used to represent the dependencies induced by all types of software coupling. The dependency model is applied to evaluate Darwin, a dual-kernel-based operating system. The study shows that (1) few strong dependencies exist between Darwin kernel modules and other modules; (2) from version XNU-517 to XNU-792, Darwin has been restructured to reduce the number of high level dependent modules induced by high level global variables to mitigate the effect of the increase of the dependency due to the growth of the kernel size and the product size.**

*Index Terms*— **Dependency, Coupling, Common Coupling, Multi-Kernel-Based Software, Darwin**

## 1. INTRODUCTION

Component-based software engineering is the production of software products through the systematic integration of existing software components. Quite frequently, existing components are not like ready-to-use building blocks, especially in the case of large-scale design-level reuse. Instead, these components need to be modified to meet the specific requirements of the new product. Furthermore, reused software components also need to be updated to meet new requirements or changes in the environment. Therefore, reusability and maintainability are two important properties of a software component [1. [2].

A considerable amount of work has been done to try to characterize reusable [3–7] and maintainable [8–10] software components. Both reusability and maintainability are related to coupling. If a software component is relatively independent, that is, if there are only a few dependencies of this component on other components, it would be easy to understand, maintain, and reuse this component. Coupling reflects the modifiability, maintainability, and reusability of a software product [11]. Certain types of coupling, especially common coupling, are considered to present risks for software development, reuse, and maintenance [12–15]. Hence, a maintainable and reusable component should be as independent as possible.

Manuscript received December 2, 2007, Revised March 12, 2008.

* Liguo Yu is an assistant professor of the Computer Science Department at Indiana University South Bend (e-mail: ligyu@iusb.edu).

** Stephen R. Schach is an associate professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University (e-mail: srs@vuse.vanderbilt.edu).

\+ Kai Chen is with Google, Inc.. (e-mail: kai.chenk@gmail.com).

++ Srinivasan Ramaswamy is professor and chairperson of the Computer Science Department at University of Arkansas at Little Rock (e-mail: srini@ieee.org / srini@acm.org).

Many software products, including most operating systems, are kernel-based. That is, as is further explained in Section 3, each implementation consists of the required kernel components, together with specific optional nonkernel components. The word "kernel" is overloaded. It can refer to a nucleus that can execute certain instructions [16, 17], or to a set of modules that are included in every installation. In this paper, "kernel" is used in the latter sense. In previous work [13–15], a categorization of common coupling was presented to measure the maintenance effort [13] and reuse effort [14] of a single-kernel-based software system. However, in software product lines and certain operating systems, closely related reused components are not always assigned to a single kernel but may be distributed among several kernels. This kind of system is called multi-kernel-based software. This paper presents a model to evaluate the dependencies induced by all types of coupling in systems with zero or more kernels. This model is then used to evaluate the dependencies of Darwin's two kernel-based components.

The remainder of the paper is organized as follows. Section 2 discusses software coupling. Section 3 describes multi-kernel-based software. Section 4 presents the coupling model for multi-kernel-based software. Section 5 analyzes various kinds of coupling, with special attention given to common coupling. Section 6 presents the application study of Darwin.

## 2. SOFTWARE COUPLING

Coupling is a measure of the degree of interaction between two software components (classes, modules, packages, or the like). Many different types of coupling have been identified, including data coupling, stamp coupling, control coupling, and common coupling [11, 18, 19]. Table 1 lists the definitions of several major types of coupling. The degree of dependency is considered in increasing order from top (data coupling) to bottom (common coupling). A good software system should have low coupling between components. Common coupling is considered to be a strong form of coupling, that is, it induces strong dependencies between software components, making software components difficult to understand, maintain, and reuse [12].

Table 1. Definitions of various kinds of coupling [19].

| Name | Definition |
|---|---|
| Data Coupling | Two components are data coupled if they pass data through a parameter that is a scalar or a record (structure) all of whose components are used. |
| Stamp Coupling | Two components are stamp coupled if they pass data through a parameter that is a record (structure), but not all of whose components are used. |
| Control Coupling | Two components are control coupled if one passes a variable to the other that is used to control the internal logic of the other. |
| Common Coupling | Two components are common coupled if they refer to the same global variable. |

Coupling between components strengthens the dependency of one component on others and increases the probability that changes in one component may affect other components, which makes maintenance difficult and more likely to introduce regression faults. It has been shown that coupling is related to fault-proneness of a software system [20–22]. Hence, strong coupling can have a detrimental effect on maintainability.

Strong coupling induces strong dependencies between software components, which also hamper software reuse. For example, if a software component has many dependencies on other components, it may be impossible to reuse this component in a new product without either (1) incorporating it together with the dependent components, or (2) redesigning and reimplementing this component to remove these dependencies. Option 1 may result in redundant reuse, whereas option 2 may result in changes to the functionality of the component. Hence, both these two approaches defeat the purpose of component reuse.

## 3. MULTI-KERNEL-BASED SOFTWARE

Software systems are growing in size. A typical software system may contain thousands of *modules*, where a module is a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier. To make the system easy to manage, modules are usually organized into components. In a large product, the components in turn may be organized into larger units. For simplicity, in this paper, the term *component* is used to denote a collection of modules, even if some of those modules have already been organized into constituent components.

For example, in operating systems and certain database management systems, the modules that are common to all installations are organized in one component, which is called the *kernel* component, whereas the architecture-specific or hardware-specific modules are organized into *nonkernel* components.

A software product that is composed of just one kernel component together with optional nonkernel components is referred as *single-kernel-based software*. In a single-kernel-based software product, a module could be either part of the kernel component (K) or the nonkernel components (NK). This is shown in Figure 1. Examples of single-kernel-based software are Linux, BSD, the Valentina database [23], and so on.

It is up to the software architect to decide whether a specific component is assigned to the kernel component or a nonkernel component. In general, the criterion utilized is that kernel components are the most frequently reused components. Ideally, kernel components should be relatively independent.

