

Change Propagations in the Maintenance of Kernel-Based Software with a Study on Linux

Liguo Yu
Computer Science and Informatics
Indiana University South Bend
1700 Mishawaka Ave. P.O. Box 7111
South Bend, IN 46634, USA
ligyu@iusb.edu

Srini Ramaswamy
Computer Science Department
University of Arkansas at Little Rock
2801 S. University Avenue
Little Rock, AR 72204, USA
sxramaswamy@ualr.edu

ABSTRACT

As a software system evolves to accommodate new features and repair bugs, changes are needed. Software components are interdependent, changes made to one component can require changes to be propagated to other components. Change propagation brings potential challenges for software maintenance. In this paper, we divide change propagations into four categories in kernel-based software. Different categories of change propagation have different effects on kernel maintenance. We use product version history to mine change propagations rules and apply the categorization to Linux operating system. Our study provides a framework for measuring, evaluating, and predicting change propagations in kernel-based software, which includes most operating systems, database management systems, game systems, and software product lines.

Categories and Subject Descriptors

D.2.7.g [Maintainability], D.2.8 [Metrics/ Measurement], D.2.10.h [Quality analysis and evaluations].

General Terms

Design.

Keywords

Change Propagation, data mining, maintenance.

1. INTRODUCTION

Software components are interdependent. Changes to one component can result in changes propagated to other components; this is called change propagation [1]. Because change propagations make software maintenance difficult and time-consuming, understanding how changes propagate between components can improve our ability to maintain the software system. For example, if the signature of a function is changed, the

calling functions have to be modified to reflect this change. Insufficient or incorrect changes can lead to software errors, which may result in system failures. Therefore, the prediction, control, and testing of change propagations are important factors that determine a successful software maintenance activity. Change propagations between software components can be studied by looking at the version history of the product. This is based on the observation that if two components are frequently changed together in the earlier release, they are more likely to be changed together in the later release.

Operating systems consist of a kernel and other nonkernel components [2]. Kernel components form the core part; it is responsible for providing secure multiplexing and arbitration of a machine's hardware and interfaces the hardware with the high level application software. Nonkernel components provide I/O operations, file management, and architecture or driver dependent tasks. Besides operating systems, other kernel-nonkernel structured software includes database management systems [3] and game systems [4]. In this paper, we refer to software that is comprised of a kernel together with optional nonkernel components as kernel-based software systems. Figure 1 shows the composition of kernel-based software: common kernel components integrated with customized nonkernel components produce various products. A major extension of kernel-based software is a software product line [5], in which kernel components are called core assets and nonkernel components are called custom assets.

In general software systems, all components are equally important; there is no difference between these components for change propagations. However, in kernel-based software, the kernel is expected to be reused in different architectures or different applications. Hence, kernel components should be more stable, reliable, and reusable than nonkernel components. Therefore, change propagations within kernel components, nonkernel components, and between kernel and nonkernel components have different implications on software quality, such as maintainability and reusability. In this paper, we divide change propagations in kernel-based software into four categories and analyze their different effects on software maintenance. In an application of this categorization, Linux is studied to understand how change propagations may affect its maintenance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACMSE 2007, March 23–24, 2007, Winston-Salem, North Carolina, USA. ©Copyright 2007 ACM 978-1-59593-629-5/07/0003...\$5.00.

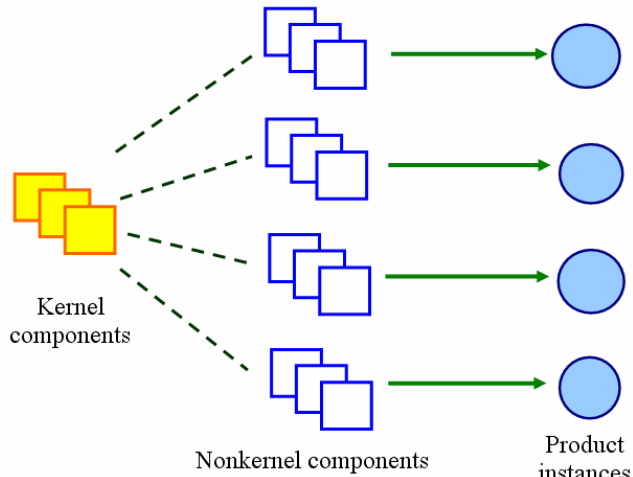


Figure 1. Description of kernel-based software.

