Empir Software Eng (2007) 12:123–141
DOI 10.1007/s10664-006-9000-x

Understanding component co-evolution with a study on Linux

Liguo Yu

Published online: 25 April 2006
© Springer Science + Business Media, LLC 2006
Editor: Per Runeson

Abstract After a software system has been delivered, it inevitably has to change to remain useful. Evolutionary coupling measures the change dependencies between software components. Reference coupling measures the architecture dependencies between software components. In this paper, we present a method to correlate evolutionary coupling and reference coupling. We study the evolution of 597 consecutive versions of Linux and measure the evolutionary coupling and reference coupling among 12 kernel modules. We compare 12 pairs of evolutionary coupling data and reference coupling data. The results show that linear correlation exists between evolutionary coupling and reference coupling. We conclude that in Linux, the dependencies between software components induced via the system architecture have noticeable effects on kernel module co-evolution.

Keywords Software evolution · Co-evolution · Coupling · Linux

1 Introduction

Software evolution (Lehman, 1980; Perry, 1994) is a process that a software system changes from a simpler or worse state to a higher or better state (Arthur, 1988). Software evolution is inevitable, because changes are needed to fix defects or to satisfy the new requirements. *Evolutionary coupling* measures the change dependencies between software components. Two software components are evolutionary coupled, if they have been changed at the same time (Weißgerber et al., 2005). Evolutionary coupling is usually studied by looking at the version history of the software system.

Evolutionary coupling has been used to predict software changes and to prevent incomplete changes (Zimmermann et al., 2005). For example, during maintenance, if you made a change on one software component, what other components should you change? Are these changes complete? The evolution history of the product may

L. Yu (✉)
Computer and Information Sciences Department, Indiana University South Bend,
1700 Mishawaka Ave., P.O. Box 7111 South Bend, IN 46634, USA
e-mail: ligyu@iusb.edu

provide useful answers to these questions. Evolutionary coupling could be used to detect coupling undetectable by program analysis, such as coupling between non-program files (Zimmermann et al., 2003). Evolutionary coupling can also assist software testing, especially regression testing. The applications of evolution coupling are based on the observation that if two components are frequently changed together in the earlier release, they are likely to be changed together in the later release.

The concept of evolutionary coupling is in contrast to the traditional software coupling that measures the degree of interactions between software components based on the architecture of the system (Stevens et al., 1974). The traditional software coupling is generally categorized to data coupling, stamp coupling, control coupling, and common coupling (Offutt et al., 1993). The definitions of these types of coupling are shown in Table 1. Because the traditional architecture based coupling is related with the references between software components, we call it *reference coupling* in contrast to the evolutionary coupling.

Reference coupling between components strengthens the dependency of one component on others and increases the probability that changes in one component may affect other components. It has been claimed that strong reference coupling has effects on software maintenance. These effects include change dependencies and fault-proneness (Selby and Basili, 1991). For corrective maintenance, the changes will not always be restricted to the fault component itself; it is not uncommon to have to modify more than one component to fix a single fault. For adaptive and perfective maintenance, changes to a function implementation in one component could result corresponding changes of a calling function in another component. In other words, it is generally agreed on that that strong reference coupling can result aggregate changes on software components.

Previous studies on evolutionary coupling and reference coupling are largely performed separately. This work addresses the relationship between evolutionary coupling and reference coupling empirically. First, we adapt and present methods to measure evolutionary coupling and reference coupling between components. Then we study the correlation between evolutionary coupling and reference coupling in a case study. The objective of this study is to show whether reference coupling plays an important role in the measurement of evolutionary coupling. This insightful understanding will be beneficial to other studies and application of software evolution.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 describes software components co-evolution, its relationship between

Table 1 Definitions of various types of reference coupling (Offutt et al., 1993)

Name	Definition
Data coupling	Two components are data coupled if they pass data through a parameter that is a scalar.
Stamp coupling	Two components are stamp coupled if they pass data through a parameter that is a record (structure).
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.

evolutionary coupling and reference coupling. Section 4 describes the representation of evolutionary coupling. Section 5 describes the representation of reference coupling. Section 6 presents our case study on Linux. Our conclusions appear in Section 7.

2 Related Work

Since Stevens et al. (1974) introduced the concept of software component dependency, reference coupling, including some specific categories, such as common coupling (Schach et al., 2003; Yu et al., 2004) has been widely studied (Chidamber and Kemerer, 1994; Briand et al., 1999; Arisholm et al., 2004). Most of the work performed in the past decades is intended to understand the relationship between reference coupling and the external quality factors of a software product. In these studies, researchers have identified clear empirical relationships between reference coupling and fault-proneness for both structured software (Kafura and Henry, 1981; Selby and Basili, 1991; Troy and Zweben, 1981) and object-oriented software (Briand and Wuest, 2002).

In contrast to the extensive research on reference coupling, the study of evolutionary coupling has begun just a few years ago because of the advances in data mining technology. However, despite of its short history, several interesting projects are undergoing and the published results are exciting and promising. Zimmermann et al. (2004, 2005) applied data mining to software version history and designed a tool to help programmers find related changes based on evolutionary coupling. The tool can predict likely future modifications and prevent errors due to incomplete changes. Independently from Zimmermann et al. (2004, 2005), Ying et al. (2004) applied association rule to determine change patterns (sets of files that were changed together) from the change history of the source code. Their study can be used to predict future changes of software modules.

Xing and Stroulia (2004, 2005) studied the evolutionary coupling by comparing the design level documents, UML diagrams. The design differences between two versions of the product indicate the evolution of the product. Hassan and Holt (2004) proposed several heuristics to predict change propagations and presented a framework to measure the performance of the proposed heuristics. Some heuristics are based on historical co-change and static dependencies.

Evolutionary coupling is also used in software quality control. Graves et al. (2000) used change history data to successfully predict the distribution of incidences of faults. Their studies show that the change patterns found based on evolutionary coupling can be used to recommend potentially relevant change and predict possible fault locations to a developer performing a maintenance task. Williams and Hollingsworth (2005) used the source code change history of a software project to help search for bugs. The studies show that their bug-finding technique is more effective than the same static analysis that does not use historical data from the source code repository.

For most of the related work described above, the study of reference coupling and evolutionary coupling was performed separately. Researchers observed the relationship between reference coupling and evolutionary coupling and several studies are performed to understand their relationship.