When a software product consists of more than one kernel-based component, it is referred as *multi-kernel-based software*. Figure 2 shows a *dual-kernel-based software* product, which is composed of two kernel-based components, $C_1$ and $C_2$, together with other components. In the remainder of this paper, the term *outer components* is used to refer to the software components external to the kernel-based components. In components $C_1$ and $C_2$, the constituent kernel components are represented as $K_1$ and $K_2$ and the nonkernel components as $NK_1$ and $NK_2$, respectively.

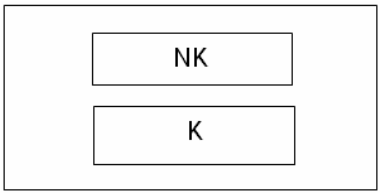


Figure 1: Depiction of single-kernel-based software.

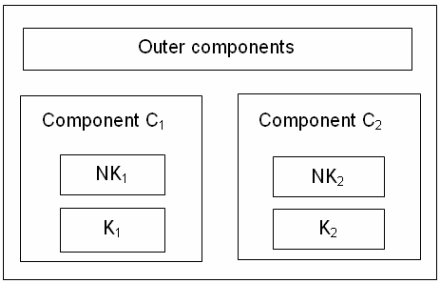


Figure 2: Depiction of dual-kernel-based software.

Figure 3 shows a multi-kernel-based software product that consists of n kernel-based components $C_1$, $C_2$, …, $C_n$, together with outer components. Darwin [24] is usually considered to be an example of dual-kernel-based software, because it consists of two kernel-based components (osfmk and bsd) and outer components. However, Darwin can also be viewed as triple-kernel-based software because it also contains a smaller kernel-based component, iokit, which deals with the input/output functions of the system. Because iokit is not as important as osfmk and bsd, in this paper Darwin is considered as a dual-kernel-based system. TrustedBSD SED Darwin [25] is another example of a multi-kernel-based operating system.

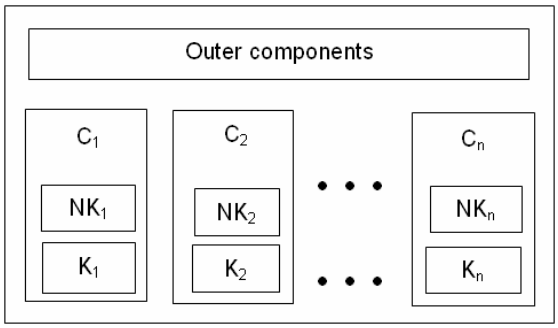

Figure 3: Depiction of multi-kernel-based software.

In general, multi-kernel-based software is associated with software reuse. Two kinds of software reuse may result in multi-kernel-based software. One is a software product line [26–28]. If a product is built from a product line with multiple input kernels (core assets), the resulting product will keep its multiple-kernel property and be multi-kernel-based software [29]. A second reuse mechanism is component-based software engineering. If the production of new software is through the integration of several kernel-based components, the new software may be multi-kernel-based. One example of this kind of multi-kernel-based software is Darwin, as described in Section 6. A component-based operating system [30–34] is built by integrating well-designed components, in which kernel modules are distributed among several components. Hence the resulting system will be multi-kernel-based.

## 4. DEPENDENCIES IN KERNEL-BASED SOFTWARE

As described before, coupling is a measure of the degree of interaction between two components. However, the concept of "coupling" does not explicitly express the directionality of the dependency between two components. For example, the statement "Module A is data coupled to module B" does not explicitly specify whether module A depends on module B or module B depends on module A. For conventional (no kernel) software in which all the modules are equally important, this does not make much difference. However, for kernel-based software, kernel module A and a nonkernel module B have different relative importance; the relationships "module A depends on module B" and "module B depends on module A" have different effects on software maintenance and reuse. Therefore, previous work explicitly specified the direction of the dependency relationship between two modules with directional coupling [35]. *Directional coupling* (*d-coupling* for short) was defined as follows: Module A is d-coupled to module B if modules A and B are coupled and certain changes made to module B have an immediate effect on module A because of the coupling. Module B is called the *dependency-inducing module* and module A is called the *dependent module*. The word "immediate" means that the dependency is not via some third module.

A graphical notation is used to represent d-coupling: A single-directional dashed arrow from module B to module A denotes that module A is d-coupled to module B. This is shown in Figure 4. The relation "module A is d-coupled to module B" is denoted by an arrow from module B to module A. This is because A is dependent on B; a change to module B can affect module A.

Suppose that modules A and B are coupled, and that modules A and B are both objects containing several methods. Suppose further that, inside module A, a message is sent to a method inside module B and a value is returned. Hence, the behavior of module A will depend on module B. That is, module A is d-coupled to module B.

A bidirectional dashed arrow between module A and module B denotes that module A is dependent on module B and module B is dependent on module A. This is shown in Figure 5. Such two-way d-coupling exists if, for example, inside module A, a message is sent to a method inside module B and a value is returned; and inside module B, a message is sent to a method inside module A and a value is returned.

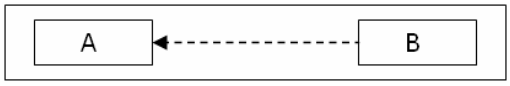

Figure 4: Depiction of module A d-coupled to module B.

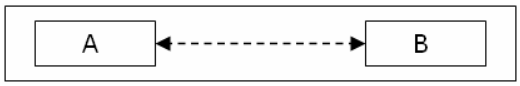

Figure 5: Depiction of module A d-coupled to module B and module B d-coupled to module A.

The kernel is the most important part of a kernel-based software product. It is the most frequently reused component. Reuse of kernel-based software usually consists of reusing the entire kernel together with certain other nonkernel components. The maintainability and reusability of the kernel determines the maintainability and reusability of the kernel-based software product as a whole. Therefore, the dependencies of kernel modules on other modules (either kernel or nonkernel) is more important than the dependency of a nonkernel module on other modules (either kernel or nonkernel are For example, in Figure 3, the dependencies of modules in $NK_1$ on modules in $NK_i$ ($2 \leq i \leq n$) are not of particular interest in this work, because the dependency is of nonkernel modules on other modules.