The remainder of the paper is organized as follows: Section 2 discusses change propagations and related work. Section 3 describes the measure of change propagation. Section 4 describes the categorization of change propagations in kernel-based software. Section 5 presents our case study on Linux. Our conclusions appear in Section 6.

2. CHANGE PROPAGATIONS AND RELATED WORK

Coupling [6] is a measure of the degree of interactions between software components (modules, classes, packages, and so on) and, hence, of the dependency between these components. If there is no coupling at all in a software system, the system would consist of one large component. Therefore, some amount of coupling clearly is needed. Hence, coupling is a necessary consequence of modularization. However, where there is coupling between two components, there is some degree of dependence between those components.

Too many dependencies between software components will make a software system fault prone, difficult to reuse, and difficult to maintain. One such effect of component dependency is change propagation: changes made to one component require corresponding changes in other components. Change propagation is an important issue for software maintenance: (1) change propagations increase the programmer effort in understanding the effect of changes; (2) change propagations increase programmer effort in testing changes to ensure no regression faults are introduced in the system; (3) insufficient or incorrect change propagations may introduce errors that cause system failures. Change propagations are also important in software reuse: a reusable component should be relatively independent; changes to other components should not result in changes to the frequently reused component.

Due to the importance of change propagations, some research has been done to understand, predict, and control change propagations. Zimmermann et al. [7] [8] [9] applied data mining to software version history and implemented a tool to detect related changes in maintenance. The tool can help programmers predict likely future modifications and prevent errors due to

incomplete changes. Ying et al. [10] applied data mining techniques on the change history data of source code and determined association rules of change patterns (sets of files that were changed together). The association rules can be used to predict future co-changes of software components due to interdependence. Hassan and Holt [1] proposed several heuristics to predict change propagations among components and presented a framework to measure the performance of the proposed heuristics. Some heuristics are based on historical co-changes; some are based on static analysis of component interdependencies.

Change propagation is also used in software quality control. Graves et al. [11] used change history data to successfully predict the distribution of incidences of faults. Their studies show that the change patterns found based on version history can be used to recommend potentially relevant changes and predict possible fault locations to a developer performing a maintenance task. Williams and Hollingsworth [12] used the source code change history to find change propagation patterns. These patterns can help programmer search for bugs. The studies showed that their bug-finding technique is more effective than the same static analysis that does not use the change history data from the source code repository.

3. THE MEASURE OF CHANGE PROPAGATION

The change propagation between software components can be determined and measured from the version history of the software system. Here we adapt the methods proposed by Zimmermann et al. [7] [8] [9]. Suppose a software system has n consecutive versions V_1 through V_n , each new release is based on the previous version. To study the change propagation between two components C_i and C_j of this software system, we use the notation $C_i \Rightarrow C_j$ to indicate the *association rule* that changes to C_i result in changes to C_j in the same release. In a system that contains m components, there exist $m^*(m-1)$ possible association rules $C_i \Rightarrow C_j$ ($i=1..m, j=1..m, i \neq j$). Obviously, such association rules are not always true. However, they have a probabilistic meaning, which is dependent on three measures:

- **Transaction count.** The transaction count is the number of changes made on C_i in these n releases. Assume C_i was modified in 10 releases, the transaction count is 10.
- **Support count.** The support count is the number of changes made on C_j while C_i is changed. Assume, in 9 releases, changes are made to both C_i and C_j . Therefore, the support count for the association rule $C_i \Rightarrow C_j$ is 9.
- **Confidence.** The confidence indicates the strength of the association rule, or the relative amount of the given consequences. It is represented as support count / transaction count. Hence, the confidence for the above association rule $C_i \Rightarrow C_j$ is $9/10 = 0.9$.