To our knowledge, Gall et al. (1998) were the first to use evolutionary coupling data to represent reference coupling (they call it logical coupling) between modules.

They developed a technique for detecting change patterns and applied it to a large Telecommunication Switching System and identified potential dependencies among modules. Zimmermann et al. (2003) analyzed the revision history of individual product files and functions to detect the fine-grained reference coupling. Although Zimmermann et al. (2003) noticed that strong evolutionary coupling exist between files that has strong reference coupling, they did not provide the empirical evidence for this relationship. However, in their series studies, Zimmermann et al. (2003, 2004, 2005) presented a method to represent evolutionary coupling, which is adapted in our study and will be further discussed in Section 4.

3 Software Components Co-Evolution

Co-evolution is a terminology used in biology to indicate change in the genetic composition of one species in response to a genetic change in another. In software engineering, we use *co-evolution* to represent the phenomenon in software evolution that change in one component C2 in response to a change in another component C1. C1 is called *causal component* and C2 is called *effect component*.

Co-evolution can be represented with the evolutionary coupling between components. However, there are differences between co-evolution and evolutionary coupling. Co-evolution is the result of cause–effect changes between two components: changes to one component require changes to another component. Evolutionary coupling is based on the evolution history of two components and is a measurement of the observation that two components are changed at the same time. Modifications made to two components at the same release do not necessarily indicate the co-evolution of the two components. These modifications could be completely unrelated and accidentally made to the two components. In contrast to co-evolution, we call this kind of evolution *independent evolution*. Independent evolution could also accidentally cause two components to be changed at the same release and it will bring noise to the measurement of evolutionary coupling. However, the noise added to the measurement of evolutionary coupling due to independent evolution is expected to be random. If the noise value is smaller compared to the measurement, it should not affect the study of co-evolution.

Co-evolution is based on the fact that software components are interdependent, changes in one component are likely to cause changes in others. Therefore, reference coupling is the theoretical basis of components co-evolution. However, as described before, software evolution is one of the most relevant software processes. Changes made to components are the results of many factors (such as code enhancement) and may not solely depend on reference coupling.

Figure 1 shows the relationships among component co-evolution, reference coupling and evolutionary coupling. Reference coupling is the basis of component co-evolution, the effect of co-evolution is contained in the measurement of evolutionary coupling. On the other hand, independent evolution can bring noise data to the measurement of evolutionary coupling. If in most cases, software components are co-evolved, which means the changes are related with reference coupling, there will be strong correlation between reference coupling and evolutionary coupling. On the other hand, if software components are evolved independently, there will be no correlation between reference coupling and evolutionary coupling.

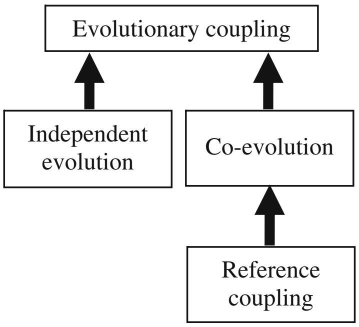

Fig. 1 The relationships among co-evolution, reference coupling, and evolutionary coupling.

As mentioned in Section 1, the objective of this study is to present a method and a case study to show how to determine whether evolutionary coupling represents software co-evolution and reference coupling.

4 Evolutionary Coupling Representation

Evolutionary coupling between two components can be determined from the evolution history of the two components. Here we adapt the methods proposed by Zimmermann et al. (2003, 2004, 2005) to represent evolutionary coupling. Suppose a software system has n consecutive versions V1 through Vn, each new release is based on the previous version. To study the evolutionary coupling between two components C1 and C2 of this software system, we use the notation C1 ⇨ C2 to indicate the *cause–effect rule* that changes to C1 result changes in C2 in the same release. C1 is the *causal component* and C2 is the *effect component*. The amount of evidence of this cause–effect rule is associated with measures:

- Transaction count. The transaction count is the number of changes made on C1 in these n releases. Assume C1 was modified in 10 releases, the transaction count is 10.
- Support count. The support count is the number of changes made on C2 while C1 is changed. Assume, in 9 releases, changes are made to both C1 and C2. Therefore, the support count for the cause–effect rule C1 ⇨ C2 is 9.
- Support ratio.[1] The support ratio determines the strength of the cause–effect rule, or the relative amount of the given consequences. It is represented as support count / transaction count. The support ratio for the above cause–effect rule C1 ⇨ C2 is 9/10 = 0.9.

In this paper, we use the support ratio of a cause–effect rule to represent evolutionary coupling. It should be noted that (1) support count does not 100% represent the co-evolution of two components. The co-changes made to two components may be accidental and have no cause–effect relation; (2) evolutionary coupling is a directional coupling. C1 ⇨ C2 and C2 ⇨ C1 have different meanings. The former cause–effect rule states that changes made to C1 result changes on C2, while the later

[1] In other research, this is called *confidence*. To avoid the confusion with the *confidence* used in statistics we use *support ratio* in this paper.

cause–effect rule states that changes made on C2 result changes on C1. The support ratios for the two rules could be dramatically different.

5 Reference Coupling Representation

Reference coupling is also a directional coupling. However, the traditional concept of "coupling" does not explicitly express the directionality of the dependency between two components. For example, the statement "Component C1 is data coupled to component C2" does not explicitly specify whether component C1 depends on component C2 or component C2 depends on component C1. The relationships "component C1 depends on component C2" and "component C2 depends on component C1" have different effects on software evolution. If component C1 depends on component C2, changes made to component C2 can lead to changes to component C1, but not vice versa. Therefore, we need to explicitly define the direction of the dependency relationship between these two components. We define *directional coupling* as follows: Component C2 is directional-coupled to component C1 if components C1 and C2 are coupled and certain changes made to component C1 have an immediate effect on component C2 because of the reference coupling. The word "immediate" means that the dependency is not via some third component.

We use a graphical notation to represent directional-coupling: a single-directional solid arrow from component C1 to component C2 denotes that component C2 is directional-coupled to component C1. This is shown in Fig. 2. We remark that the relation "component C2 is directional-coupled to component C1" is denoted by an arrow from component C1 to component C2. This is because C2 is dependent on C2; a change to component C1 can affect component C2.