Consider Figure 3. Suppose that there is a kernel module A in kernel $K_1$ of component $C_1$, and that module A is d-coupled to another module B (that is, module A depends on module B). The d-coupling of module A to module B is classified into one of six levels, depending on the location of module B within the system, as follows:

- Level 1: Module B is a kernel module in the same kernel ($K_1$) as A.
- Level 2: Module B is a nonkernel module in the same component ($C_1$) as A.
- Level 3: Module B is in an outer component.
- Level 4: Module B is a nonkernel module in a different kernel-based component.
- Level 5: Module B is a kernel module in a different

kernel.

- On the hand, if Module A is coupled to module B but does not depend on module B, then this is an instance of Level 0: Module A is not d-coupled to Module B.

Figure 6 through Figure 11 depict examples of the six levels of d-coupling. If module A is coupled but not d-coupled to module B, then a level-0 d-coupling of module A to module B occurs. This is shown in Figure 6. An example of coupling that can induce this kind of d-coupling of module A on module B is when module B sends a message to a method in module A and a value is returned to module B. It should be noted that, in the case of level-0 d-coupling, it does not matter where module B is located; it could be in component $C_1$ (kernel or nonkernel), component $C_i$ ($2 \leq i \leq n$) (kernel or nonkernel), or an outer component.

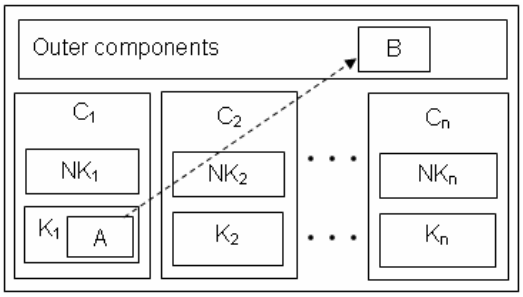


Figure 6: Level-0 d-coupling of module A on module B.

Figure 7 illustrates a level-1 d-coupling example: kernel module A in component $C_1$ is dependent on module B in kernel $K_1$ of the same component ($C_1$). Figure 8 depicts a level-2 d-coupling example: kernel module A in component $C_1$ is dependent on nonkernel module B in the same component ($C_1$).

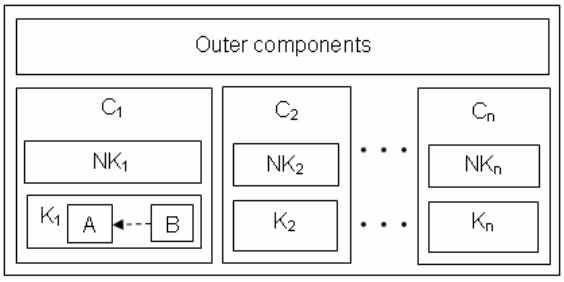


Figure 7: Level-1 d-coupling of module A to module B.

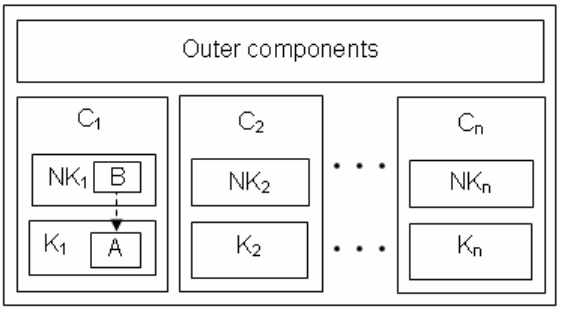


Figure 8: Level-2 d-coupling of module A to module B.

Figure 9 depicts a level-3 d-coupling example: Kernel module A in component $C_1$ is dependent on module B in an outer component. Figure 10 shows a level-4 d-coupling example: Kernel module A in component $C_1$ is dependent on nonkernel module B in another kernel-based component $C_i$ ($2 \leq i \leq n$). Figure 11 illustrates a level-5 d-coupling example: Kernel module A in component $C_1$ is dependent on kernel module B in another kernel-based component $C_i$ ($2 \leq i \leq n$).

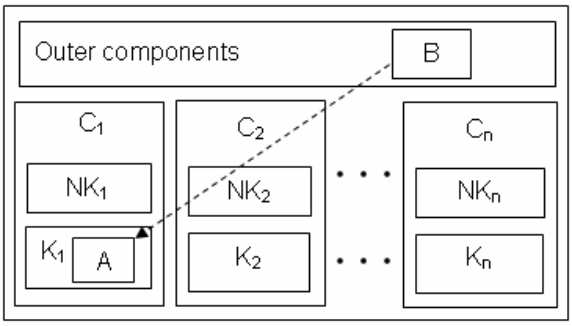


Figure 9: Level-3 d-coupling of module A to module B.

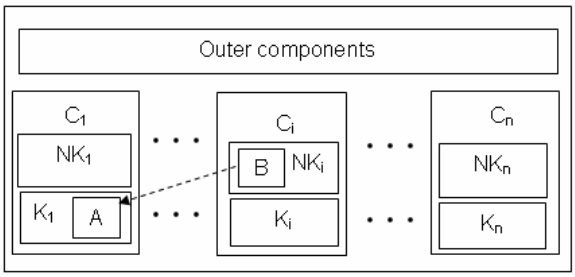


Figure 10: Level-4 d-coupling of module A to module B.

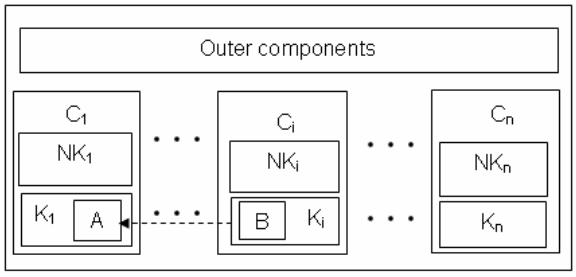


Figure 11: Level-5 d-coupling of module A to module B.

Dependencies are related to software understandability,

maintainability, and reusability. If a module is relatively independent and is not d-coupled to other modules, it will be relatively easy to understand, maintain, and reuse that module.

For level-0 d-coupling, kernel module A does not depend on module B. The coupling between module A and module B does not affect the independence of module A, which therefore makes kernel module A easy to understand, maintain, and reuse.