As with other research, we use confidence of an association rule to represent the probability of change propagation. It should be noted that (1) support count does not completely (i.e. 100 percent) represent the change propagation between two components. The co-changes made to two components may be accidental and have no propagation relations; (2) change propagation association is

directed; i.e., $C_i \Rightarrow C_j$ and $C_j \Rightarrow C_i$ have different meanings. The former association rule states that changes made to C_i result in changes to C_j , while the later association rule states that changes made to C_j result in changes to C_i . Consequently, the confidences for the two rules could be dramatically different.

4. THE CATEGORIZATION OF CHANGE PROPAGATIONS IN KERNEL-BASED SOFTWARE

As described in Section 1, in kernel-based software, there are two kinds of components, kernel components and nonkernel components. Based on the different effects on kernel maintenance, we divide change propagations into four categories:

- Category-1: change propagation is within nonkernel components;
- Category-2: change propagation is from kernel to nonkernel components;
- Category-3: change propagation is within kernel components;
- Category-4: change propagation is from nonkernel to kernel components.

Category-1 change propagation can probably be considered the least objectionable with respect to the maintenance of kernel components. There is no change dependency between kernel components and nonkernel components, so a modification to a nonkernel component cannot induce a regression fault in a kernel component. Accordingly, category-1 change propagation has no impact on the kernels.

As with category-1, category-2 change propagation cannot affect kernels either, because the change propagation is from kernel to nonkernel components. Software components must interact with each other; this applies to both kernel components and nonkernel components. Category-2 change propagation means that nonkernel components are dependent on kernel components. This propagation might affect the maintenance of nonkernel components, but it won't affect the maintenance of kernel components.

Category-3 change propagation is within kernel components. In the design of kernel-based software, each kernel component should be relatively independent with respect to both nonkernel components and other kernel components. Category-3 change propagations make the kernel components vulnerable to changes in other kernel components. On the other hand, category-3 change propagations also bring difficulties for a single kernel component reuse. If a kernel component is dependent on other kernel components, we cannot reuse it without incorporating all of these dependent components.

Category-4 change propagation is from nonkernel components to kernel components. The principle of "separation of concerns" tells us that changes to nonkernel components should not affect kernel components. Therefore, category-4 change propagation is the most unfavorable in a software system. That is, kernel-based software consisting of category-4 change propagations is vulnerable to modifications to a nonkernel component. From a maintenance perspective, it is more laborious and time-consuming to maintain a product that involves category-4 change propagations.

5. LINUX CASE STUDY

The open-source software development life-cycle model can best be described as continuous maintenance, as encapsulated in the dictum "release early and often" [13]. 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 a kernel-based software system and is one of the most widely used open-source operating systems. (2) Linux is one of the most active open-source projects. It has released about 600 versions, which provides us with a rich source of version history data to test the effectiveness of our change propagation categorization scheme.

Linux is designed for more than ten different hardware architectures. In this paper, five of the most commonly used architectures are chosen for our study. They are *alpha*, *i386*, *mips*, *sparc*, and *ppc*. From the first version 1.0.0 to the latest version 2.6.10, major changes have been made to Linux. Some components have been added, while others have been deleted. We select 10 kernel components that experienced the entire evolution of Linux in our study. Similarly, we select 6 long-evolved nonkernel components that are implemented in all five architectures for our study. These components are depicted in Figure 2. The left package contains kernel components; the right packages contain nonkernel components, which are implemented differently in different architectures. Each installation of Linux contains common kernel components together with the nonkernel components for the corresponding architecture.



Figure 2. The Linux components selected for study.

From version 2.0.0 to version 2.5.75, there are 362 formal releases. For each of the 362 versions, we determined whether the corresponding kernel component or nonkernel component was modified compared to the previous version. Each modification is considered as a transaction. We use the representation described in Section 3 to measure the change propagations.