Suppose that components C1 and C2 are coupled, and that components C1 and C2 are both objects containing several methods. Suppose further that, inside component C2, a message is sent to a method inside component C1 and a value is returned. Hence, the behavior of component C2 will depend on component C1. That is, component C2 is directional-coupled to component C1.

Directional 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. Data coupling, stamp coupling, and control coupling are induced via values returned by function calls or messages sent to methods. Common coupling is induced via global variables. The determination of the direction of the dependency induced by data coupling, stamp coupling, and control coupling can be performed using the definition-use analysis of functions. The determination of the direction of the dependency induced by common coupling can be performed using the definition-use analysis or global variables (Yu et al., 2004).

Definition of a function is the implementation of a function in one component. Use of a function is the invocation of the function in another component. Therefore, the component that uses a function depends on the component that defines the function. Data coupling, stamp coupling, and control coupling induce dependencies

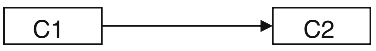

Fig. 2 Depiction of component C2 directional-coupled to component C1.

between components via the definition and use of functions. Changes made to the definition of a function, such as the passing parameter type, can result changes in the use (invocation) of the function.

Each occurrence of a variable in source code is either a definition of that variable or use of that 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 $x = 12$. The *use* of a variable x is a statement that utilizes the value of x, such as $y = x-5$. Common coupling induces dependencies between components via definition-use of a global variable. For example, if component C1 and component C2 both access a global variable gv, C1 defines gv, and C2 uses gv, we say component C2 is directional-coupled to component C1 via common coupling. In other words, because definitions can affect uses but uses cannot affect definitions, dependencies between components induced by global variables are induced by the definition-use relationship.

We use C1 ➔ C2 to represent component C2 directional-coupled to component C1 via reference coupling. We remark that we use different notations to represent evolutionary coupling and reference coupling: hollow arrow is used to represent the cause-effect rule, solid arrow is used to represent reference coupling. C1 ⇨ C2 represents the cause–effect rule that changes made to C1 result changes made to C2 in the same release based on version history; C1 ➔ C2 represents the directional reference coupling between C1 and C2 based on system architecture. The relationship between evolutionary coupling and reference coupling has been described in Section 3 and is shown in Fig. 1.

In this study, the strength of the directional reference coupling between component C1 and component C2 is associated with two measures, directional coupling count and dependency path count.

1. Directional coupling count. Directional coupling count is the number of directional coupling between C1 and C2. For data coupling, stamp coupling, and control coupling, it is the number of functions defined in C1 and used in C2. For common coupling, it is the number of global variables defined in C1 and used in C2. In Fig. 3, component C2 depends on component C1 via 1 function call, func(int), and 1 global variable, gv, the directional coupling count is 2.
2. Dependency path count. Dependency path count is the number of dependency path between components C1 and C2. For dependency induced by function call, the dependency path is a path from the definition of the function in component C1 to the use of the function in component C2. For dependency induced by global variable, the dependency path is path from the definition of a global variable in component C1 to the use of that global variable in component C2. We

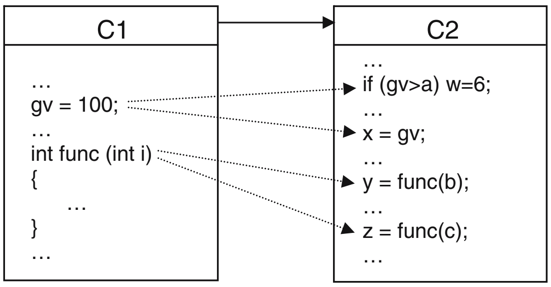

Fig. 3 Depiction of the dependency path between components C1 and C2

use dashed arrow to represent the dependency path. In Fig. 3, the dependency path count between C1 and C2 is 4 (2 for function call func(int), 2 for global variable gv).

In this paper, the strength of the dependency induced by reference coupling is represented with the dependency path count between two components. Accordingly, the strength for the above reference coupling C1 ➔ C2 is 4.

6 Linux Case Study

6.1 General Result

The open-source software development life-cycle model can best be described as continuous maintenance, as encapsulated in the dictum "release early and often" (Raymond, 2001). Usually, many versions are released for one open-source software product. There are two reasons to choose Linux for our case study: (1) Linux is one of the most active open-source projects. It has released about 600 versions, which can provide us with rich version history data; (2) A lot of research (Godfrey and Tu, 2000; Schach et al., 2002) has performed to study the evolution patterns of Linux, including the growth of size, changes of kernel structure, and the community properties. However, no research has performed to study its components co-evolution.

From version 1.0.0 to version 2.6.11, there are 597 formal releases of Linux. The evolution of the whole Linux product can be represented as an evolution tree. Linux is structured with software modules. We consider each module as a component. The most important components in Linux are kernel modules, which contain the architecture independent key functions in operating systems.

The Linux version 1.0.0 has 23 kernel modules. In the evolution process, some kernel modules are removed, some new modules are added. In version 2.6.11, there are 58 kernel modules. In these 58 kernel modules, only 12 continuously evolved from version 1.0.0. This means 12 kernel modules experienced the entire evolution of Linux.

In our preliminary study, we found that the release of Linux did not follow a predicable pattern. Some releases contain fewer changes that are less than 100 lines of code. For example, in version 2.1.84, there are only 78 lines of code changed compared to version 2.1.83. While other releases contain changes over 100 thousand lines of code, such as version 2.5.74. Because the amount of changes in each release of Linux is not uniform, the more versions we select and study, the evolutionary coupling should be more accurate. Therefore, we decide to study the co-evolution of these 12 kernel modules in 597 releases. The 12 kernel modules are *dma.c, exit.c, fork.c, itimer.c, module.c, panic.c, printk.c, ptrace.c, sched.c, signal.c, sys.c*, and *time.c*.

For the simplicity, the 597 versions are indexed from 1 to 597, which are shown in Table 2. The evolution study of the 12 kernel modules is performed on the entire evolutionary tree of Linux. For every version in Table 2, we determine whether the corresponding kernel module was modified compared to the previous version. The data extracted based on version history is used to calculate the evolutionary coupling between these 12 kernel modules. The reference coupling study is performed on version 2.4.20 using Linux Cross Reference Tool (lxr). The reason we choose version 2.4.20 is that it is one of the most stable versions. We expect that the reference

Table 2 The version index of Linux

Version	1.0	1.1	1.2	1.3	2.0	2.1	2.2	2.3	2.4	2.5	2.6
Index	1–12	13–108	109–122	123–223	224–264	265–397	398–424	425–477	478–509	510–585	586–597

coupling in version 2.4.20 can represent the architecture dependencies of other versions of Linux. We will discuss more about the validity of this representation later in Subsection 6.4.