For level 1, dependencies exist between kernel modules within the same component. This is the least unfavorable type of d-coupling. First, this d-coupling is easy to understand and maintain, because the d-coupling is localized to the same kernel component. Second, this d-coupling may affect the reuse of a single kernel module, but it does not affect the reuse of the kernel as a whole.

For level 2, a kernel module depends on a nonkernel module in the same component. This d-coupling has two effects: First, any maintenance (changes) made to this nonkernel module can affect the dependent kernel module. Second, in order to reuse the kernel module, it needs to be modified to remove the d-coupling or to be reused together with the dependency-inducing nonkernel module. Level-2 d-coupling is stronger (less desirable) than level-1 d-coupling. However, an instance of level-2 d-coupling is still located within the same component, so it does not affect the reuse of the component as a whole.

Level-3 d-coupling is from a kernel module in a kernel-based component to a module in an outer component. This d-coupling makes the maintenance and reuse of the kernel module more difficult; changes to the module in the outer component may affect the kernel module, and reuse of the kernel component may require reuse of the module in the outer component as well.

Level-4 d-coupling is from a kernel module in a kernel-based component to a nonkernel module in another kernel-based component. The d-coupling exists between two different kernel-based components and defies the independence property of kernel-based components; in multi-kernel-based software, each kernel-based component needs to be relatively independent.

Level 5 represents the most unfavorable d-coupling, d-coupling from one kernel module to another kernel module in a different component. Consider the reuse of kernel module A in component $C_1$, which is dependent on kernel module B in another in component $C_2$. Either module A would need to be modified to remove its d-coupling to module B, which may affect the functionality of module A; or module B would need to be reused together with module A. However, because module B is also a kernel module, strong dependencies are expected to exist between module B and the other modules in component $C_2$ (either kernel or nonkernel). These other modules would have to be reused together with module B, resulting in a large amount of unnecessary reuse. Level-5 d-coupling represents the dependency between two separate kernels. Because the kernels generally contain the most important and commonly used modules, such a kernel dependency is the strongest (and therefore the most undesirable) dependency between two components. Any change to one kernel may affect other kernels across components; any reuse of one kernel may require additional reuse of other kernels across components.

Each individual instance of d-coupling constitutes the dependence of one module on another module, and can therefore involve at most two components. The model as a whole is for multi-kernel-based software, that is, it applies to software that is dual-kernel-based, triple-kernel-based, or higher. However, the model can also be applied to single-kernel-based software or even to conventional (no kernel) software. For single-kernel-based software as shown in Figure 1, only the three lowest levels are needed to represent the dependency of the single kernel: level 0, level 1, and level 2. For conventional (no kernel) software, all modules and components are equally important. Only the two lowest d-coupling levels are needed to represent the module dependency: level 0 and level 1.

In summary, the six-level d-coupling model of kernel components represents a hierarchical ordering of dependency relationships. To achieve a high degree of independence of kernel components, lower-level dependencies are preferable to higher-level dependencies. This six-level d-coupling model can be used to represent the dependency of kernel modules on other modules induced by any type of coupling, including data coupling, stamp coupling, control coupling, and common coupling.

## 5. DETERMINATION OF DEPENDENCIES IN KERNEL-BASED SOFTWARE

Directional coupling (d-coupling) is a special case of classical (nondirectional) coupling. The constructs of most modern programming languages (such as C, C++, and Java) can induce the classical types of coupling: data coupling, stamp coupling, control coupling, and common coupling [11]. The coupling type can be determined according to the definitions in Table 1. Data coupling, stamp coupling, and control coupling are induced via values returned by function calls or messages sent to methods. It is easy to determine the direction of the dependency: If module A calls module B and a value is returned, then module A depends on module B. Depending on where module B is located, the d-coupling level can be determined.

On the other hand, d-coupling induced by common coupling is much more difficult to evaluate, as can be seen from the following definition-use analysis. In this paper, special attention is paid to the d-coupling induced by common coupling, for two reasons:

(1) The d-coupling induced by data coupling, stamp coupling, or control coupling can involve only two modules. For example, if module A depends on other modules only as a consequence of a function call, one cannot add more dependencies of module A on other modules without making changes to module A.

However, dependencies of module A on other modules induced by common coupling can increase without modifying module A. This is shown below using definition-use analysis.

(2) The d-coupling induced by common coupling is much more complicated than the other kinds. The fact that there is common coupling between two modules does not necessary mean there is d-coupling between them. This, too, will be shown using definition-use analysis.

### *5.1 Definition-Use Analysis*

Each occurrence of a variable in source code is either a definition of the variable or use of the variable. A *definition* of a variable x is a statement that assigns a value to x. The most common form of definition is an assignment statement, such as x = 2. The *use* of a variable x is a statement that utilizes the value of x, such as y = x + 7. From the creation 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 components. For example, if component $C_1$ and component $C_2$ both access a global variable gv, it is generally concluded that there is an instance of d-coupling from $C_1$ to $C_2$ or from $C_2$ to $C_1$. However, by applying definition–use analysis and studying this coupling in greater depth, in some instances, neither d-coupling exists.

In more detail, because definitions can affect uses but uses cannot affect definitions, dependencies between components induced by global variables are induced by the definition–use relationship. For example, if components $C_1$, $C_2$, and $C_3$ all access global variable gv and there are definitions in only $C_1$ and $C_2$ and uses in only $C_3$, dependencies exist between components $C_1$ and $C_3$ and between components $C_2$ and $C_3$, but there is no d-coupling from component $C_1$ to component $C_2$ or from component $C_2$ to component $C_1$. More precisely, the d-coupling from component $C_3$ to component $C_1$ is caused by component $C_3$ (which uses a global variable) depending on component $C_1$ (which defines a global variable). Component $C_1$ affects the maintainability and reusability of component $C_3$, because a use of a global variable depends on the value assigned to that global variable by a previous definition. Furthermore, if a new component $C_4$ that defines gv is now added to the system, $C_3$ will be d-coupled to $C_4$. That is, dependencies of $C_3$ can increase without any modification of $C_3$ [12].

Definition-use analysis has been used to study common coupling in single-kernel-based software [13] [14]. In this paper, it is used to study multi-kernel-based software. Based on the six levels of d-coupling presented in Section 4, a level-L global variable is defined as a global variable that induces level-L directed common coupling, L = 0, 1, …, 5.