5.1 Category-1 Change Propagation

We studied change propagations within nonkernel components for all five architectures. Given the space constraint, we use *i386* to illustrate the result of our study. For the selected nonkernel components in *i386*, there are total $5*6=30$ association rules ($m_i \Rightarrow m_j$, $i=1..6$, $j=1..6$, $i \neq j$). Table 1 shows the support count for different rules. The first column is the transaction count for each component. The second column through the last column shows the support count for every association rule. The value in row m_i and column m_j is the support count for association rule $m_i \Rightarrow m_j$ (changes to m_i result changes in m_j in the same release). For example, in Table 1, the transaction count for m_1 is 97. The support count for association rule $m_1 \Rightarrow m_2$ is 45. It should be noted that the support count in Table 1 is symmetric. Because support count measures the number of times m_i and m_j are modified in the same release, the values for rule $m_i \Rightarrow m_j$ and rule $m_j \Rightarrow m_i$ should be same.

Table 2 shows the confidences derived for the various association rules. The value in row m_i and column m_j is the confidence for evolutionary rule $m_i \Rightarrow m_j$ ($i \neq j$). As stated before, it is calculated by dividing the transaction count of m_i (item in row m_i and column 1) with the corresponding support count of association rule $m_i \Rightarrow m_j$ (item in row m_i and column m_j) in Table 1.

Table 1. The support count of category-1 change propagations in nonkernel components of *i386*

	m1	m2	m3	m4	m5	m6
m1 (97)	–	45	18	48	18	32
m2 (115)	45	–	28	59	38	35
m3 (52)	18	28	–	27	30	11
m4 (134)	48	59	27	–	40	39
m5 (68)	18	38	30	40	–	16
m6 (71)	32	35	11	39	16	–

Key to components: *m1: irq.c; m2: process.c; m3: ptrace.c; m4: setup.c; m5: signal.c; m6: time.c*

Table 2. The confidence of category-1 change propagations in nonkernel components of *i386*

	m1	m2	m3	m4	m5	m6
$m_1 \Rightarrow$	–	0.46	0.19	0.49	0.19	0.33
$m_2 \Rightarrow$	0.39	–	0.24	0.51	0.33	0.30
$m_3 \Rightarrow$	0.35	0.54	–	0.52	0.58	0.21
$m_4 \Rightarrow$	0.36	0.44	0.20	–	0.30	0.29
$m_5 \Rightarrow$	0.26	0.56	0.44	0.59	–	0.24
$m_6 \Rightarrow$	0.45	0.49	0.15	0.55	0.23	–

Key to components: *m1: irq.c; m2: process.c; m3: ptrace.c; m4: setup.c; m5: signal.c; m6: time.c*

The confidence for $m_i \Rightarrow m_j$ ($i \neq j$) is the probability that changes made to m_i result changes in m_j . We studied nonkernel components of five architectures (*alpha*, *i386*, *mips*, *sparc*, and *ppc*). Each architecture contains 30 association rules. To test the accuracy of the prediction, we applied all $30*5=150$ association rules to version 2.6.0 through version 2.6.10: if m_i is changed in a specific version and m_j is also changed, this is counted as an accurate prediction of rule $m_i \Rightarrow m_j$; if m_i is changed while m_j is not changed, this is counted as an inaccurate prediction of

$m_i \Rightarrow m_j$. Table 3 shows the average prediction accuracy for different range of confidences. The prediction accuracy is calculated using number of accurate prediction divided by number of total predictions. We can see that larger confidence association rules are more predictable than ones with smaller confidence association rules.

Table 3. The average prediction accuracy of category-1 change propagations

Confidence (r) range	$r < 0.2$	$0.2 \leq r < 0.4$	$0.4 \leq r < 0.6$	$0.6 \leq r$
Prediction accuracy	0.11	0.36	0.54	0.68

5.2 Category-3 Change Propagation

Table 4 and Table 5 show the support count and confidence of the category-3 change propagation that occurred in kernel. The average prediction accuracy is in Table 6. The result indicates that confidences, especially larger confidences, are good predictors of component co-changes.