Figure 4 depicts the cumulative number of changes for each module in different versions. That is, it adds all the number of changes from version index 1 to the current version. If a module is modified in a certain release, it is counted as 1 change. Otherwise, it is counted as 0. The figure shows that, in the earlier version release, more times of changes are made to *time.c* than to *printk.c*, while in recent release, more times of changes are made to *printk.c* than to *time.c*. The pitch of the line indicates the frequency of modification to a module. Also, we can see from Fig. 4 that the number of times of changes made to these 12 kernel modules is very different, with *dma.c* has the least numbers of changes and *sched.c* has the largest number.

Only based on Fig. 4, the cumulative number of changes for each module in different versions, it is hard to tell whether some modules are co-evolved or independently evolved. To answer this question, we need to study evolutionary coupling and reference coupling in these 12 kernel modules and understand their relationship.

6.2 Component Co-Evolution

There are total 12*11 = 132 cause-effect evolutionary rules among these 12 kernel modules (mi ⇨ mj, i = 1..12, j = 1..12, i ≠ j). Table 3 shows the support count for different evolutionary rules. The first column is the transaction count for each module. The second column through the last column shows the support count for every cause–effect rule. These calculations are based on the discussions in Section 4. The value in row mi and column mj is the support count for evolutionary rule mi ⇨ mj (changes to mi result changes in mj in the same release). For example, the transaction count for m1 (*dma.c*) is 21. The support count for cause–effect rule m1 (*dma.c*) ⇨ m2 (*exit.c*) is 12. The transaction count for m4 (*itimer.c*) is 26, the support count for cause–effect rule m4 (*itimer.c*) ⇨ m9 (*sched.c*) is 23.

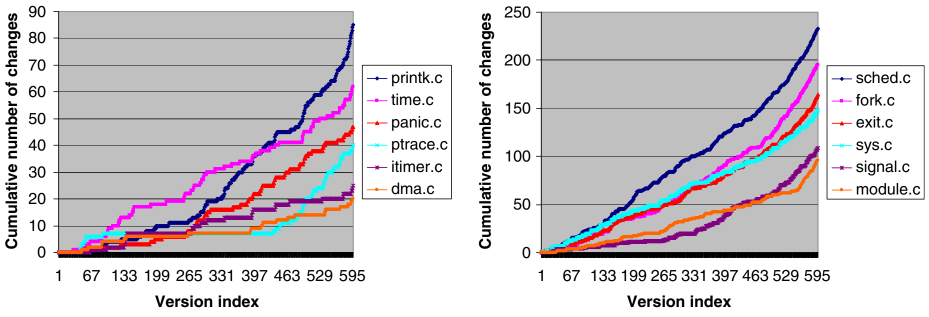

Fig. 4 The cumulative number of changes of 12 Linux kernel modules

Table 3 The support count of 132 cause–effect evolutionary rules

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
m1 (21)	–	12	11	5	8	3	9	3	14	9	8	3
m2 (165)	12	–	111	14	48	24	40	24	104	64	75	39
m3 (196)	11	111	–	11	53	27	55	28	120	68	77	32
m4 (26)	5	14	11	–	15	9	12	4	23	16	12	14
m5 (97)	8	48	53	15	–	21	35	14	56	36	44	26
m6 (48)	3	24	27	9	21	–	22	9	35	22	25	15
m7 (86)	9	40	55	12	35	22	–	15	57	37	39	22
m8 (41)	3	24	28	4	14	9	15	–	31	22	23	12
m9 (234)	14	104	120	23	56	35	57	31	–	77	89	48
m10 (110)	9	64	68	16	36	22	37	22	77	–	54	29
m11 (149)	8	75	77	12	44	25	39	23	89	54	–	40
m12 (63)	3	39	32	14	26	15	22	12	48	29	40	–

m1 dma.c, m2 exit.c, m3 fork.c, m4 itimer.c, m5 module.c, m6 panic.c, m7 printk.c, m8 ptrace.c, m9 sched.c, m10 signal.c, m11 sys.c, m12 time.c

Table 4 shows the support ratio of various evolutionary rules. The value in row mi and column mj is the support ratio for evolutionary rule mi ⇨ mj ($i \neq j$). It is calculated by dividing the transaction count of mi (item in row mi and column 1) with the corresponding support count of cause–effect rule mi ⇨ mj (item in row mi and column mj) in Table 3. It can been from Table 4, all the evolutionary rules have the support ratio value greater than zero, which means, all 132 pairs of components evolved together to some extent. However, only based Table 4, we cannot tell these support ratio values are the results of co-evolution (reference coupling) or accidentally independent evolution.

There are also 12*11=132 directional reference coupling among 12 kernel modules (mi ➔ mj, i = 1..12, j = 1..12, $i \neq j$). Table 5 shows the various pairs of directional reference coupling measured in dependency path count as discussed in Section 5. The value in row mi and column mj is the dependency path count of directional reference coupling mi ➔ mj ($i \neq j$). It can be seen from Table 5 that some measurements of directional reference coupling, such as m1 ➔ m2 and m5 ➔ m4, are zero, which indicate no possibility of direct co-evolution of these components.[2] Therefore, we would expect the corresponding evolutionary coupling in Table 4, such as m1 ⇨ m2 and m5 ⇨ m4, to be zero. However, Table 4 shows these values are not zero. This means, measurements of evolutionary coupling in 12 Linux kernel modules not only represent components co-evolution—as a result of reference coupling—but also contain (to some extent) the accidental data of independent evolution.

Now, based on the measurements of evolutionary coupling of 132 cause–effect rules and the measurements of 132 reference coupling among 12 modules, we would like to answer two questions. First, is the evolutionary coupling in Table 4 an accurate representation of component co-evolution or an accidental result of independent evolution? Second, did reference coupling play an important role in Linux kernel component co-evolution? To answer these two questions, we need to study the correlation between evolutionary coupling and reference coupling. In other words, we

[2] In this paper, we only consider the direct co-evolution of two components. We ignore the co-evolution of two components as the result of their common dependencies on a third component.