### *5.2 Level-0 Global Variables*

A level-0 global variable is defined in one or more kernel modules but has no uses in kernel modules. A level-0 global variable could be defined in other modules, either a nonkernel module in the same component or a module in another component, but this does not affect the independence of the kernel modules that define that global variable. Because there is no use of a level-0 global variable in a kernel component, definitions in other components (either kernel or nonkernel) cannot affect kernel components. Accordingly, the presence of a level-0 global variable will not cause difficulties in understanding, maintaining, or reusing a kernel component.

### *5.3 Level-1 Global Variables*

A level-1 global variable is defined and used within the same kernel component but not defined in any other component, thereby inducing level-1 d-coupling. The kernel modules that use the level-1 global variable depend on the kernel modules that define the global variable. However, as previously mentioned, an instance of level-1 d-coupling does not affect the independence of the kernel as a whole.

### *5.4. Level-2 Global Variables*

In order to induce an instance of level-2 d-coupling, a level-2 global variable is used in kernel modules and is defined in one or more nonkernel modules of the same component. A kernel module that uses a level-2 global variable depends on the nonkernel modules that define the global variable.

### *5.5. Level-3 Global Variables*

To induce an instance of level-3 d-coupling, a level-3 global variable is used in kernel modules of one component and is defined in one or more modules of an outer component. This dependency on an outer component hampers the maintenance and reuse of the kernel component in which the level-3 global variable is used.

### *5.6. Level-4 Global Variables*

In order to induce an instance of level-4 d-coupling, a level-4 global variable is used in kernel modules of one component and is defined in one or more nonkernel modules of another kernel-based component.

### *5.7. Level-5 Global Variables*

To induce an instance of level-5 d-coupling, a level-5 global variable is used in kernel modules of one component and is defined in one or more kernel modules of another component.

### *5.8. Multiple Levels*

The relation between the six-level d-coupling model and the six-level global variable categorization is not one-to-one, because a single global variable may induce more than one level of d-coupling. For example, one global variable may induce both level-4 and level-5 dependencies. In such a case, the higher level of d-coupling is assigned to the global variable, because the higher level is a more accurate predictor of reuse and maintenance problems.

## 6. DEPENDENCIES IN THE DARWIN KERNEL

### *6.1. Darwin Overview*

MAC OS X is an operating system for Macintosh computers [24, 36]. OS X is structured around Darwin, an open-source core. Darwin, in turn, conceptually consists of two subsystems: One subsystem is based on Mach version 3.0 [37], the other subsystem is largely based on FreeBSD 5.1 [38, 39]. Both Mach and FreeBSD are single-kernel-based systems.

This paper analyzes version XNU-517 and version XNU-792 of Darwin. The two major components in Darwin are osfmk, which contains the Mach subsystem, and bsd, which contains the FreeBSD subsystem. Because osfmk and bsd preserve the kernel-based property of the Mach and FreeBSD operating systems, the resulting new product, Darwin, is dual-kernel-based.

The two reused kernel-based components, osfmk and bsd, were written in the C language. Other components of Darwin were written in C or C++. In the remainder of this paper, the term *module* is used to refer to a source code component written in C or C++ (".c" file, ".cpp" file, or ".h" file).

The objective of this study is first to show how to apply the d-coupling model to analyze a multi-kernel-based software product and second to study the evolution of the kernel dependency of Darwin. This study was performed using the Linux cross-referencing tool, lxr, together with a source code analysis program written in Perl.

### *6.2. Common Coupling in XNU-517*

As discussed before, the study of common coupling needs to incorporate the definition-use analysis of global variables, which makes it more complicated than function calls. Therefore, in this subsection, Darwin XNU-517 is used to illustrate how to analyze common coupling in multi-kernel-based software. First, global variables appearing in the osfmk kernel or the bsd kernel were identified. Every instance of a global variable was labeled as either a definition or a use of that variable. Next, each global variable was characterized to one of the six levels as defined in Section 5.

Examples of global variables of the six levels of Section 5 are represented graphically in Figure 12 through Figure 17. This graphical notation was first introduced in [13] to represent common coupling in single-kernel-based software. Here it is extended to represent common d-coupling in multi-kernel-based software. A solid arrow is used to represent a definition–use relation between modules. A unidirectional arrow from module $M_1$ to module $M_2$ means that the global variable is defined in $M_1$ and used in $M_2$. A double–headed arrow between module $M_1$ and module $M_2$ means the global variable is defined in $M_1$ and $M_2$ and used in $M_1$ and $M_2$. A pair $(d, u)$ is used to indicate (number of definitions, number of uses) of a global variable within a module, and the triple $(d, u, m)$ is used to indicate (number of definitions, number of uses, number of modules) of a variable in a component that contains a set of modules.

Figure 12 shows a level-0 global variable halt_in_debugger in the osfmk kernel. It is defined in kernel module debug.c once and appears in four nonkernel modules; in those four modules, there are eight definitions and five uses of halt_in_debugger. It induces level-0 d-coupling from kernel module debug.c to other modules. Although debug.c is common coupled to other modules, it has no dependency on those other modules.

Figure 13 shows a level-1 global variable nlinesw in the bsd kernel. It is defined in kernel module tty_conf.c once and used in tty.c once. nlinesw induces level-1 d-coupling from kernel module tty.c to kernel module tty_conf.c.

Figure 14 shows a level-2 global variable not_in_kdp in the osfmk kernel. It has five uses in bsd_kern.c. It appears in five nonkernel modules, in which there are six definitions and seventeen uses. not_in_kdp induces level-2 level d-coupling from kernel module bsd_kern.c to nonkernel modules. It depends on nonkernel modules within the same component.

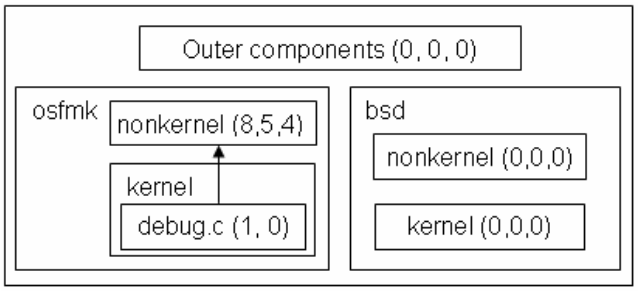


Figure 12: Level-0 global variable halt_in_debugger in the osfmk kernel of XNU-517.