Table 4. The support count of category-3 change propagations in kernel

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
m1 (11)	–	6	3	0	2	2	4	1	5	0
m2 (109)	6	–	78	9	32	19	26	18	50	23
m3 (143)	3	78	–	6	40	23	42	24	57	22
m4 (15)	0	9	6	–	8	7	7	2	10	9
m5 (65)	2	32	40	8	–	15	21	8	24	15
m6 (38)	2	19	23	7	15	–	17	5	18	11
m7 (63)	4	26	42	7	21	17	–	11	26	14
m8 (30)	1	18	24	2	8	5	11	–	17	7
m9 (90)	5	50	57	10	24	18	26	17	–	19
m10 (38)	0	23	22	9	15	11	14	7	19	–

Key to components: *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: signal.c; m10: time.c*

Table 5. The confidence of category-3 change propagations in kernel

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
$m_1 \Rightarrow$	–	0.55	0.27	0.00	0.18	0.18	0.36	0.09	0.45	0.00
$m_2 \Rightarrow$	0.06	–	0.72	0.08	0.29	0.17	0.24	0.17	0.46	0.21
$m_3 \Rightarrow$	0.02	0.55	–	0.04	0.28	0.16	0.29	0.17	0.40	0.15
$m_4 \Rightarrow$	0.00	0.60	0.40	–	0.53	0.47	0.47	0.13	0.67	0.60
$m_5 \Rightarrow$	0.03	0.49	0.62	0.12	–	0.23	0.32	0.12	0.37	0.23
$m_6 \Rightarrow$	0.05	0.50	0.61	0.18	0.39	–	0.45	0.13	0.47	0.29
$m_7 \Rightarrow$	0.06	0.41	0.67	0.11	0.33	0.27	–	0.17	0.41	0.22
$m_8 \Rightarrow$	0.03	0.60	0.80	0.07	0.27	0.17	0.37	–	0.57	0.23
$m_9 \Rightarrow$	0.06	0.56	0.63	0.11	0.27	0.20	0.29	0.19	–	0.21
$m_{10} \Rightarrow$	0.00	0.61	0.58	0.24	0.39	0.29	0.37	0.18	0.50	–

Key to components: *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: signal.c; m10: time.c*

Table 6. The average prediction accuracy of category-3 change propagations

Confidence (r) range	$r < 0.2$	$0.2 \leq r < 0.4$	$0.4 \leq r < 0.6$	$0.6 \leq r$
Prediction accuracy	0.09	0.30	0.55	0.74

5.3 Category-2 and Category-4 Change Propagation

Category-2 change propagation is from kernel components to nonkernel components; category-4 change propagation is from nonkernel components to kernel components. We select 3 components to study change propagations between kernel and nonkernels. They are `ptrace.c`, `signal.c`, and `time.c`. These three components are special because they are implemented in both kernel and nonkernel components (Figure 2): the kernel components are same for all the architectures, the nonkernel components are architecture dependent. In other words, the architecture independent functions of `ptrace.c`, `signal.c`, and `time.c` are implemented in the kernel component; the additional architecture specific functions are implemented in the nonkernel component. We expect to find change propagations between each pair of `ptrace.c`, `signal.c`, and `time.c` in kernel and nonkernel components, because each pair of `ptrace.c`, `signal.c`, and `time.c` is logically related. We use *i386* to illustrate such change propagations, which is shown in Figure 3. The arrows pointing from kernel to nonkernel indicate category-2 change propagations; the arrows pointing from nonkernel to kernel indicate category-4 change propagations. Certainly, change propagations can occur between different components in kernel and nonkernel component. However, in this study we decide to study category-2 and category-4 change propagations using logically related components in kernel and nonkernel components.

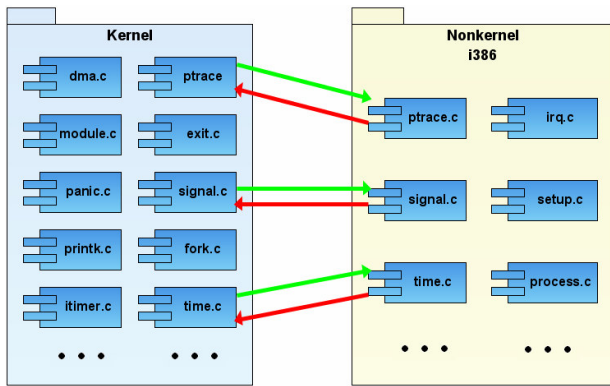


Figure 3. The studied category-2 and category-4 change propagations in *i386*.