Table 4 The support ratio of the cause–effect evolutionary rules (mi ⇨ mj, i ≠ j)

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
m1 ⇨	–	0.57	0.52	0.24	0.38	0.14	0.43	0.14	0.67	0.43	0.38	0.14
m2 ⇨	0.07	–	0.67	0.08	0.29	0.15	0.24	0.15	0.63	0.39	0.45	0.24
m3 ⇨	0.06	0.57	–	0.06	0.27	0.14	0.28	0.14	0.61	0.35	0.39	0.16
m4 ⇨	0.19	0.54	0.42	–	0.58	0.35	0.46	0.15	0.88	0.62	0.46	0.54
m5 ⇨	0.08	0.49	0.55	0.15	–	0.22	0.36	0.14	0.58	0.37	0.45	0.27
m6 ⇨	0.06	0.5	0.56	0.19	0.44	–	0.46	0.19	0.73	0.46	0.52	0.31
m7 ⇨	0.1	0.47	0.64	0.14	0.41	0.26	–	0.17	0.66	0.43	0.45	0.26
m8 ⇨	0.07	0.59	0.68	0.1	0.34	0.22	0.37	–	0.76	0.54	0.56	0.29
m9 ⇨	0.06	0.44	0.51	0.1	0.24	0.15	0.24	0.13	–	0.33	0.38	0.21
m10 ⇨	0.08	0.58	0.62	0.15	0.33	0.2	0.34	0.2	0.7	–	0.49	0.26
m11 ⇨	0.05	0.5	0.52	0.08	0.3	0.17	0.26	0.15	0.6	0.36	–	0.27
m12 ⇨	0.05	0.62	0.51	0.22	0.41	0.24	0.35	0.19	0.76	0.46	0.63	–

would like to determine whether the evolutionary coupling increases as the reference coupling increases.

Both evolutionary coupling and reference coupling are between two modules. We want to study the cause–effect rule of co-evolution. Therefore, we divide 132 cause–effect rules into 12 groups based on the causal component. Each group is made of the same rows taken from Table 4 (evolutionary coupling) and Table 5 (reference coupling). As an example, Table 6 contains a group of data that is based on the causal component m11. The other groups are similar to Table 6 and they are not shown here. In more detail, we test the following 12 null hypotheses:

- *H01: There is no linear relationship between the evolutionary coupling and reference coupling for co-evolutions of causal component* m1 *and other components.*
- *H02: There is no linear relationship between the evolutionary coupling and reference coupling for co-evolutions of causal component* m2 *and other components.*
- ...

Table 5 The measurement of directional reference coupling (mi ➔ mj, i ≠ j)

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
m1 ➔	–	0	0	0	0	0	0	0	0	0	0	0
m2 ➔	0	–	39	22	0	15	15	26	43	83	131	22
m3 ➔	0	37	–	11	0	3	3	14	42	71	117	10
m4 ➔	0	42	15	–	0	8	19	8	122	76	32	36
m5 ➔	0	0	0	0	–	0	0	0	0	0	0	0
m6 ➔	0	0	1	0	0	–	0	0	1	0	0	0
m7 ➔	0	0	1	0	0	0	–	0	3	0	0	0
m8 ➔	0	2	4	0	0	0	0	–	8	0	1	0
m9 ➔	0	69	45	23	0	15	15	31	–	91	142	25
m10 ➔	0	48	38	22	0	14	14	25	42	–	128	21
m11 ➔	0	84	74	58	0	50	50	61	98	118	–	57
m12 ➔	0	37	27	11	0	3	3	14	31	71	117	–

Table 6 The data for hypothesis test *H011* with causal component m11.

Effect component	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m12
Evolutionary coupling	0.05	0.5	0.52	0.08	0.3	0.17	0.26	0.15	0.6	0.36	0.27
Reference coupling	0	84	74	58	0	50	50	61	98	118	57

- *H012: There is no linear relationship between the evolutionary coupling and reference coupling for co-evolutions of causal component* m12 *and other components.*

To test these hypotheses, we would need to calculate the correlation, which summarizes the strength of the relationship between the two variables, evolutionary coupling and reference coupling. Several different correlation coefficients have been put forward, including Pearson's correlation coefficient and Spearman's rank correlation coefficient (Nolan, 1994). For Pearson's correlation coefficient to be valid, two variables need to be normally distributed. However, it is unlikely that either the evolutionary coupling or the reference coupling has a normal distribution. Therefore, we use Spearman's rank correlation coefficient. If the rank correlation coefficient proves to be statistically significant at the 0.1 level, we will reject the null hypothesis.

Table 7 shows the result of the hypothesis tests. A correlation coefficient can take on the values from −1.0 to 1.0. Where −1.0 is a perfect negative (inverse) correlation, 1.0 is a perfect positive correlation, and 0.0 is no correlation. In 12 tests, 9 have significance level 0.05 (two-tailed), 1 has significance level 0.01 (two-tailed).

Let's pay attention to the two null hypotheses, *H01* and *H05*. The causal components for these two tests are m1 and m5. Table 5 shows that, for both of them, the directional reference coupling m1 ➔ mi ($i \neq 1$) and m5 ➔ mi ($i \neq 5$) are constant zero values. Therefore, it is not surprising that no correlation between reference coupling and evolutionary coupling is identified in those two tests. Reference coupling with zero value means that other component does not depend on the causal component (m1 and m5). Therefore, in theory, changes made to m1 and m5 could not result changes to other components. On the other hand, the measurement of evolutionary coupling is based on the version history of components. Table 4 shows the corresponding evolutionary couplings (row m1 and row m5) are not zero. There are three possible reasons that lead to this discrepancy: (1) the support count of the cause–effect rule mi ⇨ mj ($i \neq j$) is also the support count for rule mj ⇨ mi ($i \neq j$). We can see from Table 3 that support count is symmetric along the principal diagonal. Therefore, the support count for rules mi ⇨ m1 ($i \neq 1$) and mi ⇨ m5 ($i \neq 5$) can cause the