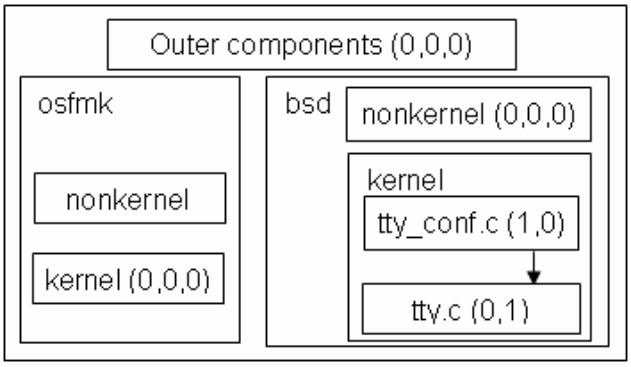


Figure 13: Level-1 global variable nlinesw in the bsd kernel of XNU-517.

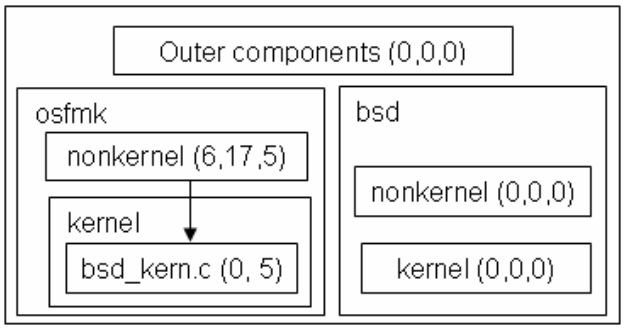


Figure 14: Level-2 global variable not_in_kdp in the osfmk kernel of XNU-517.

Figure 15 shows a level-3 global variable debug_mode in the osfmk kernel. It represents the dependencies of kernel-based component osfmk on other components.

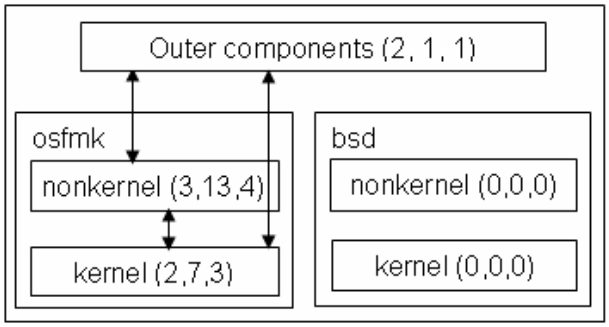


Figure 15: Level-3 global variable debug_mode in the osfmk kernel of XNU-517.

Figure 16 shows an example of a level-4 global variable pc_trace_cnt in the bsd kernel. Changes to osfmk nonkernel modules may affect bsd kernel modules. A level-4 global variable may induce dependencies between two kernel-based components: a kernel module of one kernel-based component may depend on a nonkernel module of another kernel-based component. In general, component dependencies induced by level-4 global variable are stronger than those induced by level-0 through level-3 global variables, because the dependency involves two kernel-based components.

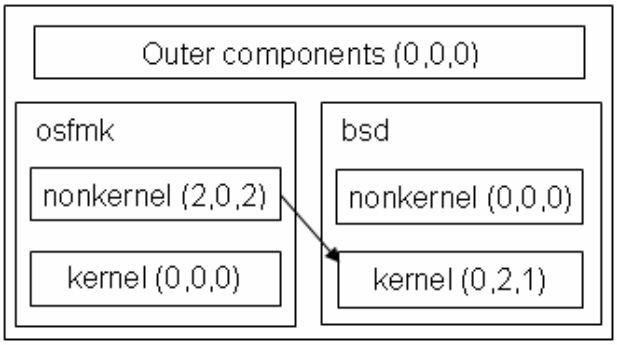


Figure 16: Level-4 global variable pc_trace_cnt in the bsd kernel of XNU-517.

Figure 17 shows the example of level-5 global variable panicstr in the bsd kernel. Changes to osfmk kernel modules may affect bsd kernel modules. There are five definitions of panicstr in the osfmk kernel but none in the bsd kernel. Reuse of the bsd kernel may require the entire osfmk kernel to be reused, too, because the value of panicstr used in the bsd kernel is defined in the osfmk kernel. Level-5 global variables induce dependencies between two kernel-based components: A kernel module of one kernel-based component depends on a kernel module of another kernel-based component. In general, component dependencies induced by level-5 global variables are the strongest. Changes in kernel modules of one kernel-based component will result in modifications to kernel modules of another kernel-based component. Reuse of one kernel may require reuse of modules from another kernel.

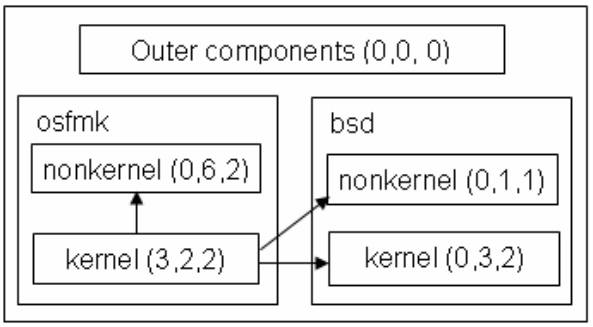


Figure 17: Level-5 global variable panicstr in the bsd kernel of XNU-517.

The categorization of global variables into these six levels is associated with a specific kernel-based component. Depending on the kernel being considered, the same global variable may be assigned to different levels. For example, in Figure 17, global variable panicstr is categorized as level-5 with respect to the bsd kernel. However, if considering the osfmk kernel, panicstr is a level-0 global variable.