Table 7 shows the confidence of category-2 change propagation association rules of `ptrace.c` (kernel) \Rightarrow `ptrace.c` (nonkernel), `signal.c` (kernel) \Rightarrow `signal.c` (nonkernel), and `time.c` (kernel) \Rightarrow `time.c` (nonkernel). For different architectures, the kernel contains the same components, while the nonkernel contains different components. Table 7 shows that different instances of Linux for different architectures evolved differently. For example, if a change is made to kernel component `ptrace.c`, the probability that the nonkernel component `ptrace.c` needs to be changed is 0.33 for *i386* and 0.18 for *mips*.

Table 7. The confidence of category-2 change propagation

Association rule	<i>alpha</i>	<i>i386</i>	<i>mips</i>	<i>sparc</i>	<i>ppc</i>
<code>ptrace.c</code> (kernel \Rightarrow nonkernel)	0.27	0.33	0.18	0.30	0.39
<code>signal.c</code> (kernel \Rightarrow nonkernel)	0.30	0.40	0.18	0.30	0.25
<code>time.c</code> (kernel \Rightarrow nonkernel)	0.21	0.45	0.14	0.31	0.26

Table 8 shows the confidence of category-4 change propagation association rules of `ptrace.c` (nonkernel) \Rightarrow `ptrace.c` (kernel), `signal.c` (nonkernel) \Rightarrow `signal.c` (kernel), and `time.c` (nonkernel) \Rightarrow `time.c` (kernel). It shows that component `signal.c` induces high confidence of category-4 change propagations: there are over 50 percent possibilities that changes made to nonkernel component `signal.c` in any architecture may result in changes to the kernel component `signal.c`. However, as discussed in Section 4, category-4 propagation is the most unfavorable category. To reduce the effect of category-4 change propagations on kernel maintenance, `signal.c` is a potential component that needs to be restructured to reduce its dependency on nonkernel components.

Table 8. The confidence of category-4 change propagation

Association rule	<i>alpha</i>	<i>i386</i>	<i>mips</i>	<i>sparc</i>	<i>ppc</i>
<code>ptrace.c</code> (nonkernel \Rightarrow kernel)	0.23	0.20	0.22	0.40	0.34
<code>signal.c</code> (nonkernel \Rightarrow kernel)	0.59	0.53	0.50	0.52	0.50
<code>time.c</code> (nonkernel \Rightarrow kernel)	0.18	0.25	0.21	0.35	0.22

5.4 Discussions

We proposed a categorization scheme for studying change propagations and studied its effectiveness using Linux. Table 9 summarizes the results. A larger confidence means higher probability of change propagations. Category-1 and category-2 change propagations have no effect on kernel maintenance. Table 9 shows that all the confidences for category-1 and category-2 association rules are less than 0.6, which means that weaker category-1 and category-2 change propagations do exist in Linux. On the contrary, larger values of confidence of category-3 and category-4 change propagation rules are also found. Category-3 change propagations are between kernel components; these change propagations are localized in kernel components and they won't affect the reuse of the entire kernel. However, category-3 change propagation affects the reuse of a single kernel component.

Table 9 shows a 33 percent of category-4 change propagations have confidence between 0.4 and 0.6. This indicates potential difficulties in the maintenance of Linux: changes to nonkernel components require changes to kernel components; reuse of a kernel component needs to consider its dependence on nonkernel components.