Table 7 The correlation coefficient of 12 hypothesis tests

Hypothesis	*H01*	*H02*	*H03*	*H04*	*H05*	*H06*	*H07*	*H08*	*H09*	*H010*	*H011*	*H012*
Causal component	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
Number of pairs (*N*)	11	11	11	11	11	11	11	11	11	11	11	11
Coefficient (*r*)	–	0.65	0.63	0.68	–	0.67	0.68	0.86	0.64	0.64	0.64	0.71
Significance level (*p*)	–	0.05	0.05	0.05	–	0.05	0.05	0.01	0.05	0.05	0.05	0.05

Table 8 The non-zero data for hypothesis test *H011* with causal component m11

Effect component	m2	m3	m4	m6	m7	m8	m9	m10	m12
Evolutionary coupling	0.5	0.52	0.08	0.17	0.26	0.15	0.6	0.36	0.27
Reference coupling	84	74	58	50	50	61	98	118	57

support count for the corresponding rules m1 ⇨ mi (i ≠ 1) and m5 ⇨ mi (i ≠ 5) to be nonzero; (2) two components that both depend on and co-evolve with a third component can bring support count for the evolutionary rule between these two components; (3) independent evolution can also generate accidental support count for rules m1 ⇨ mi (i ≠ 1) and m5 ⇨ mi (i ≠ 5) and result the nonzero measurements of evolutionary coupling.

In Table 5, there are many zero values for reference coupling. The reference coupling with the same zero values will receive the same rank in Spearman's test. Could these same zero values increase the correlation coefficient? Could they affect the results of the hypothesis tests? To assess the effect of these many same (zero) values in reference coupling, we performed the correlation tests again by using only the nonzero data of reference coupling. As an example, Table 8 shows the adjusted data for hypothesis test *H011*. Table 9 shows the adjusted correlation coefficient of 12 hypothesis tests. Because in these tests, only nonzero reference coupling data is used, different tests have different pairs of data, which is indicated in row 3 of Table 9.

Compare Table 7 with Table 9, we can see that four significance levels changed from 0.05 to 0.1 (the significance of the correlations decreased), the corresponding hypothesis tests are *H03*, *H09*, *H010*, and *H011*; two significance levels changed from 0.05 to 0.01 (the significance of the correlations increased), the corresponding hypothesis tests are *H04* and *H07*. According to this comparison, we deduce that reference coupling with many zero values does not necessarily increase the correlation.

Therefore, based on the result shown in Table 7, we reject 10 null hypotheses: *H02, H03, H04, H06, H07, ...,* and *H012*. We cannot reject null hypotheses *H01* and *H05*. Whereas, based on the result shown in Table 9, we reject 9 null hypotheses, *H02, H03, H04, H07, ...,* and *H012*. We cannot reject null hypotheses *H01*, *H05* and *H06*. The reason that we cannot reject *H01*, *H05* or *H06* is the data for these hypotheses is not valid to perform the test.

The fact that we rejected 9 null hypotheses out of 12 means that (1) evolutionary coupling in Linux kernel modules is relatively an accurate representation of components co-evolution; the noise resulted from independent evolution does not affect

Table 9 The adjusted correlation coefficient of 12 hypothesis tests

Hypothesis	*H01*	*H02*	*H03*	*H04*	*H05*	*H06*	*H07*	*H08*	*H09*	*H010*	*H011*	*H012*
Causal component	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
Number of pairs (*N*)	–	9	9	9	–	2	2	4	9	9	9	9
Coefficient (*r*)	–	0.65	0.62	0.98	–	–	1.0	1.0	0.62	0.60	0.63	0.68
Significance level (*p*)	–	0.05	0.1	0.01	–	–	0.01	0.01	0.1	0.1	0.1	0.05

the accuracy of the measurement of evolution coupling; (2) reference coupling played a noticeable role in Linux kernel components co-evolution.

It has been claimed that strong reference coupling has effects on software maintenance. These effects include change dependencies and fault-proneness (Kafura and Henry, 1981; Selby and Basili, 1991; Troy and Zweben, 1981). Our study shows that in Linux kernel modules, changes in one module are likely to result changes in another module due to reference coupling. This is an indirect demonstration of the relationship between reference coupling and software maintenance: strong reference coupling between software components can result components co-evolution, which in turn can bring difficulties for software maintenance.

6.3 The Effect of Independent Evolution

In the above discussion, we assume that the noise to the measurement of evolutionary coupling is random and it does not affect the accuracy of the hypothesis tests. Is the noise really random? How do we assess the effect of independent evolution? In this subsection, we discuss these two questions.

The noise to the measurement of evolutionary coupling is expected to be random is based on the assumption that independent change frequency of different modules is uniform, which means that in each release, the probability that each module is independently changed is same for all the modules. However, this assumption has been shown to be invalid for some systems. For example, Myers (1979) reported that 47% changes due to software faults were associated with only 4% of the modules in OS/370. This means that the fault densities are not same for different modules. Hence, the independent change frequencies of different modules are different. Under this condition, the measurements of transaction count and support count for these frequently independently changed modules will be higher than the measurements of others. This will affect support ratio, the representation of evolutionary coupling.

Another factor that can lead to this uneven distribution of change frequency among modules is the module size. The larger size modules are more likely to be modified than the smaller ones. On one hand, large size module may contain more functions; they are more likely to be maintained in each release. On the other hand, large size modules may be more complex and have more errors, which may require more frequent corrections. Therefore, more transaction counts are expected to be found for large size modules, and this may affect the support count and the support ratio—the measure of evolutionary coupling.

Table 10 shows the size of 12 kernel modules in version 2.4.20 of Linux measured in LOC (lines of code). We tested the correlation between module size and transaction count (column 1 of Table 3). The Spearman's correlation coefficient is 0.839, which is significant at 0.01 level. This shows that in Linux kernel modules, the measure of size is strongly correlated with the measure of the frequency of changes.