*6.3. The Evolution of Kernel Dependency of Darwin*

To study the changes of kernel dependency of Darwin with the evolution of the product, coupling in two versions of Darwin, XNU-517 and XNU-792, were compared. These two versions were released in November 2003 and April 2005, respectively, and related data are provided in Table 2 and Table 3. From version XNU-517 to version XNU-792, the size of kernel (measured in KLOC) increased by 7.5 percent and the size of the product (measured in KLOC) increased by about 11.4 percent.

Table 2. The structure of Darwin XNU-517.

| | Number of modules | | Size (KLOC) | |
|---|---|---|---|---|
| Component | kernel | nonkernel | kernel | nonkernel |
| osfmk | 115 | 554 | 48 | 188 |
| bsd | 81 | 770 | 62 | 351 |
| Others | 332 | | 94 | |
| Total | 1852 | | 744 | |

Table 3. The structure of Darwin XNU-792.

| | Number of modules | | Size (KLOC) | |
|---|---|---|---|---|
| Component | kernel | nonkernel | kernel | nonkernel |
| osfmk | 100 | 568 | 38 | 210 |
| bsd | 90 | 799 | 80 | 400 |
| Others | 377 | | 99 | |
| Total | 1934 | | 829 | |

The osfmk kernel and the bsd kernel were studied in two versions. For each kernel module, all the dependent modules induced by function calls and by global variables were identified. As illustrated in Table 1, function calls are associated with data coupling and stamp coupling, and global variables are associated with common coupling. Dependencies induced by function calls will make the program easier to understand and maintain than dependencies induced by global variables. Accordingly, dependencies (especially low-level d-coupling) induced by function calls are weaker and safer than dependencies (especially high-level d-coupling) induced by global variables.

Table 4 shows the evolution of kernel dependencies induced by function calls (an instance of level-0 d-coupling clearly involves no dependent modules). It can be seen that majority of the dependent modules of the Darwin kernel belong to level 1 or level 2; less than 10 percent of the dependent modules belong to levels 3 to 5. From the viewpoint of d-coupling induced by function calls, Darwin kernel has a weak dependency.

Table 5 shows the evolution of the global variables in Darwin. It can be seen that, in both versions, the majority of the global variables in the osfmk and bsd kernels fall into level 0 or level 1. As previously described, level-0 or level-1 d-coupling does not affect the dependencies of the kernel as a whole; it is level-2, -3, -4, and -5 global variables that affect the dependencies of the kernel as a whole. From the viewpoint of d-coupling induced by global variables, the Darwin kernel is well structured.

Table 4. The number of dependent modules in the Darwin kernels induced by function calls.

| Level-number | osfmk kernel | | bsd kernel | |
|---|---|---|---|---|
| | XNU-517 | XNU-792 | XNU-517 | XNU-792 |
| 0 | – | – | – | – |
| 1 | 389 | 476 | 70 | 143 |
| 2 | 226 | 593 | 928 | 1224 |
| 3 | 0 | 0 | 7 | 36 |
| 4 | 0 | 0 | 62 | 132 |
| 5 | 0 | 0 | 0 | 0 |

Table 5. The number of global variables in the Darwin kernels.

| Level-number | osfmk kernel | | bsd kernel | |
|---|---|---|---|---|
| | XNU-517 | XNU-792 | XNU-517 | XNU-792 |
| 0 | 20 | 64 | 5 | 16 |
| 1 | 45 | 27 | 40 | 69 |
| 2 | 12 | 11 | 12 | 12 |
| 3 | 1 | 2 | 0 | 1 |
| 4 | 2 | 0 | 2 | 3 |
| 5 | 2 | 1 | 9 | 3 |
| Total | 82 | 105 | 68 | 104 |

From version XNU-517 to version XNU-792, for the bsd kernel, the number of global variables of level 2 remained unchanged, the number of level-3 and level-4 global variables increased by 2, the number of level-5 global variables decreased by 6; for the osfmk kernel, the number of global variables of levels 2, 4, and 5 decreased; the number of level-3 global variables increased by 1. Considering that a level-0 or level-1 global variable does not affect kernel maintenance, it can deduce that from version XNU-517 to version XNU-792, the number of global variables that can induce high level dependencies is reduced.

Table 6. The number of dependent modules induced by global variables in Darwin kernels.

| Level-number | osfmk kernel | | bsd kernel | |
|---|---|---|---|---|
| | XNU-517 | XNU-792 | XNU-517 | XNU-792 |
| 0 | – | – | – | – |
| 1 | 58 | 37 | 45 | 87 |
| 2 | 44 | 35 | 18 | 14 |
| 3 | 2 | 3 | 0 | 4 |
| 4 | 1 | 0 | 22 | 7 |
| 5 | 2 | 1 | 12 | 6 |

Like Table 4 that shows the dependencies induced by function calls, Table 6 summarizes the dependencies induced by global variables. For the osfmk kernel and the bsd kernel, the number of dependent modules of levels 2, 4, and 5 decreased and the number of dependent modules of level 3 increased by 5. Again, considering that it is dependent modules of levels 2, 3, 4, and 5 that affect kernel dependency, Table 6

shows that from version XNU-517 to version XNU-792, Darwin has reduced the number of high-level dependent modules (induced by high-level global variables), which are potential obstacles to kernel maintenance.

From version XNU-517 to version XNU-792, both the size of kernel and the size of the product increased, so it is expected that the dependencies of the kernel modules would also increase for all modules to interact and work together properly. Tables 4 through 6 show that, to accommodate the increase in kernel dependency, Darwin evolved with an increase of dependent modules induced by functions and low-level global variables, instead of high-level global variables. As previously mentioned, dependencies induced by global variables, especially high-level global variables, make maintenance more difficult compared to dependencies induced by function calls. The study shows that, from version XNU-517 to version XNU-792, Darwin has been restructured with a lowering of the number of high-level dependent modules and the number of high-level global variables through which the dependencies are induced.

## 7. ACKNOWLEDGEMENTS

This work was based in part, upon research supported by the National Science Foundation (CNS-0619069, EPS-0701890 and OISE 0650939), Acxiom Corporation (# 281539) and NASA EPSCoR Arkansas Space Grant Consortium (# UALR 16804).