Table 9. Summary of the change-propagations in Linux

Confidence (r)	Category-1	Category-2	Category-3	Category-4
$r < 0.2$	10%	20%	37%	7%
$0.2 \leq r < 0.4$	47%	67%	30%	60%
$0.4 \leq r < 0.6$	43%	13%	21%	33%
$0.6 \leq r$	0%	0%	12%	0%

As the most widely used open-source operating system, Linux is designed with kernel-based structure to support various hardware architectures. Our study of change propagations show that some Linux kernel components have strong dependencies on nonkernel components and other kernel components, which result in change propagations within kernel components and from nonkernel components to kernel components. These change propagations represent potential obstacles in kernel maintenance.

It is worth noting that (1) the association rules studied in this research do not represent causality. They only provide empirical evidence of co-changes between software components; (2) Linux contains about 30 kernel components and 9000 nonkernel components. Our study represents the change propagations within a small set of Linux components. To obtain a deeper understanding of change propagations, more components should be selected to study. However, the major objective of our work reported in this paper is to show how to measure and interpret change propagations in kernel-based software and how these measurements can be used effectively to predict future component co-changes. Our framework is simple and attractive to use for studying maintenance issues in kernel-based systems, such as operating systems, database systems, game systems, and software product lines.

6. CONCLUSIONS

In this paper, we divided change propagation in kernel-based software into four categories and discussed why different categories have different effects on kernel maintenance. We used product version history to measure change propagations and apply it on a case study of Linux.

Our categorization of change propagation can help us with the following:

- understand the dependency of kernel components;
- understand the evolution of kernel-based software;
- target kernel component that needs to be restructured in order to reduce its dependence on other kernel or nonkernel components.

In this study, we use product version history to mine component change propagation rules. In the future, we plan to study change propagations using non-program entities. We will perform a detailed analysis of change log (including bug report) to extract change propagations between components, because change log can provide additional information about particular modifications made to components. We will measure change propagations based on change log and compare with the measure based on version history.

We hope this empirical study help software organizations to understand change propagations between software components and thereby follow effective software development and change control practices for software product releases.

7. ACKNOWLEDGEMENTS

This work is supported partly by the Indiana University South Bend Faculty Research Grant (2007).

8. REFERENCES

- [1] Hassan, A. E. and Holt, R. C., "Predicting Change Propagation in Software Systems", Proceedings of the 20th International Conference on Software Maintenance, Chicago Illinois, USA, 2004: 284–293.
- [2] Hansen, P. B., "The Nucleus of a Multiprogramming System", Communications of the ACM, 1970, 4(4): 238–241.
- [3] Härden, T., "New Approaches to Object Processing in Engineering Databases", Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, 1986: 217–217.
- [4] Xbox365, 2006. Xbox system software overview.
- [5] Clements, P. and Northrop, L., Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.
- [6] Stevens, W. P., Myers, G. Z., and Constantine, L. L., "Structured Design", IBM Systems Journal, 1974, 13(2): 115–139.
- [7] Zimmermann, T., Diehl, S., and Zeller, A., "How History Justifies System Architecture (or not)", Proceedings of the 6th International Workshop on Principles of Software Evolution, Helsinki, Finland, 2003: 73–83.
- [8] Zimmermann, T., Weißgerber, P., Diehl, S., and Zellers, A., "Mining Version Histories to Guide Software Changes", Proceedings of the 26th International Conference on Software Engineering, Scotland, UK, 2004: 563–572.
- [9] Zimmermann, T., Weißgerber, P., Diehl, S., and Zellers, A., "Mining Version Histories to Guide Software Changes", IEEE Transactions on Software Engineering, 2005, 31(6): 429–445.
- [10] Ying, A. T. T., Ng, R., Chu-Carroll, M. C., and Murphy, G. C., "Predicting Source Code Changes by Mining Change History", IEEE Transactions on Software Engineering, 2004, 30(9): 574–586.
- [11] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H., "Predicting Fault Incidence Using Software Change History", IEEE Transactions on Software Engineering, 2000, 26(7): 653–661.
- [12] Williams, C. C. and Hollingsworth, J. K., "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques", IEEE Transactions on Software Engineering, 2005, 31(6): 466–480.
- [13] Raymond, E. S., The Cathedral & the Bazaar, First Edition, O'Reilly, 2001.