Table 10 The module size in version 2.4.20

Module	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
Size (LOC)	129	602	831	171	1283	147	688	233	1360	1301	1289	412

Table 11 The correlation between evolutionary coupling and module size

Causal component	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
Number of pairs (*N*)	11	11	11	11	11	11	11	11	11	11	11	11
Coefficient (*r*)	0.70	0.88	0.87	0.79	0.86	0.82	0.82	0.79	0.79	0.87	0.86	0.82
Significance level (*p*)	>0.1	0.01	0.01	0.05	0.01	0.05	0.05	0.05	0.05	0.01	0.01	0.05

To study the correlation between module size and evolutionary coupling, we use data in Table 4 and Table 10 to construct 12 groups of data. Each group consists of one row data from Table 4 and one row data of the module size from Table 10. In all the 12 tests, each group contains 11 pairs of valid data. The correlation test results are in Table 11, which shows in 12 tests, the significance level for 11 correlations is at 0.05 (less than or equal to 0.05) level. Therefore, we conclude that there is linear relationship between evolutionary coupling and module size.

Now we have a question: "is the strong correlation between module size and evolutionary coupling the result of independent evolution or co-evolution?" Because large size module not only can lead to more frequent independent changes, it may also result more frequent co-changes of components. For example, if two modules have large size and more reference coupling exist between them, they will be changed together more frequently. To answer this question, we need to determine the correlation between reference coupling and module size.

Similar to the study of correlation between evolutionary coupling and module size, we construct 12 groups of data from Table 5 and Table 10. Each group contains 11 pairs of data for reference coupling and module size. The results are in Table 12, which shows that in 12 tests, the significance level for 3 correlations is at 0.1 (less than or equal to 0.1) level. All other correlations are not significant at 0.1 level. This means, we did not find the linear relationship between module size and reference coupling.

To summarize this analysis, we found linear correlation between module size and evolutionary coupling, but we did not find linear relationship between module size and reference coupling. Based on the discussions in Section 3 and description of the relationships among co-evolution, independent evolution, and evolutionary coupling as shown in Fig. 1, we conclude that the size effect on the measurement of evolutionary coupling of Linux is introduced via independent evolution instead of co-evolution. This means, large modules are more likely to be modified independently in Linux.

Table 12 The correlation between reference coupling and module size

Causal component	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
Number of pairs (*N*)	–	11	11	11	–	11	11	11	11	11	11	11
Coefficient (*r*)	–	0.66	0.60	0.61	–	0.45	0.47	0.51	0.56	0.50	0.54	0.59
Significance level (*p*)	–	>0.1	>0.1	>0.1	–	>0.1	>0.1	>0.1	0.1	>0.1	0.1	0.05

In spite of the size effect as independent evolution on evolutionary coupling, our study shows that there exist linear relationship between evolutionary coupling and reference coupling in Linux kernel modules. The noise data (size effect) introduced to evolutionary coupling via independent evolution did not affect us to detect the linear relationship between reference coupling and evolutionary coupling.

6.4 Threats to Validity

There are several threats to the validity of our study. One major construct threat is the accuracy of reference coupling. Our reference coupling data is extracted from one version of Linux. However, in the release of 597 versions of Linux, the reference coupling value could change from version to version. To reduce this threat, we choose version 2.4.20, which is one of the most stable versions. To assess the degree of the threat by choosing version 2.4.20, we randomly selected three other versions of Linux (1.09, 1.2.13, and 2.5.65) and studied their reference coupling. The results show that, for some individual measurements of reference coupling, there are some differences from version to version. However, for a specific item in each group of reference coupling (each row in Table 5), there is few difference in the ranked order from version to version. Table 13 shows such an example for version 2.5.65. If we carefully compare Table 13, which contains reference coupling data of version 2.5.65, with Table 5, which contains reference coupling data of version 2.4.20, we can see that each item in the same row has the same rank.

Because we use Spearman's rank correlation to test the relationship between reference coupling and evolutionary coupling, what matters is the ranked order of reference coupling, but not the value of reference coupling itself. We use reference coupling data of these three versions (1.09, 1.2.13, and 2.5.65) of Linux to test their correlations with evolutionary coupling separately and find that all the correlations are significant at 0.1 level. Therefore, we deduce that even reference coupling changes from version to version for some individual measurements, these changes may not affect the rank correlation of Spearman's test.

The threat to internal validity is the accuracy of data. To reduce this threat, both the evolutionary coupling data and reference coupling data is extracted using automatic tools to avoid the mistakes of manual work. The threat to the statistical validity

Table 13 The reference coupling of Linux version 2.5.65

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12
m1 ➔	–	0	0	0	0	0	0	0	0	0	0	0
m2 ➔	0	–	39	22	0	15	15	31	43	83	125	22
m3 ➔	0	37	–	11	0	3	3	14	42	77	117	10
m4 ➔	0	41	15	–	0	8	19	8	118	76	32	36
m5 ➔	0	0	0	0	–	0	0	0	0	0	0	0
m6 ➔	0	0	1	0	0	–	0	0	1	0	0	0
m7 ➔	0	0	1	0	0	0	–	0	3	0	0	0
m8 ➔	0	2	4	0	0	0	0	–	8	0	1	0
m9 ➔	0	69	45	23	0	15	15	31	–	80	140	25
m10 ➔	0	48	36	22	0	14	14	25	42	–	135	21
m11 ➔	0	84	74	55	0	50	50	61	95	118	–	57
m12 ➔	0	37	27	11	0	3	3	14	31	69	123	–

is the selection of significance level. In our study, we choose $\alpha = 0.1$. This may result a type I error—we mistakenly rejected a null hypothesis. To reduce this threat in the future research, α value should be increased to 0.05 for more accuracy.

The threats to external validity primarily include the subject software and modules are not representative of other software products and other software components. Linux is an open-source software, the management difference between software projects (either open-source or closed source) may result the difference in component evolution. The co-evolution of module may not represent the co-evolution of other software entities, such as functions, classes, or packages. To reduce these threats, more studies should be performed on other systems and other different evolution components in the future.

7 Conclusions and the Future Research

We presented a method to measure the correlation between evolutionary coupling and reference coupling and studied the evolution of 12 Linux kernel modules. The results showed that linear correlation exists between evolutionary coupling and reference coupling in Linux kernel components. The observation in this study can help us understand software components co-evolution, which is relevant to various software engineering fields.

The study of component co-evolution can help understand the software design. Establishing the correlation between reference coupling and evolutionary coupling enables us to detect coupling between software components by investigating their evolution history. For example, the coupling between a front-end application and a backend database schema can be easily extracted from the change history of the product without a very specific analysis, which requires the knowledge of both the database and the application program.