## REFERENCES

[1] W. Lim, "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software*, vol. 11, no. 5, pp. 23–30, 1994.

[2] W.B. Frakes and G. Succi, "An Industrial Study of Reuse, Quality, and Productivity," *Journal of Systems and Software*, vol. 57, no. 2, pp. 99–106, 2001.

[3] M.W. Price and S.A. Demurjian, "Analyzing and Measuring Reusability in Object-Oriented Design," *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 22–33, Atlanta, Georgia, 1997.

[4] T.J. Biggerstaff, and A.J. Perlis, *Software Reusability: Concepts and Models*, vol. 1, ACM Press, New York, NY, 1989.

[5] L. Briand, S. Morasca, and V.R. Basili, "Defining and Validating High-Level Design Metrics," *Computer Science Technical Report Series*, vol. CS-TR-3301, University of Maryland at College Park, College Park, MD, 1994.

[6] D.N. Card and R.L. Glass, *Measuring Software Design Quality*, Prentice-Hall, Upper Saddle River, NJ, 1990.

[7] F. Dandashi, "Software Engineering: Theory, Application and Practice: A method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements," *Proceedings of the 2002 ACM Symposium on Applied Computing*, pp. 997–1003, Madrid, Spain, March 2002.

[8] G.M. Berns, "Assessing Software Maintainability," *Communications of the ACM*, vol. 27, no. 1, pp. 14–23, 1984.

[9] V.R. Gibson and J.A. Senn, "System Structure and Software Maintenance Performance," *Communications of the ACM*, vol. 32, no.3, pp. 347–358, March 1989.

[10] R.D. Banker, S.M. Datar, C.F. Kemerer, and D. Zweig, "Software Complexity and Maintenance Costs" *Communications of the ACM*, vol. 36, no. 11, pp. 81–94, 1993.

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

[12] 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.

[13] 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*, vol. 30, no. 10, pp. 694–706, 2004.

[14] L. Yu, S.R. Schach, and K. Chen, "Common Coupling as a Measure of Reuse Effort in Kernel-Based Software," *Proceedings of 19th International Conference on Software Engineering and Knowledge Engineering*, pp. 39–44, Boston, MA, July 2007.

[15] L. Yu and S. Ramaswamy, "Categorization of Common Coupling in Kernel-Based Software," *Proceedings of the 43rd ACM Southeast Conference,* Kennesaw, GA, vol. 2, pp. 207–210, 2005.

[16] P. Brinch Hansen, "The Nucleus of a Multiprogramming System," *Communications of the ACM*, vol. 4, no. 4, pp. 238–241, 1970.

[17] T. Härden, "New Approaches to Object Processing in Engineering Databases," *Proceedings of International Workshop on Object-Oriented Database Systems*, pp. 217, Sept. 1986.

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

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

[20] D. Kafura and S. Henry, "Software Quality Metrics Based on Interconnectivity," *Journal of Systems and Software*, vol. 2, no. 2, pp. 121–131, 1981.

[21] R.W. Selby and V.R. Basili, "Analyzing Error-Prone System Structure," *IEEE Transactions on Software Engineering*, vol. 17, no. 2, pp. 141–152, 1991.

[22] D.A. Troy and S.H. Zweben, "Measuring the Quality of Structured Design," *Journal of Systems and Software*, vol. 2, no. 2, pp. 113–120, 1981.

[23] Paradigmasoft, 2005, http://www.paradigmasoft.com/kernel.html.

[24] Apple Computer, "Inside Mac OS X," May 2000, www.apple.com.pl/infotech/varia/macosx/InsideMacOSXSystemOverview.pdf.

[25] http://www.trustedbsd.org/sedarwin.html.

[26] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns,* Addison-Wesley, Reading, MA, 2001.

[27] D. Weiss, C. Lai, and R. Tau, *Software Product-Line Engineering: a Family-Based Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[28] C. Atkinson, *Component-Based Product Line Engineering with UML*, Addison-Wesley, London, New York, 2002.

[29] http://softwareproductlines.com/index.html.

[30] T. Jaeger, J. Liedtke, V. Panteleenko, Y. Park, and N. Islam, "Security Architecture for Component-Based Operating Systems," *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pp. 222–228, Sintra, Portugal, 1998.

[31] G. Law and J. McCann, "Decomposition of Preemptive Scheduling in the Go! Component-Based Operating System," *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop*, Kolding, Denmark, pp. 201–206, 2002.

[32] D.R. Engler, M.F. Kaashoek, and J. O'Toole, "Exokernel: an Operating System Architecture for Application-Level Resource Management," *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 251–266, Copper Mountain, Colorado, 1995.

[33] E. Gabber, J. Bruno, J. Brustoloni, A. Silberscatz, and C. Small, "The Pebble Component-Based Operating System," *Proceedings of the USENIX Technical Conference*, Monterey, CA, June 1999.

[34] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg, "The Performance of µ-Kernel-Based Systems," *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 66–77, Saint Malo, France, 1997.

[35] L. Yu, "Understanding Component Co-Evolution with a Study on Linux," *Empirical Software Engineering*, vol.12, no.2, pp. 123–41, 2007.

[36] http://www.apple.com/pr/library/2001/mar/21osxstore.html.

[37] http://www2.cs.cmu.edu/afs/cs/project/mach/public/www/sources/sources_top.html.

[38] http://www.kernelthread.com/mac/osx/arch_xnu.html.

[39] J. West, "How Open is Open Enough? Modeling Proprietary and Open Source Platform Strategies," *Research Policy*, vol. 32, no. 7, pp. 1259–1285, 2003.

Liguo Yu received the PhD degree in computer science from Vanderbilt University. He is an assistant professor of computer science at Indiana University South Bend. His research concentrates on software dependency and open-source software development.

Stephen R. Schach is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University. He is the author of over 130 refereed research papers and 11 software engineering textbooks.

Kai Chen received the Ph.D. degree from the Department of Electrical Engineering and Computer Science at Vanderbilt University. He is working at Google, Inc.

Srini Ramaswamy earned his Ph.D. degree in Computer Science in 1994 from the University of Southwestern Louisiana. He is currently the chairperson of the Department of Computer Science, University of Arkansas at Little Rock.