Comparing evolutionary and reference coupling which are determined independently can help improve the software design. A good design should have high cohesion within components and low coupling between components. For example, from the viewpoint of reference coupling, a system is designed with low coupling between components. However, if the analysis of the version history reveals high evolutionary coupling between components, this is an indication of possible targets for restructuring to decrease unnoticeable coupling between components.

The study of evolutionary coupling can also help detect coupling between non-program entities, such as requirement specification, design specification, and testing plan. This is especially important for the maintenance of legacy systems. The lack of full documentation is the problem of most legacy systems. Extract software dependencies from available documents can help recover original analysis and design in order to reengineer the legacy systems.

Understand coupling is also important for successful software maintenance. Evolutionary coupling has been used to predict future changes of software components, and this is mainly based on the unquantified evidence that two components are usually changed at the same time. Strong correlation between reference coupling and evolutionary coupling provide the statistical support for these predictions.

The study of component co-evolution is also beneficial to software testing, especially regression testing. When changes are made to one component, other components that have strong reference coupling or evolutionary coupling should be tested to ensure the changes do not introduce regression faults in other components.

Based on the current result, in the future, we plan for the following research:

- Study component co-evolution in shorter period. In the research presented in this paper, we examined the evolutions of 12 modules in 597 releases. In the future, we plan to divide these 597 releases into shorter release spans, in which, more co-evolved kernel modules could be selected. We will investigate the correlation between evolutionary coupling and reference coupling in these shorter evolution periods. Within a shorter release span, fewer changes in reference coupling are expected to be found between different versions.
- Study the co-evolution of nonkernel modules. Linux is designed for many different hardware architectures. For different architectures, there are different implementations of the same modules. These modules in different architecture also have different evolution histories. We will study the co-evolution of the same set of modules that belong to different architecture and have different evolution coupling and reference coupling.
- Study the evolution coupling in non-program entities. We will perform a detailed analysis of change log (including bug report) to extract change dependencies between components, because change log can provide additional information about particular modifications made to components. We will determine a method to measure change dependency based on change log and study its correlation with evolutionary coupling, which is based on version history, and reference coupling, which is based on system architecture. This study can help us to understand more deeply about the change dependencies between software components.

Acknowledgments The author would like to thank the editor and the anonymous reviewers for their valuable feedback on an earlier version of this paper.

References

Arisholm E, Briand LC, Foyen A (2004) Dynamic coupling measurement for object-oriented software. IEEE Trans Softw Eng 30(8):491–506

Arthur LJ (1988) Software evolution: the software maintenance challenge. Wiley, New York, New York, USA

Briand LC, Wuest J (2002) Empirical studies of quality models in object-oriented systems. Adv Comput 59:97–166

Briand LC, Daly JW, Wust JK (1999) A unified framework for coupling measurement in object-oriented systems. IEEE Trans Softw Eng 25(1):91–121

Chidamber SR, Kemerer CF (1994) A metrics suite for object-oriented design. IEEE Trans Softw Eng 20(6):476–493

Gall H, Hajek K, Jazayeri M (1998) Detection of logical coupling based on product release history. Proceedings of the 14th International Conference on Software Maintenance, Bethesda, Maryland, USA, pp 190–198

Godfrey MW, Tu Q (2000) Evolution in open source software: a case study. Proceedings of International Conference on Software Maintenance, San Jose, California, pp 131–142

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661

Hassan AE, Holt RC (2004) Predicting change propagation in software systems. Proceedings of the 20th International Conference on Software Maintenance, Chicago Illinois, USA, pp 284–293

Kafura D, Henry S (1981) Software quality metrics based on interconnectivity. J Syst Softw 2(2):121–131

Lehman MM (1980) Life cycles and laws of software evolution. Proceedings of IEEE (Special Issue on Software Engineering), pp 1060–1076

Myers GJ (1979) The art of software testing. Wiley, New York

Nolan B (1994) Data analysis, an introduction. Polity, Cambridge Massachusetts

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

Perry DE (1994) Dimensions of software evolution. Proceedings of International Conference on Software Maintenance, Sorrento, Italy, pp 296–303

Raymond ES (2001) The Cathedral & the Bazaar, 1st edn. O'Reilly

Schach SR, Jin B, Wright DR, Heller GZ, Offutt AJ (2002) Maintainability of the Linux kernel. IEE Proc, Softw 149:18–23

Schach SR, Jin B, Wright DR, Heller GZ, Offutt J (2003) Quality impacts of clandestine common coupling. Softw Qual J 11:211–218

Selby RW, Basili VR (1991) Analyzing error-prone system structure. IEEE Trans Softw Eng 17(2):141–152

Stevens WP, Myers GZ, Constantine LL (1974) Structured design. IBM Syst J 13(2):115–139

Troy DA, Zweben SH (1981) Measuring the quality of structured design. J Syst Softw 2(2):113–120

Weißgerber P, Klenze L, Burch M, Diehl S (2005) Exploring evolutionary coupling in eclipse. Eclipse technology exchange workshop, San Diego, California

Williams CC, Hollingsworth JK (2005) Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans Softw Eng 31(6):466–480

Xing Z, Stroulia E (2004) Data-mining in support of detecting class co-evolution. Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering, Banff, Alberta, Canada, 123–128

Xing Z, Stroulia E (2005) Analyzing the evolutionary history of the logical design of object-oriented software. IEEE Trans Softw Eng 31(10):850–868

Ying ATT, Ng R, Chu-Carroll MC, Murphy GC (2004) Predicting source code changes by mining change history. IEEE Trans Softw Eng 30(9):574–586

Yu L, Schach SR, 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

Zimmermann T, Diehl S, Zeller A (2003) How history justifies system architecture (or not). Proceedings of the 6th International Workshop on Principles of Software Evolution, Helsinki, Finland, pp 73–83

Zimmermann T, Weißgerber P, Diehl S, Zellers A (2004) Mining version histories to guide software changes. Proceedings of the 26th International Conference on Software Engineering, Scotland, UK, pp 563–572

Zimmermann T, Weißgerber P, Diehl S, Zellers A (2005) Mining version histories to guide software changes. IEEE Trans Softw Eng 31(6):429–445

Liguo Yu received the Ph.D. 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, software evolution, and open-source software development. Before working in software engineering, his research focused on modeling and system identification, and fault detection and isolation of hybrid systems.